



Norwegian University of
Science and Technology

Robotic Assembly Using 3D and 2D Computer Vision

Asgeir Bjørkedal
Kristoffer Larsen

Subsea Technology

Submission date: June 2016

Supervisor: Olav Egeland, IPK

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

MASTEROPPGAVE 2016

Asgeir Bjørkedal & Kristoffer Larsen

Tittel: Robotisert montasje ved bruk av et «computer vision» system bestående av 3D og 2D kameraer.

Tittel (engelsk): Robotic assembly using 3D and 2D computer vision.

Oppgavens tekst:

Et håndmontert kamera er nyttig i montasjeoppgaver for å få god bildeinformasjon for innjustering av montasjeoperasjoner. Et håndmontert kamera har i tillegg den fordelen at det kan automatisk rettes inn mot montasjeoppgaven. I denne oppgave skal dette kombineres med et stasjonært 3D-kamera som skal gi en grov, men pålitelig posisjonsbestemmelse av deler, mens det håndholdte kameraet skal gi en nøyaktig posisjonsbestemmelse. Systemet skal testes ut i instituttets Agilus-lab.

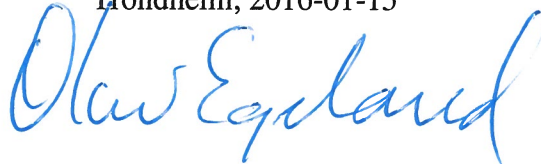
1. Beskriv kinematikken av en robotcelle bestående av et fastmontert 3D-kamera og et håndmontert 2D-kamera.
2. Lag et system for å detektere deler ved bruk av et håndmontert 2D-kamera.
3. Lag et system for å detektere 3D modellerte deler ved hjelp av 3D-kamera.
4. Implementer et system som drar nytte av både 3D og 2D bildebehandling for nøyaktig posisjon-estimering av ønskede deler.
5. Gjennomfør et praktisk eksperiment hvor detekteringsystemet benyttes for montering av kjente deler.

Oppgave utlevert: 2016-01-15

Innlevering: 2016-06-10

Utført ved Institutt for produksjons- og kvalitetsteknikk, NTNU.

Trondheim, 2016-01-15



Professor Olav Egeland
Faglærer

Preface

This thesis is the result of a Master's project in the *Subsea Technology* study programme at NTNU Trondheim. The problem description was written by Professor Olav Egeland and the project work was carried out in a two person group where the work was divided equally. The time period for this project was the spring semester of 2016 (January to June 2016). It is assumed that the reader possesses some basic knowledge regarding robotic systems, 2D and 3D computer vision.

Trondheim, 2016-06-08



Kristoffer Larsen



Asgeir Bjørkedal

Acknowledgment

We would like to thank the following persons for their great help during this Master's thesis.

Professor Olav Egeland for constructing a problem description that turned out to be both interesting and challenging, as well as providing guidance during the project work.

Ph.D candidate Adam Leon Kleppe for giving us valuable feedback and guidance during the project work.

Their help during the project contributed to steady progress, and we are thankful for their insight in the subject of Production Technology.

We would also like to thank our fellow students for providing a social and positive work environment.

K.L & A.B

Abstract

The content of this thesis concerns the development and evaluation of a robotic cell used for automated assembly. The automated assembly is made possible by a combination of an eye-in-hand 2D camera and a stationary 3D camera used to automatically detect objects. Computer vision, kinematics and programming is the main topics of the thesis. Possible approaches to object detection has been investigated and evaluated in terms of performance. The kinematic relation between the cameras in the robotic cell and robotic manipulator movements has been described. A functioning solution has been implemented in the robotic cell at the *Department of Production and Quality Engineering* laboratory.

Theory with significant importance to the developed solution is presented. The methods used to achieve each part of the solution is anchored in theory and presented with the decisions and guidelines made throughout the project work in order to achieve the final solution.

Each part of the system is presented with associated results. The combination of these results yields a solution which proves that the methods developed to achieve automated assembly works as intended. Limitations, challenges and future possibilities and improvements for the solution is then discussed.

The results from the experiments presented in this thesis demonstrates the performance of the developed system. The system fulfills the specifications defined in the problem description and is functioning as intended considering the instrumentation used.

Sammendrag

Innholdet i denne avhandlingen dreier seg rundt utviklingen og evalueringen av en robotcelle for automatisert montering. Den automatiserte monteringen blir muliggjort gjennom en kombinasjon av et håndmontert 2D kamera og et stasjonært 3D kamera brukt for automatisk detektering av objekter. Datasyn, kinematikk og programmering er hovedtemaene i denne avhandlingen. Mulige fremgangsmåter for objekt-detektering har blitt undersøkt og evaluert i forhold til ytelse. Den kinematiske sammenhengen mellom kameraene i robotcellen og de robotiserte manipulatorene blir presentert. En fungerende løsning implementert i verkstedet ved NTNU Trondheims *Institutt for produksjons- og kvalitetsteknikk* blir presentert.

Teori med betydningsfull verdi for løsningen er presentert. Videre er metodene som er benyttet for å oppnå hver del av den endelige løsningen forankret i teori og presentert sammen med avgjørelser og retningslinjer som har blitt bestemt gjennom arbeidet med oppgaven for å kunne nå den endelige løsningen.

Hver del av det utviklede systemet er presentert sammen med tilhørende resultat. Ved å kombinere disse resultatene oppnåes en løsning som beviser at de utviklede metodene for å oppnå automatisert montering fungerer som tiltenkt. Begrensninger, utfordringer, fremtidige muligheter og forbedringer for løsningen blir deretter diskutert.

Resultatene fra eksperimentene utført gjennom denne avhandlingen blir presentert. Disse resultatene demonstrerer ytelsen til det ferdige systemet. Systemet oppfylder spesifikasjonene som er definert i problembeskrivelsen og fungerer som forventet tatt i betraktning instrumenteringen som blir benyttet for løsningen.

Contents

Preface	i
Acknowledgment	ii
Abstract	iv
Sammendrag	v
Terms and Abbreviations	xviii
Materials and Software	xx
1 Introduction	1
1.1 Background	1
1.2 Problem description	1
1.3 Thesis structure	2
2 Theory	3
2.1 Kinematics	3
2.1.1 Orientation	3
2.1.2 Transformation	4
2.1.3 Denavit-Hartenberg convention	4
2.1.4 Forward kinematics	6
2.1.5 Inverse kinematics	6
2.1.6 2D computer vision	7
2.2 3D computer vision	8
2.2.1 Time of flight	8
2.2.2 Structured light	9
2.3 Processing point clouds	10
2.3.1 Passthrough filtering	10
2.3.2 Voxel grid filtering	10
2.3.3 Bilateral filtering	11
2.3.4 Outlier removal filtering	12
2.3.5 Model segmentation	12
2.3.6 Cluster extraction	13
2.4 Point cloud features	14
2.4.1 Normal estimation	14
2.4.2 Keypoint selection	15
2.4.3 Local descriptor estimation	15
2.4.4 Global descriptor estimation	19
2.4.5 Creating training sets	22
2.5 Aligning point clouds	24
2.5.1 Pipelines	24
2.5.2 Iterative closest point	26

2.5.3	Initial alignment	27
2.5.4	Registration	27
2.5.5	Object detection	29
2.6	2D computer vision	30
2.6.1	Scale invariant feature transform	31
2.6.2	Speeded-up robust features	36
2.6.3	Binary robust invariant scalable keypoints	42
2.6.4	Oriented FAST and rotated BRIEF	46
2.6.5	Descriptor matching	48
2.6.6	Planar homography	51
2.7	Robot operating system	51
2.7.1	The ROS architecture	52
2.7.2	Nodes	52
2.7.3	Services	53
2.7.4	Topics	53
2.7.5	Messages	54
3	Method	55
3.1	Physical setup	55
3.1.1	Robotic cell setup	55
3.1.2	Calibrating 3D camera position	57
3.1.3	Parts used for assembly	58
3.1.4	Calibrating 2D intrinsic parameters	59
3.1.5	Calibrating eye-in-hand transform	60
3.2	Software development	64
3.2.1	Acquiring 3D point clouds	64
3.2.2	Acquiring 2D images	64
3.2.3	Control the robotic manipulator	64
3.2.4	2D object detection	65
3.2.5	3D object detection	68
3.2.6	Creating training sets	69
3.2.7	ROS communication	70
3.3	Testing setup	72
3.3.1	3D object detection accuracy	72
3.3.2	Testing global descriptors	73
3.3.3	2D object detection processing time	74
3.3.4	2D object detection matching stability	74
3.3.5	2D object detection orientation stability	75
4	Result	77
4.1	Physical setup	77
4.1.1	Robotic cell networking	77
4.1.2	Calibrating 3D camera position	77
4.1.3	Calibrating eye-in-hand transform	78
4.2	3D Computer Vision	80
4.2.1	Accuracy test of 3D object detection	80
4.2.2	Testing and selecting global descriptor	82

4.2.3	Selecting keypoint and local descriptor estimator	83
4.2.4	Creating training sets	84
4.2.5	3D object detection	85
4.3	2D computer vision	87
4.3.1	Processing time	87
4.3.2	Matching stability	89
4.3.3	Orientation stability	91
4.3.4	2D object detection	92
4.4	Software solution	94
4.4.1	ROS communication	94
4.4.2	Automated assembly sequence	95
4.4.3	Applications	96
4.5	Automated assembly solution	102
5	Discussion	105
5.1	2D computer vision	105
5.2	3D computer vision	106
5.3	Combining 2D and 3D computer vision	107
5.4	Hardware	107
6	Conclusion	109
6.1	Future work	109
	Bibliography	111
	Appendix	
A	Software Tools Created	1
B	The image_processor Application	31
C	The agilus_planner Application	57
D	Digital Appendix	63

List of Figures

2.1	Elementary rotation about the X axis.	3
2.2	Elementary rotation about the Y axis.	3
2.3	Elementary rotation about the Z axis.	3
2.4	Denavit-Hartenberg kinematic parameters (Siciliano et al., 2010).	5
2.5	Rotational directions about the joints of a KUKA KR 6 R900 sixx (GmbH, 2016).	6
2.6	Central-projection camera model. Image from Corke (2013).	7
2.7	The working principle of a Microsoft Kinect™One sensor.	9
2.8	The working principle of a structured light camera. Figure from Gaskell (2014).	9
2.9	Point cloud before passthrough filtering.	10
2.10	Point cloud after passthrough filtering.	10
2.11	Point cloud before voxel grid filtering.	11
2.12	Point cloud after voxel grid filtering.	11
2.13	Input cloud to the left is noisy, but has a sharp edge. The filtered output (to the right) preserves the edge while smoothing the point cloud. Figure from (Kadambi et al., 2014).	11
2.14	Point cloud before outlier removal filtering.	12
2.15	Point cloud after outlier removal filtering.	12
2.16	A scene with multiple objects placed on a table.	13
2.17	The table surface is detected using random sample consensus.	13
2.18	The scene after removing the segmented table.	13
2.19	A scene with multiple objects placed on a table before cluster extraction.	13
2.20	Scene after cluster extraction. Note, the table in the scene was removed using model segmentation before the cluster extraction process.	13
2.21	A scene with multiple objects placed on a table.	14
2.22	Shows the scene with the corresponding surface normals.	14
2.23	Shows how a point p_q is paired with the neighbouring p_{kn} points. Image from Rusu (2009).	16
2.24	Illustrates two paired points and the fixed reference frame \mathbf{uvw} . Image from Rusu (2009).	16
2.25	Shows the influence region for a query point using a Fast Point Feature Histogram. Image from Rusu (2009).	17
2.26	Shows the virtual sphere used to encode topological data for the Signature of Histogram of Orientations. Image from Tombari et al. (2010a).	18
2.27	Shows the pairing of points between the clusters surface points and the clusters centroid \mathbf{c} . Figure from Rusu et al. (2010).	20
2.28	Shows the central viewpoint direction \mathbf{v}_p used to calculate the relative angles between each surface normal and it. Figure from Rusu et al. (2010).	20
2.29	Shows a complete VFH histogram. The two separate component are marked. Figure from Rusu et al. (2010).	21

2.30	The different regions resulting from a region growing is illustrated with different colours. Figure from Aldoma et al. (2011)	21
2.31	Shows the point cloud of a box from the view port of the depth sensor (right) and from the side (left) to illustrate the missing part of the model.	22
2.32	Illustrates a typical setup for creating a training set based on a physical model. Image from (PCL, a).	23
2.33	Illustrates the virtual position of the model and the camera when rendering a model from different view ports. Image from (ROBOTICA, 2015).	23
2.34	Illustrates a typical pipeline using local descriptors.	24
2.35	Illustrates a typical pipeline using global descriptors.	25
2.36	Illustrates two point separate point clouds (red and green) with correspondences (drawn as a line between the points of the two clouds). The right most figure is a result of the ICP algorithm. Figure from PCL (d)	27
2.37	A typical local pipeline applied to the registration task.	28
2.38	Multiple scenes of the same room taken from different viewpoints. Figure from Rusu (2009)	28
2.39	Multiple scenes registered to form a complete model of a room. Image from Rusu (2009)	29
2.40	A typical global pipeline applied to the object detection task.	30
2.41	A scene with multiple objects placed on a table.	30
2.42	A model from a training set is registered onto a detected object, estimating its position and orientation.	30
2.43	Two Gaussian kernels with window size 41×41 . Subtracting kernel one (left), which has $\sigma = 3.2$, from kernel two (middle), which has $\sigma = 6.4$, results in the DoG between them (right). Illustrations generated using MATLAB.	32
2.44	Convolution of an image with the kernels illustrated in Figure 2.43. Generated using MATLAB.	32
2.45	Illustration of the DoG from different scales and octaves. Image from Lowe (2004)	33
2.46	Detection of maxima and minima of the DoG images. Pixel marked X is the current sample point. Image from Lowe (2004)	33
2.47	Illustration of SIFT descriptor computed from gradient magnitude and orientation. Image from Lowe (2004)	36
2.48	Illustration of an integral image. Image from Bay et al. (2008)	37
2.49	The left half shows the discretized and cropped Gaussian second order partial derivative in y - (L_{yy}) and xy -direction (L_{xy}). The right half shows the approximation for the second order Gaussian partial derivative in y - (D_{yy}) and xy -direction (D_{xy}), using box filters. Image from Bay et al. (2008)	38
2.50	Iteratively reducing the image size as in SIFT (left). The use of integral images allows the up-sampling of the filter (right). Image from Bay et al. (2008)	39
2.51	The length of the dark lobe can only be increased by an even number of pixels to guarantee the presence of the central pixel. Mask size 9×9 (left) and 15×15 (right). Image from Bay et al. (2006a)	39
2.52	Graphical representation of the filter side lengths for three successive octaves. The octaves are overlapping in order to cover all possible scales seamlessly. Image from Bay et al. (2008)	39

2.53	Haar wavelet filters to compute the responses in x (left) and y (right) direction. The weights of the dark and bright parts are illustrated.	40
2.54	The dominant orientation of the Gaussian weighted Haar wavelet response is detected within a sliding orientation window. Image from Bay et al. (2008) . . .	40
2.55	Left: An oriented quadratic grid with 4×4 square sub-regions centred around the interest point. Right: The actual fields of the descriptor, the sums d_x , d_y , $ d_x $ and $ d_y $, are computed for each sub-region relatively to the orientation of the grid. The sub-regions in this figure are 2×2 instead of 5×5 for reasons of illustration. Image from Bay et al. (2008)	41
2.56	Illustration of the nature of a SURF descriptor. Left: A homogeneous region will make all values relatively low. Middle: Frequencies in x direction will make the value of $\sum d_x $ high, but all others remain low. Right: In the case of a gradually increasing intensity in x direction, both values $\sum d_x$ and $\sum d_x $ are high. Image from Bay et al. (2008)	41
2.57	Illustrates a segment test corner detection using a 12-16 mask. The dotted arc indicates 12 pixels which are brighter than pixel p by more than a given threshold. Image from Rosten and Drummond (2006)	43
2.58	Scale-space interest point detection illustrated. The 1D parabola is fitted along the scale-axis to determine the true (interpolated) scale of the keypoint. Image from Leutenegger et al. (2011)	44
2.59	A sampling pattern with $N = 60$ sampling points including the center point, equally spaced on four concentric circles around the keypoint. For clarity, only one point in each circle is marked with a circle denoting the radius σ of the Gaussian kernel used to smooth the intensity values of the sampling points. Image from Fan et al. (2015)	44
2.60	The set of short-distance pairs, \mathcal{S} , of sampling points used for constructing the descriptor is illustrated to the left. The set of long-distance pairs, \mathcal{L} , of sampling points used for computing orientation is illustrated to the right. Each colour indicates a pair. Image from Fan et al. (2015)	45
2.61	Pyramid with 5 levels. A scale factor close to 1 will need more pyramid levels to cover a large scale range, thus increasing the computational cost. A large scale factor will on the other hand weaken the invariance to scale.	47
2.62	Keypoints detected with the ORB detector. The cyan rings denote keypoints with its respective orientation. Notice that some keypoints of differing scale overlaps each other in a concentric manner, which means they are detected from different levels of the pyramid.	47
2.63	Distance measure in a 2D plane. Image from Nixon and Aguado (2012)	50
2.64	Brute-force matching of SIFT descriptors using a distance ratio of 0.9 (left) and 0.7 (right). The cyan lines denote a match between the object and the scene. Notice that some of the lines represents false positives as shown in the left half of the figure, while there are no false positives clearly represented in the right half.	50
2.65	A simple illustration of the ROS architecture.	52
2.66	Illustrates the interaction between two nodes in a service call.	53
2.67	Illustrates the interaction between nodes when communication using topics. . .	54
3.1	Shows a simulated view of the robotic cell.	55

3.2	Picture of the robotic cell.	55
3.3	Illustrates the global origin of the robotic cell, as well as the tool and optical reference frames for the robots and cameras.	56
3.4	A simulated view of the table as seen from the 3D camera.	57
3.5	A simulated view of the table as seen from the 2D web camera.	57
3.6	Shows three typical augmented reality tags. Image from Liebhardt (2016)	57
3.7	A 3D model of <i>part A</i>	59
3.8	A 3D model of <i>part B</i>	59
3.9	Calibration procedure of camera parameters using a chessboard of size 5×7	60
3.10	Illustration of the orientations of the camera frame \mathcal{C} and world frame \mathcal{W}	61
3.11	Shows the world frame \mathcal{W} (floor), the object frame \mathcal{O} (object center) and the camera frame \mathcal{C} (camera lens).	63
3.12	Illustrates the simple computation of the objects orientation.	67
3.13	Shows the test setup used to measure the positional accuracy of the 3D object detection process.	72
3.14	The scene used to test match the different global descriptors.	73
3.15	The view from the camera during testing of SIFT.	74
4.1	Shows the network connections between the critical hardware in the robotic cell.	77
4.2	Shows the output of the <i>ar_track_alvar</i> application used for position calibration of the 3D camera.	78
4.3	Left: Image with lens distortion. Notice the curvature of the image along edges compared to the red lines. Right: The same image, but with correction for lens distortion. The curvature is minimized.	79
4.4	Illustrates the <i>eye-in-hand</i> robot control based on image coordinates. The optical center of the image is marked with three concentric circles in red, green and blue. The object center is marked with a thick red circle at the intersection of the diagonals.	80
4.5	Shows the view from the 2D object detection camera when <i>Agilus 2</i> is positioned in the initial position found using 3D object detection.	82
4.6	A collection of different point clouds that illustrates the different viewpoints generated in the process of creating a training set.	85
4.7	Illustrates the pipeline used for 3D object detection.	85
4.8	Illustrates the virtual separation of the two work areas used when performing the 3D object detection routine.	86
4.9	Shows the result from a 3D object detection process.	87
4.10	Left: Time needed to detect keypoints in a test scene. Right: Time needed to extract descriptors from the detected keypoints.	87
4.11	Left: Time needed to extract descriptor from the detected keypoints. Right: The total time needed to detect keypoints, extract descriptors from the test scene and match the descriptors with descriptors from a query image.	88
4.12	The query images used to detect the parts. From the left: Part A - top view, part A - bottom view, part B - bottom view, part B - top view.	89
4.13	Shows the desired pose of the parts before assembly. Part A (right) is to be assembled into part B (left).	91
4.14	Orientation of <i>part A</i> detected at approximately 0 degrees (left) and -90 degrees (right). Each data point is the mean of 10 measurements.	91

4.15	Orientation of <i>part B</i> detected at approximately 0 degrees (left) and -90 degrees (right). Each data point is the mean of 10 or 20 measurements.	92
4.16	Illustrates the implemented object detection.	93
4.17	Successful detection of <i>part A</i> rotated approximately -30 degrees and <i>part B</i> rotated approximately 180 degrees.	93
4.18	Shows the different applications that run in the ROS framework in order to perform the automated assembly task.	94
4.19	Illustrates the sequence used to perform an automatic assembly of two parts.	95
4.20	The main window of the <i>agilus_master_project</i> application as displayed at launch.	97
4.23	Shows the user input section that is used to run the automated assembly sequence.	99
4.24	Camera positioned above <i>part A</i> using the position acquired from the 3D system.	102
4.25	Gripper picking up <i>part A</i> at the refined position and orientation acquired from the 2D system.	102
4.26	Illustration of a successful assembly of <i>part A</i> into <i>part B</i>	103
4.27	Left: Assembly operation conducted without correction for in-plane rotation offset between query images. Right: Another assembly operation with correction for in-plane rotation offset.	103

Terms and Abbreviations

Voxel - A three dimensional pixel with a given size in x , y and z direction.

LIDAR - Light detection and ranging. A sensor used to record distance.

CAD - Computer Aided Design. A tool used to model virtual objects.

SHOT - Unique Signature of Histograms. A local descriptor.

PFH - Point Feature Histogram. A local descriptor.

FPFH - Fast Point Feature Histogram. A local descriptor.

VFH - Viewpoint Feature Histogram. A global descriptor.

CVFH - Clustered Viewpoint Feature Histogram. A global descriptor.

ICP - Iterative Closest Point. A brute-force algorithm used for refined alignment.

RANSAC - Random Sample Consensus. A method used to match data points in two different data sets.

DoG - Difference-of-Gaussian. A function used to enhance image features using Gaussian kernels.

LoG - Laplacian-of-Gaussian. A detector method using Gaussian kernels.

DoH - Determinant-of-Hessian. A detector method using the Hessian matrix.

SIFT - Scale Invariant Feature Transform. A 2D keypoint detector and descriptor extractor.

SURF - Speeded-Up Robust Features. A 2D keypoint detector and descriptor extractor.

ORB - Oriented FAST and Rotated BRIEF. A 2D keypoint detector and descriptor extractor.

BRISK - Binary Robust Invariant Scalable Keypoints. A 2D keypoint detector and descriptor extractor.

FAST - Feature from Accelerated Segment Test. A 2D keypoint detector.

BRIEF - Binary Robust Independent Elementary Features. A 2D descriptor extractor.

AGAST - Adaptive and Generic Corner Detection based on the Accelerated Segment Test. A 2D keypoint detector.

Pose - A pose defines an objects orientation and position in operational space.

Materials and software used

Hardware

KUKA KR 6 R900 sixx (KR AGILUS) - Two six axis robotic manipulators.

Schunck PSH 22-1 - Linear pneumatic gripper.

Logitech C930e - USB web camera (1080p resolution) used for 2D computer vision.

Microsoft Kinect™ One - RGB-D sensor used for 3D computer vision.

Intel NUC NUC5i5RYH - A mini computer used for the acquisition of 3D point clouds.

Software

Ubuntu 14.04 - Operating system used for the development of this project.

QtCreator - Integrated Development Environment used for C++ software development.

CLion - Integrated Development Environment used for C++ software development.

Matlab - Numerical computing environment used for matrix verification and graph plotting.

Blender - Rendering tool used to create illustrations.

Point Cloud Library 1.7 - Open source C++ library used in 3D computer vision.

OpenCV 3.1 - Open source C++ library used in 2D computer vision.

Robot Operating System - Software framework for robot system development.

Eigen3 - C++ library used for matrix calculations.

Chapter 1: Introduction

1.1 Background

The topic of this thesis is to investigate the viability of using a visual detection system consisting of both a traditional camera and a 3D camera. The long term goal of this work is to produce a robotic cell capable of automatic assembly of a wide variety of parts (within physical tooling limitations). Achieving this would increase the number of use cases where robotic assembly is viable, and provide industries that focus on low volume but highly versatile production a flexible automated solution.

In recent years, 3D camera technology has become commercially available through the Microsoft Kinect™ camera. This is an inexpensive camera, and is not intended for industrial use. Throughout this thesis, we explore the viability of using such a sensor in an industrial application. Knowing this, the camera is used in this thesis as a fast and reliable way to achieve an initial position estimation for a part. The limitations of the sensor in terms of accuracy is dealt with using an additional sensor to detect the refined position of the part.

The long term goal of creating a robust and flexible robotic cell for assembly stems from the need to increase the flexibility of traditional automation in industry. Norwegian production industries are typically based on producing a low volume of parts, where each produced product has a high price. In addition to low volume, it is typical to create a variety of different versions of the same product. This increases the difficulty of automating production in a cost effective manner even more. Because of this a more dynamic and flexible automation solution is wanted. This thesis sets out to create a basis for a vision detection system used in such an application.

It was of great interest to develop the vision detection system in a way that allows expansion in terms of parts to be assembled, and provide a simple way of defining the object that is to be detected. The goal was to successfully detect and assemble given parts at random positions and orientations. The motivation to reach the goal was driven from thorough investigation and extensive testing of different approaches of object detection.

1.2 Problem description

An *eye-in-hand* camera is useful in order to gain good image information in robotic assembly tasks. Such a camera also has the benefit of being positionable and can be aimed at a given point. This will be combined with a stationary 3D camera in this task. The 3D camera is meant to give a rough, but reliable positioning of a given part, while the *eye-in-hand* camera is meant to give an accurate positioning. The system will be tested at the *Department of Production and Quality Engineering* laboratory.

1. Describe the kinematics of a robotic cell consisting of a fixed 3D camera and an *eye-in-hand* 2D camera.

2. Create a system able to detect objects using an *eye-in-hand* 2D camera.
3. Create a system able to detect 3D models in a scene captured using a 3D camera.
4. Implement a system that utilizes both 2D and 3D computer vision to estimate an accurate position of a physical part.
5. Carry out a practical experiment where the object detection system is used for robotic assembly.

1.3 Thesis structure

This thesis is structured in the following way:

Chapter 1. Introduction - The background and motivation for this thesis is presented together with the problem description.

Chapter 2. Theory - The theory for all the technical aspects of this thesis is presented.

Chapter 3. Method - Methods used to perform tests and develop the different solutions is presented.

Chapter 4. Result - All test results and solutions are presented.

Chapter 5. Discussion - A discussion regarding the different solutions and test results obtained is made. Some personal thoughts regarding the different solutions are presented.

Chapter 6. Conclusion - The thesis work is concluded.

Chapter 2: Theory

2.1 Kinematics

Kinematics is defined as: *The branch of mechanics that deals with pure motion, without reference to the masses or forces involved in it* (dictionary.com, 2016). A more descriptive way of describing kinematics is the study of movement, position, velocity and acceleration. Using kinematics, one can create a mathematical model of links and joints, and describe the relation between rigid bodies in a model (Siciliano et al., 2010).

2.1.1 Orientation

The orientation of an object describes the objects rotation about a reference frame. Figure 2.1, 2.2 and 2.3 shows the three elementary rotations available in \mathbb{R}^3 space.

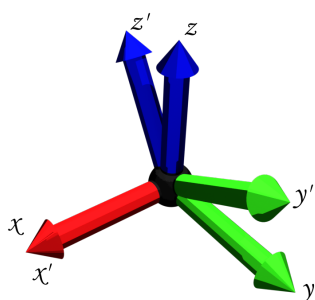


Figure 2.1: Elementary rotation about the X axis.

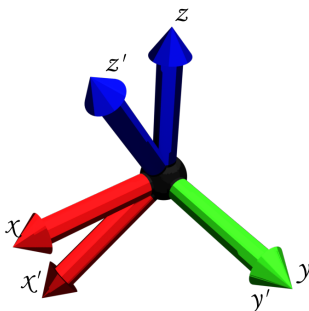


Figure 2.2: Elementary rotation about the Y axis.

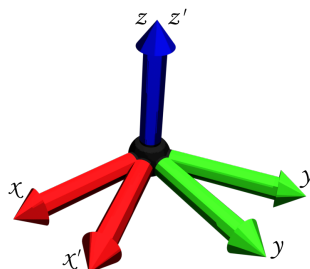


Figure 2.3: Elementary rotation about the Z axis.

There are multiple ways of describing a rotation. Among these are:

- Euler Angles
- Quaternions
- Angle Axis Description
- Rotation Matrix

A comprehensive description of the above mentioned rotation notations is available in Corke (2013). Equations 2.1, 2.2 and 2.3 shows the rotation matrices used to describe the elementary

rotations about the coordinate axes (Siciliano et al., 2010).

$$\mathbf{R}_x(\gamma) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \gamma & -\sin \gamma \\ 0 & \sin \gamma & \cos \gamma \end{bmatrix} \quad (2.1)$$

$$\mathbf{R}_y(\beta) = \begin{bmatrix} \cos \beta & 0 & \sin \beta \\ 0 & 1 & 0 \\ -\sin \beta & 0 & \cos \beta \end{bmatrix} \quad (2.2)$$

$$\mathbf{R}_z(\alpha) = \begin{bmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (2.3)$$

2.1.2 Transformation

A transformation in kinematics, is a combination of change in both orientation (rotation) and translation (position). In the context of transformation, the rotation is often described using rotation matrices, and the translation is described using a column vector. The orientation and translation can be combined to a single matrix describing both aspects. This matrix is called the homogeneous transformation matrix. Equation 2.4 shows how the translation vector \mathbf{t}_n^m and the rotation matrix \mathbf{R}_n^m is combined to the homogeneous transformation matrix \mathbf{T}_n^m :

$$\mathbf{T}_n^m = \begin{bmatrix} \mathbf{R}_n^m & \mathbf{t}_n^m \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2.4)$$

The homogeneous transformation matrix is a powerful tool because it fully describes the complete pose of an object in operational space. This matrix is commonly used in combination with the Denavit-Hartenberg convention to describe a systems forward kinematics, and to create numerical inverse kinematics solvers.

2.1.3 Denavit-Hartenberg convention

The Denavit-Hartenberg convention is a tool used in robotics to define the relation between the links that a robotic manipulator consists of. The Denavit-Hartenberg convention defines link i in connection with link $i - 1$ in terms of rotations about and translations along the x - and z -axes of the joint frame. The resulting table defines the robotic manipulators forward kinematics. The following is a simplified recipe for constructing a Denavit-Hartenberg table for a link chain as illustrated in Figure 2.4. A more comprehensive approach of defining the Denavit-Hartenberg table is described in Siciliano et al. (2010).

- Rotation about joint i is always about the z -axis of the corresponding joint frame. The rotation is denoted θ_i .
- Rotation about the x -axis is denoted α_i and refers to the next joint frame $i + 1$.
- Translation a_i applies along the joint x -axis.

- Translation d_i applies along the joint z -axis.

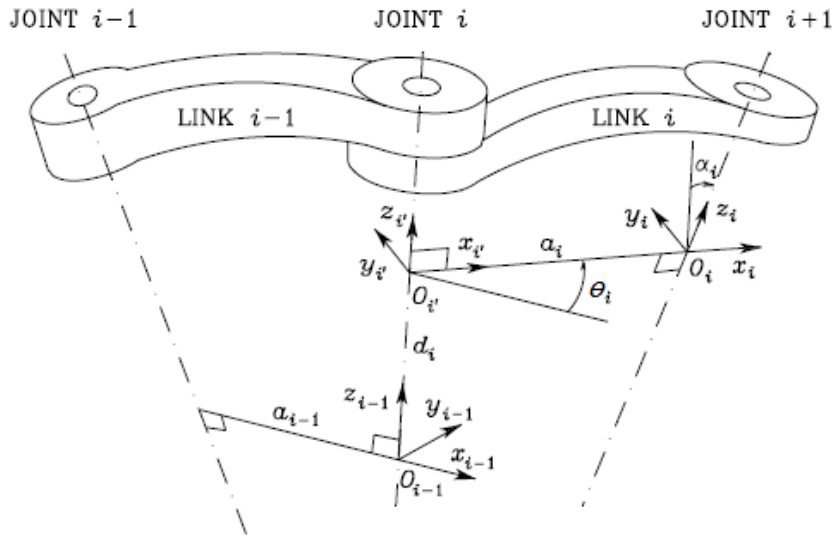


Figure 2.4: Denavit-Hartenberg kinematic parameters (Siciliano et al., 2010).

Table 2.1 shows the connection between the values shown in figure 2.4 and the actual Denavit-Hartenberg table.

Link i	a_i	α_i	d_i	θ_i
----------	-------	------------	-------	------------

Table 2.1: Link i in connection with link $i - 1$.

Table 2.2 shows an example of a complete Denavit-Hartenberg table. This table shows the Denavit-Hartenberg parameters for a KUKA KR 6 R900 sixx (GmbH, 2016) robotic manipulator. This manipulator is illustrated in figure 2.5.

Link	$a_i[m]$	$\alpha_i[rad]$	$d_i[m]$	$\theta_i[rad]$	Note
0	0	π	0	0	Rotate $z - axis$ downwards Offset: $\theta_3 = \theta_3 - \pi/2$
1	0.025	$\pi/2$	-0.400	θ_1	
2	0.455	0	0	θ_2	
3	0.035	$\pi/2$	0	θ_3	
4	0	$-\pi/2$	-0.420	θ_4	
5	0	$\pi/2$	0	θ_5	
6	0	0	-0.080	θ_6	

Table 2.2: Denavit-Hartenberg parameters of KUKA KR 6 R900 sixx.

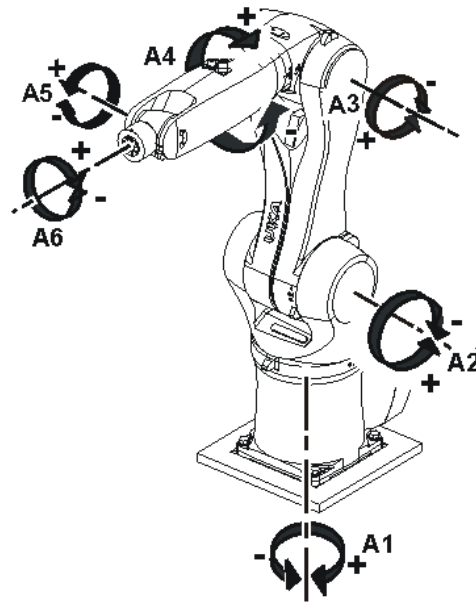


Figure 2.5: Rotational directions about the joints of a KUKA KR 6 R900 sixx (GmbH, 2016).

2.1.4 Forward kinematics

Forward kinematics is one of the two basic problems in robotics: *Given a set of joint values for a manipulator, what is the resulting position and orientation of the end effector?* Thus, forward kinematics is a way of mathematically expressing the position and orientation of the end effector frame (manipulator pose) as a function of the joint values of the manipulator arm. For a typical open chain manipulator consisting of n links, the forward kinematics can be expressed as a resulting homogeneous transformation matrix from the following equation:

$$\mathbf{T}_n^0 = \prod_{i=1}^n \mathbf{T}_i^{i-1}(\theta_i) \quad (2.5)$$

Equation 2.5 establishes the functional relationship between the joint variables of the manipulator and the end effector position and orientation (Siciliano et al., 2010). This means that the forward kinematics problem can be calculated as long as the joint values of the manipulator are known.

2.1.5 Inverse kinematics

Inverse kinematic is the second basic problem in robotics: *Given a target position and orientation of the manipulator end effector, what are the joint values?* This problem is often seen as the opposite of the forward kinematics problem, and is usually harder to solve. What makes this problem harder to solve than forward kinematics is the fact that forward kinematics gives a singular solution for a given set of joint values. This is not the case in inverse kinematics, where a given pose (position and orientation) of the end effector might have unlimited different valid solutions.

There are several ways of solving the inverse kinematics problem. One might use a pure linear algebraic approach, geometric algebra or numerical solvers. Most numerical solvers are based on the differential kinematics of a system, which defines a relationship between the joint velocities and the corresponding end-effector linear and angular velocities.

2.1.6 2D computer vision

In order to convert pixel coordinates from the 2D image matching process to 3D coordinate that is useful in robotic applications, a mathematical model of the camera setup is used. One typical camera model is called the *central-projection model* and is commonly used in computer vision (Corke, 2013). This model of a camera places the image plane in front of the camera at a depth $z = f$, which results in a non-inverted image. As seen in Figure 2.6, the wanted position in space $P = (X, Y, Z)$ is projected on the image plane at point $\mathbf{p} = (x, y)$ by:

$$x = f \frac{X}{Z}, y = f \frac{Y}{Z} \quad (2.6)$$

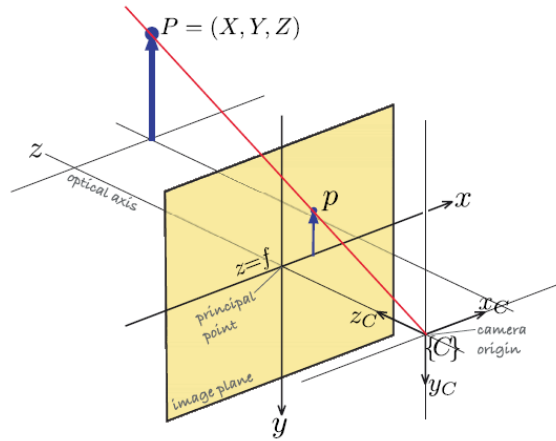


Figure 2.6: Central-projection camera model. Image from Corke (2013).

This is a perspective projection, and it gives us a simple connection between the pixel-coordinates of the object in the image $\mathbf{p} = (u, v)$, the image coordinates $\mathbf{p} = (x, y)$ and the actual position in space $P = (X, Y, Z)$. The camera parameter matrix \mathbf{K} is expressed:

$$\mathbf{K} = \begin{pmatrix} \frac{f}{\rho_w} & 0 & u_0 \\ 0 & \frac{f}{\rho_h} & v_0 \\ 0 & 0 & 1 \end{pmatrix} \quad (2.7)$$

where f is the focal length, $\rho_w = \rho_h$ is the pixel size and u_0 and v_0 is the optical center in pixels. The conversion from pixel coordinate $\mathbf{p} = (u, v)$ to point in space $P = (X, Y, Z)$ is done using the following:

$$\mathbf{p} = \begin{pmatrix} u \\ v \end{pmatrix}, \tilde{\mathbf{p}} = \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \quad (2.8)$$

The normalized image coordinate vector $\tilde{\mathbf{s}}$ is defined as:

$$\tilde{\mathbf{s}} = Z \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \quad (2.9)$$

The relation between the pixel coordinate $\tilde{\mathbf{p}}$ and the normalized image coordinate $\tilde{\mathbf{s}}$ is:

$$\tilde{\mathbf{p}} = \mathbf{K} \tilde{\mathbf{s}} \quad (2.10)$$

$$\tilde{\mathbf{s}} = \mathbf{K}^{-1} \tilde{\mathbf{p}} \quad (2.11)$$

2.2 3D computer vision

3D computer vision is the process in which a 3D image (often referred to as a point cloud) containing pixels with positional values (x, y and z) is used as a tool, allowing a computer to interpret reality. The main difference between 3D and 2D computer vision is that a depth sensor is used for 3D in stead of a regular camera. The 3D image produced by a depth sensor is dimensionally correct, and represent each point in the cloud with a position in relation to the sensors optical reference frame. This technology was, for a long time, not available on the consumer market. In recent years, this technology has been made available to consumers by Microsoft and their depth sensor that is used as a video game input device.

Depth sensors are used in a wide variety of different applications ranging from video game input to environmental mapping. Typical industrial applications for depth sensors and 3D computer vision is object detection and quality control.

There are multiple different types of depth sensors, but the majority operates based on either the principle of *time of flight* or *structured light*. Microsoft's first Kinect™ sensor operated on the principle of *structured light*, but their newest version (Microsoft Kinect™ One) operates on the *time of flight* principle. These operating principles are explained in section 2.2.1 and 2.2.2.

2.2.1 Time of flight

Time of flight cameras record the depth of a scene using the time of flight principle. In the depth measuring operation, the depth of a pixel in the scene corresponds with the flight time of light. This technology is used in many other applications like sonar, spectrometry and spectroscopy. Advances in hardware technology now allows this technology to be used for close range applications where the detectable delay between electrical signals is in the time order of 100 picoseconds (Kadambi et al., 2014).

Some time of flight depth sensors operate by measuring the depth of one pixel at the time, scanning the whole scene. This is the typical operating principle of a *light detecting and ranging* (LIDAR) sensor. 3D cameras use a different operating principle in order to achieve a higher data acquisition rate and, as a result, higher frames rate of the camera (Kadambi et al., 2014). Figure 2.7 illustrates the working principle of Microsoft's second generation Kinect™ sensor.

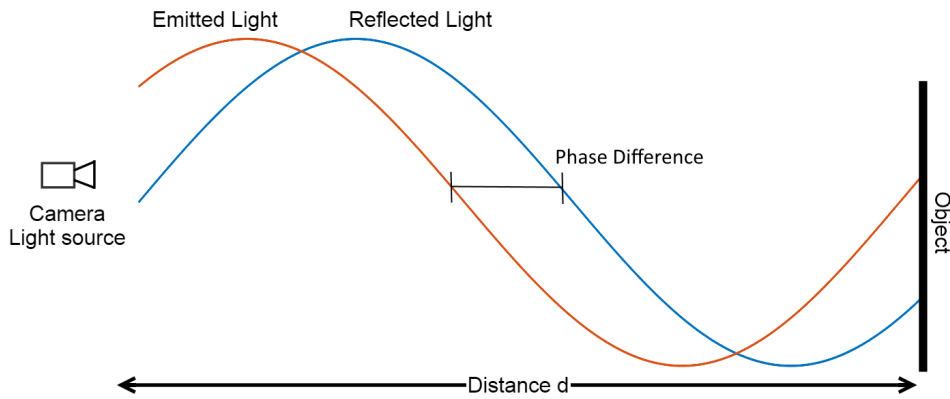


Figure 2.7: The working principle of a Microsoft Kinect™ One sensor.

The camera operate by modulating a light source periodically on the form:

$$e(t) = 1 + s_0 \cos \omega t$$

The light reflected by an object assumes the form:

$$r(t) = \rho(1 + s_0 \cos(\omega(t - \frac{2d}{c})))$$

where the fraction $\frac{2d}{c}$ contains the depth information for the pixel. The phase between the emitted and reflected light corresponds to the depth, and is calculating using cross-correlation (Kadambi et al., 2014).

2.2.2 Structured light

Structured light cameras work by projecting a known pattern onto a subject of interest. The distortion in the projected pattern is then used to calculate the depth information for each pixel in the scene. These systems operate using regular high resolution cameras, and thus have the ability for a high spatial resolution. Figure 2.8 illustrates the working principle of a structured light system.

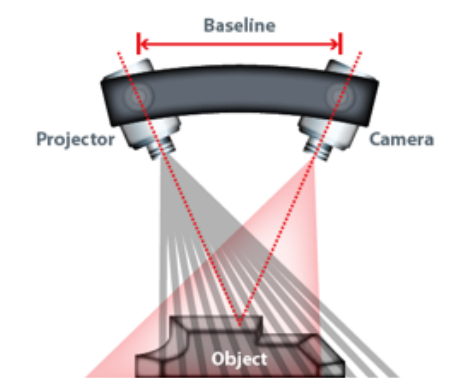


Figure 2.8: The working principle of a structured light camera. Figure from Gaskell (2014).

2.3 Processing point clouds

2.3.1 Passthrough filtering

A passthrough filter is used to section out a part of the point cloud. Usually this is done to remove unwanted parts of the scene. A typical implementation of a passthrough filter lets you set minimum and maximum limits in each axis. All the points that is contained within these limits are kept, the rest are discarded. The limits are set in relation to a reference coordinate frame. This frame is, in most cases, the optical frame of the depth sensor.

Passthrough filtering is often the first step when processing a point cloud. This is because it often drastically reduces the amount of data necessary to process. Figure 2.9 and 2.10 shows the result of a passthrough filter operation.

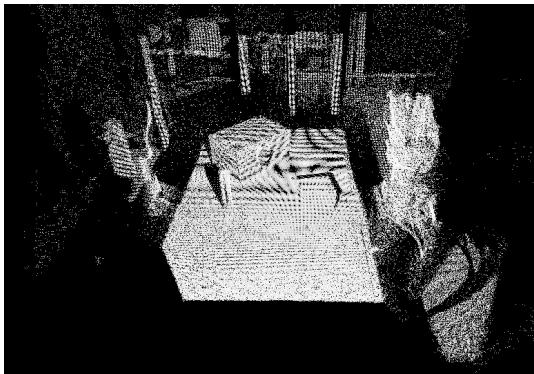


Figure 2.9: Point cloud before passthrough filtering.

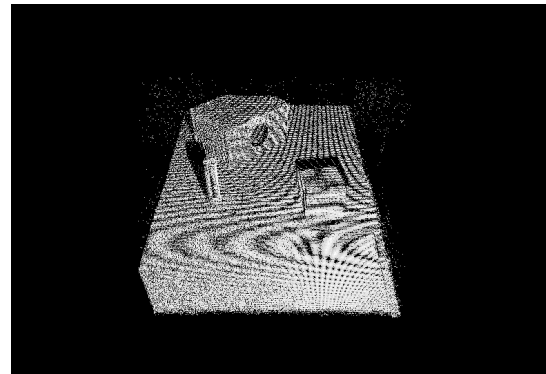


Figure 2.10: Point cloud after passthrough filtering.

2.3.2 Voxel grid filtering

Voxel grid filtering is one way of down-sampling a point cloud. Down-sampling is done to reduce the number of data points and, as a result, the processing time is reduced. In addition, a down-sampling process often results in a uniform point cloud, where the density of points is constant through out the cloud. A uniform cloud is important when estimating descriptors, since the descriptor contains data about a group of points.

This filter works by virtually sectioning the cloud into voxels with definable size. The average position for all points contained within a voxel is calculated and a new point is created at the resulting position. All the original points are discarded and the only point left within a voxel is the newly created average point.

The number of points left after a voxel grid operation is defined by the voxel size defined before the operation. A voxel is defined by its width, height and depth. Figure 2.11 and 2.12 shows the result of a voxel grid filter operation.

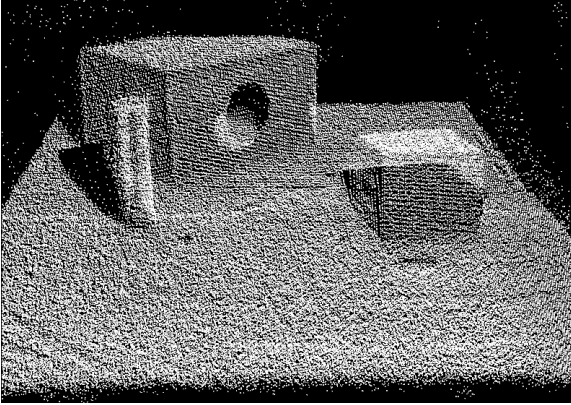


Figure 2.11: Point cloud before voxel grid filtering.

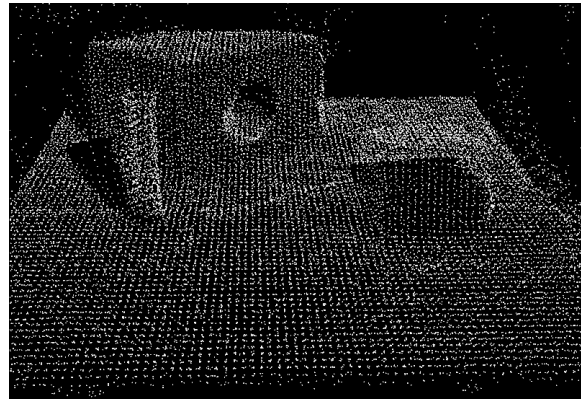


Figure 2.12: Point cloud after voxel grid filtering.

2.3.3 Bilateral filtering

A bilateral filter is a filter commonly used in computer graphics in order to smooth noise while still preserving edge discontinuities. A version of this filter transposed to work on 3D point clouds is often used in 3D computer vision for just the same purpose. In addition, the bilateral filter used in 3D computer vision also serves the purpose of filling holes generated during the depth measuring process of the sensor (Kadambi et al., 2014). Holes are quite common in the output of a depth sensor and is caused by missing depth data from the sensor. Equation 2.12 shows the formula used in bilateral filtering of 3D point clouds.

$$D_f^p = \frac{H(C_{map}, \Omega^p)}{k^p} \sum_{q \in \Omega^p} \hat{D}^q f(p, q) h(\|I^p - I^q\|) \quad (2.12)$$

Figure 2.13 shows the process of bilateral filtering of a point cloud. The process goes from left to right with a noisy input point cloud filtered to a less noisy output while still preserving sharp edges.

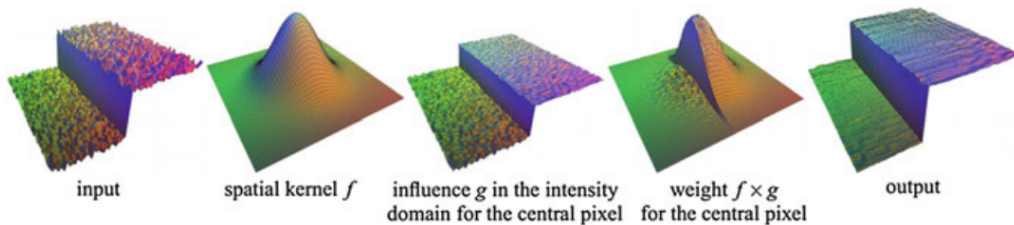


Figure 2.13: Input cloud to the left is noisy, but has a sharp edge. The filtered output (to the right) preserves the edge while smoothing the point cloud. Figure from (Kadambi et al., 2014).

2.3.4 Outlier removal filtering

Outlier removal filters is used to remove, as the name suggest, outliers. Outliers are points that are not considered as part of an object. Figure 2.14 shows a scene with some outliers (floating points above the table with objects). These outliers have been removed in Figure 2.15 using an outlier removal filter.

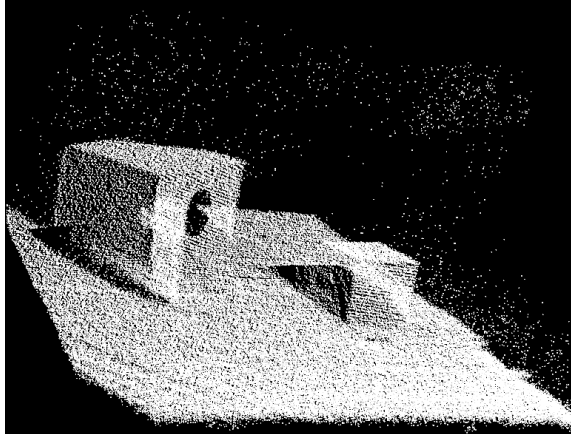


Figure 2.14: Point cloud before outlier removal filtering.

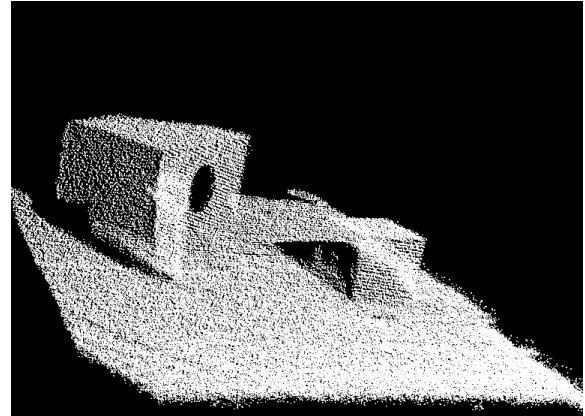


Figure 2.15: Point cloud after outlier removal filtering.

A typical implementation of a outlier removal filter calculates the average distance to each points k nearest neighbours. The resulting data set is assumed to be Gaussian distributed with a mean and standard deviation. All points with a mean distance to its k nearest neighbours greater than the mean and standard deviation obtained on the complete data set are considered to be outliers and removed from the point cloud (PCL, e).

2.3.5 Model segmentation

Model segmentation is the process of separating a part of a point cloud scene. This is done using random sample consensus (RANSAC) in conjunction with a mathematical model of the object to segment. The random sample consensus approach uses parameters for the mathematical model set by the user to fit the largest possible number of points and marks them as inliers (inliers are points that are considered to be part of the model). These points can then be extracted to an individual point cloud, or removed.

A typical application for model segmentation is using a model of a plane in order to remove large unwanted features from a scene like a roof, floor, walls or a table surface. Doing this makes it possible to use cluster extraction in order to separate all parts located on a large surface.

Figure 2.16 shows a typical scene consisting of three objects placed on a table. The table is detected in Figure 2.17 using planar model segmentation (the plane inliers are marked with the colour red). Figure 2.18 shows the scene when the table surface have been segmented and removed.

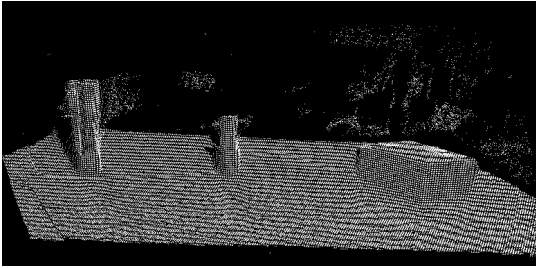


Figure 2.16: A scene with multiple objects placed on a table.

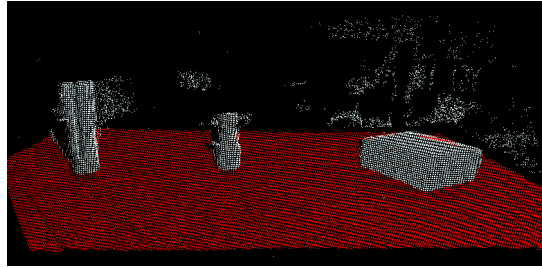


Figure 2.17: The table surface is detected using random sample consensus.

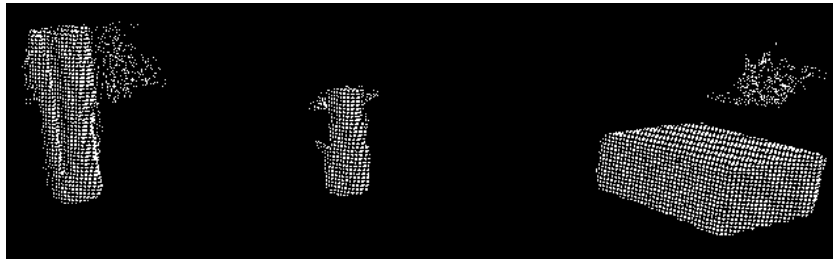


Figure 2.18: The scene after removing the segmented table.

2.3.6 Cluster extraction

Cluster extraction when working with 3D point clouds is the process of separating different parts of a scene into individual point clouds. A common application for this is object detection, where multiple objects might be present in a scene. By separating the different objects the task of recognizing each object is simplified. Figure 2.19 shows a scene containing three individual objects. Note that the table in this scene have been removed using model segmentation in order to better illustrate cluster extraction. Figure 2.20 shows the result of the cluster extraction where each object have been given a distinct colour for illustrative purposes.

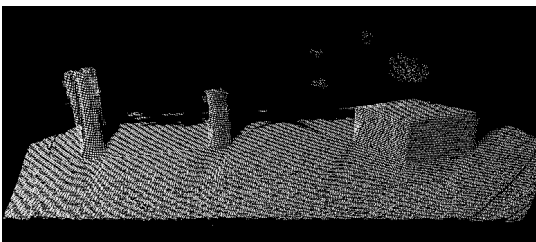


Figure 2.19: A scene with multiple objects placed on a table before cluster extraction.

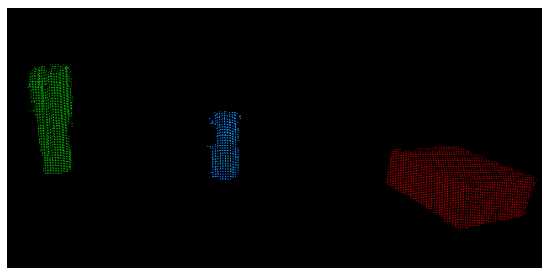


Figure 2.20: Scene after cluster extraction. Note, the table in the scene was removed using model segmentation before the cluster extraction process.

The cluster extraction process works by first defining a random point as part of a cluster. A virtual sphere is used to search for neighbouring points within a given distance. All points

within this sphere are also considered to be part of the same cluster. This is repeated for all newly added points until there are no new points found. Next, a new point, not contained in the first cluster, is defined as a part of a new cluster. The process is repeated for all points in the data set. The final step is to reduce the number of clusters caused by noise. This is done by limiting the minimum and maximum amount of points allowed in a cluster. The resulting clusters are separated into individual point clouds (PCL, c).

Due to the ability to limit the minimum amount of points in a cluster. Cluster extraction also serves the purpose of removing outliers.

2.4 Point cloud features

2.4.1 Normal estimation

The problem of estimating surface normals for a point cloud can be approximated by the problem of estimating the normal of a plane tangent to the surface. This problem is reduced to an analysis of a covariance matrix created from a points nearest neighbours where the covariance matrix's eigenvectors and eigenvalues are the point of interest. This is called a *Principal Component Analysis* (PCL, b).

The covariance matrix \mathcal{C} is constructed for each point \mathbf{p}_i as shown in equation 2.13.

$$\mathcal{C} = \frac{1}{k} \sum_{i=1}^k \cdot (\mathbf{p}_i - \bar{\mathbf{p}}) \cdot (\mathbf{p}_i - \bar{\mathbf{p}})^T, \quad \mathcal{C} \cdot \vec{\mathbf{v}}_j = \lambda_j \cdot \vec{\mathbf{v}}_j, \quad j \in \{0, 1, 2\} \quad (2.13)$$

The sign value of the normal vector resulting from the principal component analysis is ambiguous, thus further calculations are needed. This is done by orienting all the surface normals towards the viewpoint \mathbf{v}_p of the point cloud. The surface normal $\vec{\mathbf{n}}_i$ is oriented towards the viewpoint when the following equation is satisfied:

$$\vec{\mathbf{n}}_i \cdot (\mathbf{v}_p - \mathbf{p}_i) > 0 \quad (2.14)$$

Figure 2.21 and 2.22 shows the result of a normal estimation on a scene.

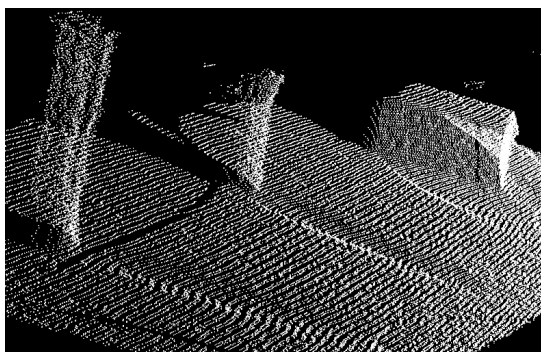


Figure 2.21: A scene with multiple objects placed on a table.

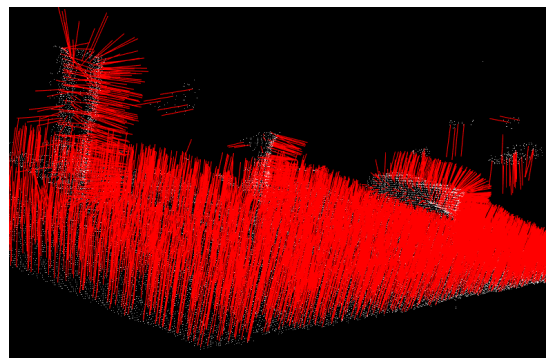


Figure 2.22: Shows the scene with the corresponding surface normals.

2.4.2 Keypoint selection

Keypoints in 3D computer vision serves the same purpose as in traditional computer vision. The total number of points is reduced to only the ones that contain the most information. The selection of good keypoints is critical to achieve well performing object detection and registration when working with point clouds. This is because most features calculated for the point cloud are based on keypoints, and not the full data set. Features are descriptive properties that are used to identify a particular region or area of a point cloud.

The most commonly used keypoint detectors are:

- Harris3D (Harris and Stephens, 1988)
- SIFT3D (Rusu and Cousins, 2011)
- SUSAN (Smith and Brady, 1997)
- ISS3D (Zhong, 2009)

A study conducted by Filipe and Alexandre (2014) set out to compare these 3D keypoints detectors and concluded that SIFT3D and ISS3D are the most stable keypoint selectors based on repeatability.

The Scale Invariant Feature Transform (SIFT) keypoint detector was proposed by Lowe (2004). This keypoint detector is described in detail in section 2.6.1. The modified algorithm used for SIFT on 3D data sets was presented by Rusu and Cousins (2011). The most notable difference between the two algorithms are that SIFT3D uses a 3D version of the Hessian to select interest points, and that the intensity of a pixel is changed to the principal curvature of a given point.

2.4.3 Local descriptor estimation

A local descriptor is an object that describes the local geometrical area for one single point in a point cloud. This type of descriptor is typically calculated for each point in a 3D point cloud, or for a selected number of points like keypoints. The goal of a local descriptor is to create a description of a point and its surroundings that is not limited to the data contained in the point cloud (which is only a points cartesian coordinate x , y and z). Local descriptors were specifically created for tasks like registration (see section 2.5.4) and object detection (see section 2.5.5).

There are multiple ways of creating a description based on a points geometrical surroundings. These methods use different mathematical principle to encode a description of a point.

Notable local descriptors are

- Point Feature Histogram (Rusu, 2009)
- Fast Point Feature Histogram (Rusu et al., 2009)
- Signature of Histogram of Orientation (Tombari et al., 2010a)
- 3-D Shape Context (Frome et al., 2004)
- Spin Images (Johnson, 1997)

- Unique Shape Context (Tombari et al., 2010b)

The operating principle of the three most commonly used local descriptors is explained below.

Point feature histogram

The Point Feature Histogram (Rusu, 2009) captures the surrounding geometrical information by analyzing the difference between the normals of the points in a surrounding area of a selected point. Firstly, points within the same vicinity are paired. Then a fixed coordinate system is calculated based on the normals for each point pair. The fixed coordinate frame is used as a reference to encode the difference between the normals in the three angular values α , ϕ and θ .

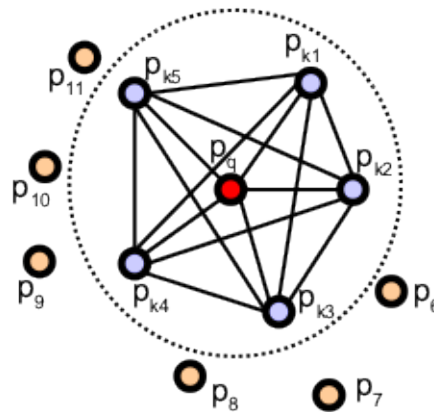


Figure 2.23: Shows how a point p_q is paired with the neighbouring p_{kn} points. Image from Rusu (2009).

Figure 2.23 shows how a point p_q is paired to the neighbouring p_{kn} points. Equation 2.15 shows how the reference frame \mathbf{uvw} used to encode the angular differences is calculated. This is also illustrated in Figure 2.24

$$u = n_s, \quad v = u \times \frac{p_t - p_s}{\|p_t - p_s\|_2}, \quad w = u \times v \quad (2.15)$$

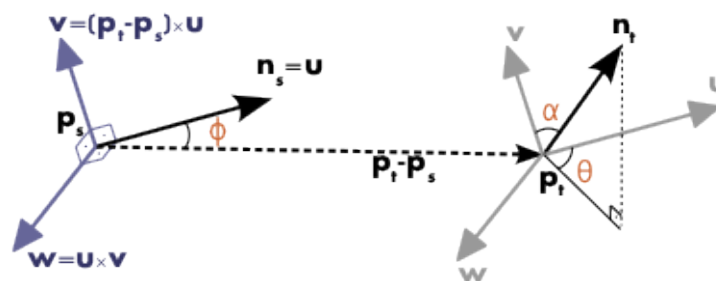


Figure 2.24: Illustrates two paired points and the fixed reference frame \mathbf{uvw} . Image from Rusu (2009).

The angular values α, ϕ and θ are calculated using the following equations.

$$\alpha = v \cdot n_t, \quad \phi = u \cdot \frac{p_t - p_s}{\|p_t - p_s\|}, \quad \theta = \arctan(w \cdot n_t, u \cdot n_t) \quad (2.16)$$

For a given point cloud with n points and k number of neighbours used when pairing, the computational complexity for this descriptor is nk^2 .

Fast point feature histogram

The Fast Point Feature Histogram (Rusu et al., 2009) is a simplification of the Point Feature Histogram. It is simplified in a way that reduces the computational complexity of a point cloud with n points and k neighbours considered from nk^2 to nk . Because of this, the FPFH descriptor requires less computational time, allowing it to be used for real-time applications.

The mathematical concept used for the FPFH descriptor is the same as for the PFH descriptor. There is however a big difference in how the final result is prepared. The simplification is done after point pairs are created and the values α, ϕ and θ are calculated. Next, all k neighbouring points are re-determined and the initially calculated α, ϕ and θ are weighted by the distance ω_k between the query point p_q and the neighbouring points p_k . The initially calculated α, ϕ and θ are called the Simplified Point Feature Histogram (SPFH) resulting in the following formula:

$$FPFH(p_q) = SPFH(p_q) + \frac{1}{k} \sum_{i=1}^k \frac{1}{\omega_k} \cdot SPFH(p_k) \quad (2.17)$$

For a query point p_q and its neighbouring point p_k with k neighbouring points and a distance ω_k between the points.

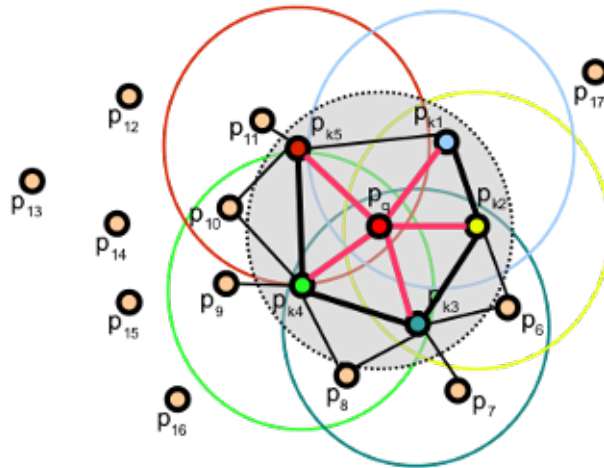


Figure 2.25: Shows the influence region for a query point using a Fast Point Feature Histogram. Image from Rusu (2009).

By comparing Figure 2.23 to Figure 2.25 a clear difference is apparent. The FPFH descriptor has a much larger influence region, but each point has, in general, less connections than a query point using the Point Feature Histogram descriptor.

Unique signature of histograms

The *Unique Signature of Histograms* (SHOT) descriptor (Tombari et al., 2010a) differs from both PFH and FPFH. Both PFH and FPFH utilizes the surface normal to encode data describing the geometrical area surrounding a point. The SHOT descriptor uses a different approach. Here a virtual sphere is created around each query point. For each query point, the surrounding points location within the virtual sphere is used to encode the topological traits in the area. The virtual sphere used is shown in Figure 2.26.

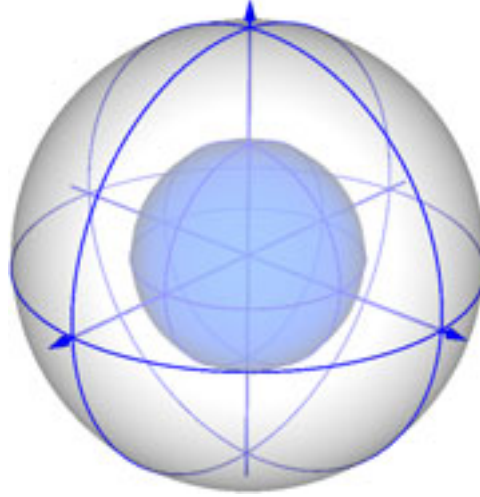


Figure 2.26: Shows the virtual sphere used to encode topological data for the Signature of Histogram of Orientations. Image from Tombari et al. (2010a).

The SHOT method focuses on developing a reliable way of creating a repeatable reference frame. This frame is used to encode data regarding a query points neighbouring points (those contained in the virtual sphere). This reference frame is created using the same mathematical principle as the one used for *Normal Estimation* (see section 2.4.1), using eigenvector decomposition of a covariance matrix M created by the k nearest neighbouring points p_i surrounding the query point p . This is shown in equation 2.18

$$M = \frac{1}{k} \sum_{i=0}^k (p_i - \hat{p})(p_i - \hat{p})^T, \quad \hat{p} = \frac{1}{k} \sum_{i=0}^k p_i \quad (2.18)$$

This equation is modified slightly in order to achieve a weighting of points based on distance. This is done to improve robustness and repeatability in presence of clutter. This change is shown in equation 2.19

$$M = \frac{1}{\sum_{i:d_i \leq R} (R - d_i)} \sum_{i:d_i \leq R} (R - d_i)(p_i - p)(p_i - p)^T \quad (2.19)$$

Where d_i is the distance between two points $\|p_i - p\|_2$.

Next, the sign of the coordinate axes are calculated in a way to achieve high repeatability. The following shows how this is done for the x axis.

$$S_x^+ \doteq \{i : d_i \leq R \wedge (p_i - p) \cdot x^+ \geq 0\} \quad (2.20)$$

$$S_x^- \doteq \{i : d_i \leq R \wedge (p_i - p) \cdot x^- > 0\} \quad (2.21)$$

$$x = \begin{cases} x^+, & |S_x^+| \geq |S_x^-| \\ x^-, & \text{otherwise} \end{cases} \quad (2.22)$$

The z axis is calculated using the same equations as for the x axis, and the final axis (y) is obtained by: $z \times x$

2.4.4 Global descriptor estimation

A global descriptor is, in many ways, similar to a local descriptor. The main difference is that while a local descriptor describes a single point and the local area around it, a global descriptor describes a cluster of points. Because of this, the global descriptor is highly suitable for applications like object detection and object classification, where a description of a full object is useful.

Similar to the local descriptor, the purpose of the global descriptor is to describe an area with more detail than what is available in the point cloud (x , y and z coordinates). Multiple different implementations of global descriptors are available, and they use different operating principles.

Notable global descriptors are

- Viewpoint Feature Histogram (Rusu et al., 2010)
- Clustered Viewpoint Feature Histogram (Aldoma et al., 2011)
- Oriented, Unique and Repeatable Clustered Viewpoint Feature Histogram (Aldoma et al., 2012b)
- Ensemble of Shape Functions (Wohlkinger and Vincze, 2011)
- Global Radius-based Surface Descriptor (Marton et al., 2011)

The operating principle of the two most commonly used global descriptors is explained in the following two sections.

Viewpoint feature histogram

The Viewpoint Feature Histogram (Rusu et al., 2010) describes a cluster of points using a combination of an extended Fast Point Feature Histogram component and a viewpoint direction component. The extended Fast Point Feature Histogram is a modified version of the Fast Point Feature Histogram local descriptor that allows the descriptor to be estimated for an

entire object cluster. This is done by creating point pairs between the surface points and the centroid of the object. The encoded variables for this global version is the same as in the local version (α , ϕ and θ). The same equations are used:

$$\alpha = v \cdot n_t, \quad \phi = u \cdot \frac{p_t - p_s}{\|p_t - p_s\|}, \quad \theta = \arctan(w \cdot n_t, u \cdot n_t) \quad (2.23)$$

Figure 2.27 illustrates the point pairs between the surface points and the objects centroid.

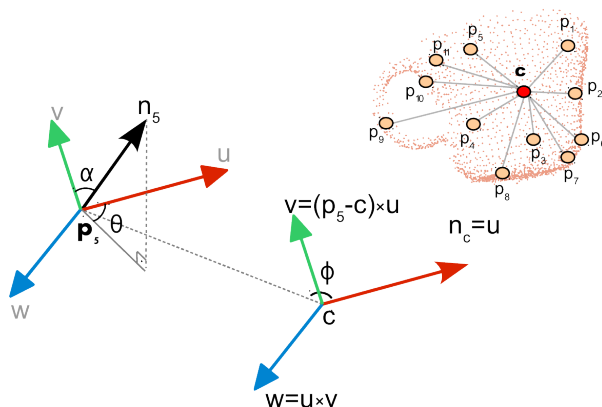


Figure 2.27: Shows the pairing of points between the clusters surface points and the clusters centroid c . Figure from Rusu et al. (2010).

The second component in the Viewpoint Feature Histogram is the viewpoint direction component. This component is calculated as the relative angles between each surface normal for the cluster and the central viewpoint direction. This is illustrated in Figure 2.28.

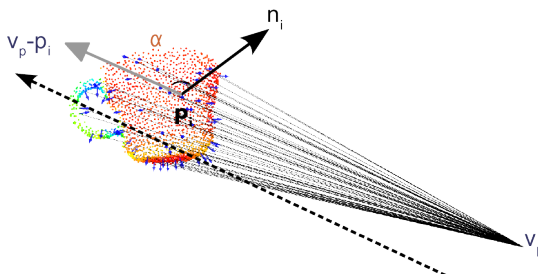


Figure 2.28: Shows the central viewpoint direction v_p used to calculate the relative angles between each surface normal and it. Figure from Rusu et al. (2010).

These two components are combined to a single histogram containing both the viewpoint data (viewpoint direction component) and the surface normal data (extended fast point feature histogram component). A complete Viewpoint Feature Histogram is shown in Figure 2.29.

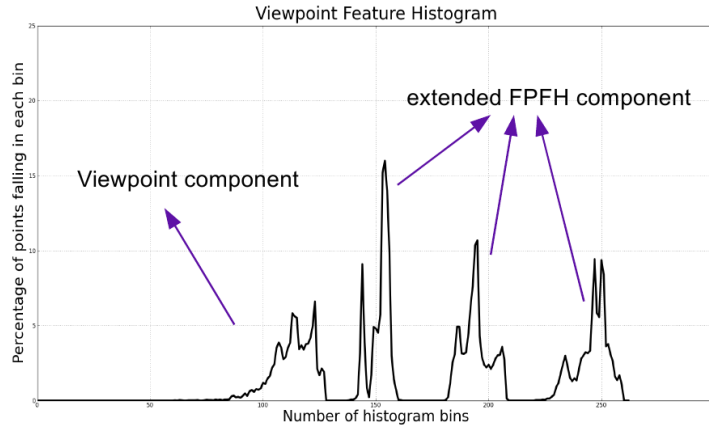


Figure 2.29: Shows a complete VFH histogram. The two separate component are marked. Figure from Rusu et al. (2010).

Clustered viewpoint feature histogram

The Clustered Viewpoint Feature Histogram (Aldoma et al., 2011) builds on the Viewpoint Feature Histogram in order to capture a higher level of detail. This is done by dividing the object cluster into multiple stable and smooth regions. The separation is done using region growing segmentation. For each region, a separate Viewpoint Feature Histogram is calculated. Figure 2.30 shows the result of the region growing on a typical household object.

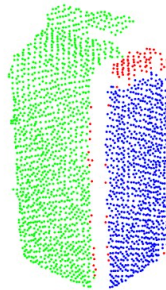


Figure 2.30: The different regions resulting from a region growing is illustrated with different colours. Figure from Aldoma et al. (2011).

The benefits of using *Clustered Viewpoint Feature Histogram* (CVFH) instead of *Viewpoint Feature Histogram* (VFH) is that the CVFH descriptor is more robust to occlusion than the more basic VFH descriptor. This is because the CVFH will allow detection of an object as long as one of the regions of the object is visible to the depth sensor. Note that in order for this descriptor to function properly, it is essential that the matching model and the object cluster are quite similar. This is important in order to assure that the region growing process used for this descriptor produces the same regions for both the object cluster and the model point cloud. If the process of region growing results in different regions for the two point clouds, the descriptors can not be compared.

2.4.5 Creating training sets

A typical 3D object detection setup usually consists of a depth sensor mounted on a known location. The output of the depth sensor produces a scene from the viewpoint of the camera. This means that three dimensional objects is not fully visible to the camera. Figure 2.31 illustrates this effect.

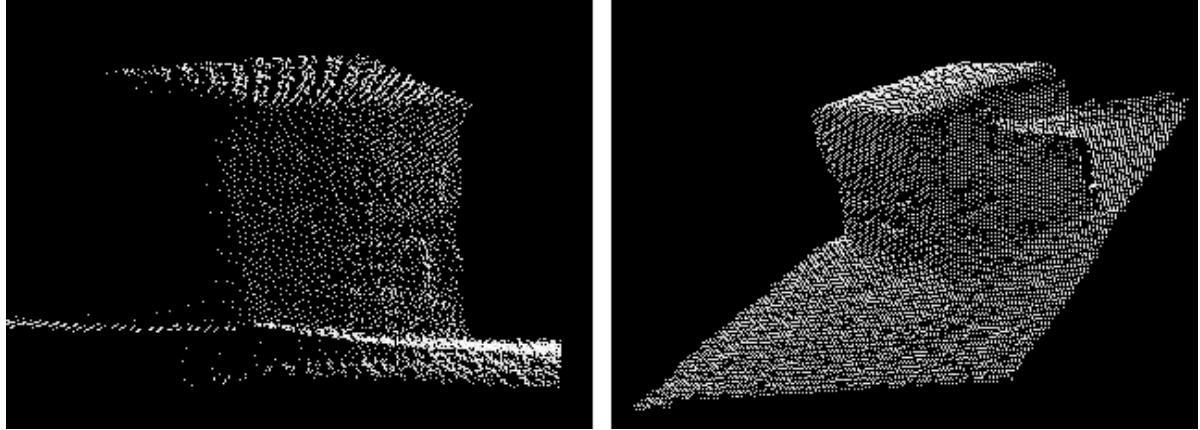


Figure 2.31: Shows the point cloud of a box from the view port of the depth sensor (right) and from the side (left) to illustrate the missing part of the model.

The alignment method used for 3D object detection uses a brute force approach (this is explained in section 2.5.2). Because of this, the model used for matching should be as close as possible to the object captured using the depth sensor. This means that only the parts of a model visible from a specific viewpoint should be included in the model. This can be achieved using two main approaches.

The first method is using a rotating pan-tilt platform and a depth sensor at a known location. The object is placed on the platform and several scenes are captured from multiple different viewpoints (the object is rotated about all three axes). Figure 2.32 shows a typical physical setup. This approach requires some post processing of the scene after capture in order to isolate the model in the scene. The isolated model is stored along with the orientation of the part for each capture.

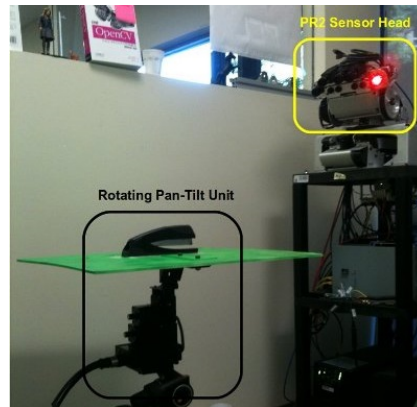


Figure 2.32: Illustrates a typical setup for creating a training set based on a physical model. Image from (PCL, a).

The second method is a virtual process that simulates the physical approach. The object that is to be found in the scene is modelled using CAD (computer aided design). This model is then rendered to a point cloud using a virtual camera. A typical approach places the model in the center of a tessellated sphere where the model is rendered to a point cloud with the virtual camera placed at the intersecting points of the faces of the tessellated sphere. Figure 2.33 illustrates the position of the virtual camera in relation to the object. The position of the camera for each rendered point cloud is stored along side with the point cloud rendering of the model.

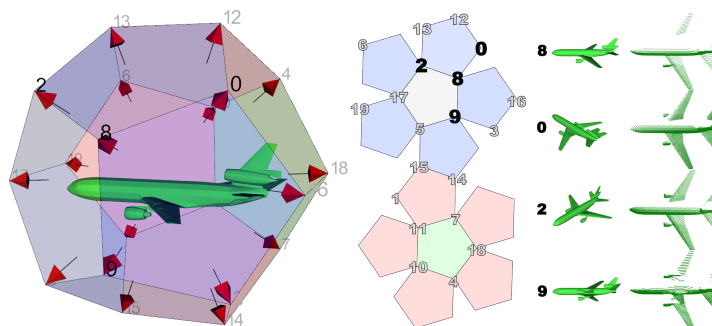


Figure 2.33: Illustrates the virtual position of the model and the camera when rendering a model from different view ports. Image from (ROBOTICA, 2015).

Both the physical and virtual method results in the same output, which is a set of point clouds of the object from different viewpoints with corresponding orientation data for each point cloud. In addition to this, it is typical to calculate a complete set of features for each point cloud (features include local keypoints, local descriptors, global descriptors and surface normals). This reduces the processing time required for the actual object detection pipeline later on. The resulting files are saved to disk for future use. A complete training set will contain multiple point clouds with corresponding:

- Object pose
- Surface normals

- Local descriptors
- Global descriptor

2.5 Aligning point clouds

2.5.1 Pipelines

A pipeline in the context of 3D computer vision is a suggested sequence of operations used to fulfill a particular goal. Typical goals for 3D computer vision is object detection, registration and object classification. Pipelines are usually divided into two different categories: global and local. The separation is done based on the type of descriptors used (local or global).

It is important to note that it is fully possible to use both local and global pipelines for object detection, whereas registration is typically done using a local pipeline. In addition, it is possible to use different parts of both types of pipeline in combination (an example of this is using global descriptors for object matching, and local descriptors for initial alignment).

The versions of both pipelines described in the sections below were originally presented by [Aldoma et al. \(2012a\)](#). Note that if the goal is 3D object detection, one additional step not included in the model is required. This is the creation of a training set (described in section 2.4.5).

Local pipeline

Figure 2.34 shows a graphical representation of a typical local pipeline.

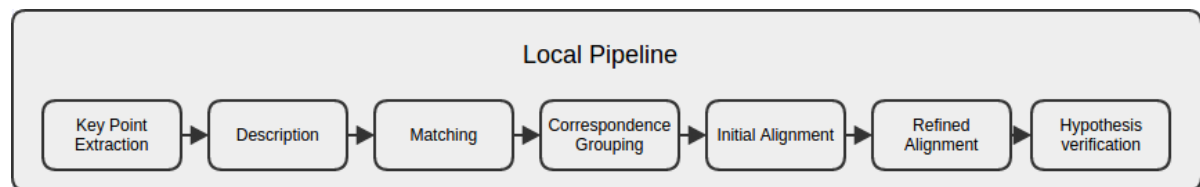


Figure 2.34: Illustrates a typical pipeline using local descriptors.

The following is a short description of the different steps in the local pipeline.

1. **Keypoint Extraction** - Keypoints are selected for both the source point cloud and the target point cloud.
2. **Description** - Local descriptors are calculated for all keypoints in both the source and target point cloud.
3. **Matching** - The descriptors for the source and target point clouds are matched, creating correspondences. Correspondences are point pairs between the source and target point cloud, effectively matching regions from the source point cloud with regions in the target cloud.
4. **Correspondence Grouping** - This step is only used for object detection. The correspondences found in the previous step are grouped based on geometric constraints. This

is done to group all correspondences that applies for one particular object. This step is essential when using the local pipeline for object detection.

5. **Initial Alignment** - A rigid transform between the point pairs contained in the correspondences is estimated. This is typically done using a Random Sample Consensus approach (RANSAC).
6. **Refined Alignment** - The alignment between the source and target point cloud is refined using a brute force approach (*Iterative Closest Point*). The output from this step is the final transformation between the source and the target point cloud.
7. **Hypothesis Verification** - This step is not always necessary and mainly used when the goal is to detect objects in cluttered or heavily occluded scenes. This is an algorithm that applies geometrical constraints to the positive matches between the source and target cloud, minimizing the number of false positives.

Global pipeline

Figure 2.35 shows a graphical representation of a typical global pipeline.

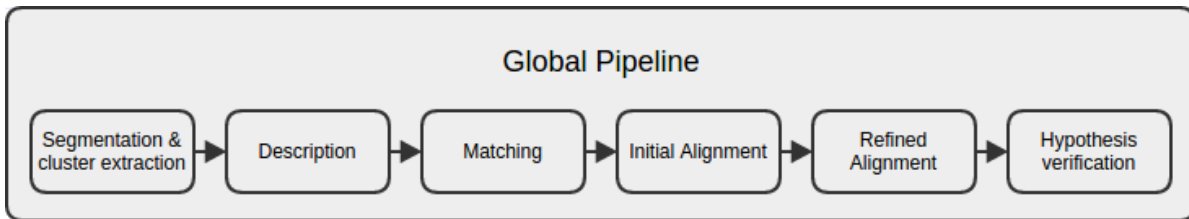


Figure 2.35: Illustrates a typical pipeline using global descriptors.

The following is a short description of the different steps in the global pipeline.

1. **Segmentation & Cluster Extraction** - The scene is segmented and clusters are extracted in order to isolate one or more object clusters.
2. **Description** - A global descriptor is calculated for both the object cluster (source point cloud) and for all models in the training set (target point cloud).
3. **Matching** - The global descriptor from the source point cloud is matched to the descriptors of the training set. This step selects the model from the training set that matches the object point cloud the best.
4. **Initial Alignment** - This step is the same for both the local and global pipeline. A rigid transform between the point pairs contained in the correspondences is estimated. This is typically done using a Random Sample Consensus approach (RANSAC).
5. **Refined Alignment** - This step is the same for both the local and global pipeline. The alignment between the source and target point cloud is refined using a brute force approach (*Iterative Closest Point*). The output from this step is the final transformation between the source and the target point cloud.

6. **Hypothesis Verification** - This step is the same for both the local and global pipeline, and is mostly used for applications where the scene is cluttered or heavily occluded. This is an algorithm that applies geometrical constraints to the positive matches between the source and target cloud, minimizing the number of false positives.

2.5.2 Iterative closest point

The Iterative Closest Point (ICP) algorithm is an iterative method for registration of two sets of points. The registration is done by minimizing the distance between corresponding points. This can be done using different mathematical approaches. A popular method, presented by [Arun et al. \(1987\)](#) uses *singular value decomposition* to minimize the error in rotation \mathbf{R} and translation \mathbf{T} between two sets of points. The problem that ICP set out to solve is described in [Arun et al. \(1987\)](#) as follows:

Given a two set of 3D points $\{p_i\}; i = 1, 2, \dots, N$ where p_i is considered as 3×1 column vector. The point set is modeled as a rigid object with a rotation \mathbf{R} , a translation \mathbf{T} and a noise vector N_i :

$$p_i' = \mathbf{R}p_i + \mathbf{T} + N_i \quad (2.24)$$

The problem of minimizing the rotation \mathbf{R} and translation \mathbf{T} between the two sets is expressed as:

$$\Sigma^2 = \sum_{i=1}^N \|p_i' - (\mathbf{R}p_i + \mathbf{T})\|^2 \quad (2.25)$$

The least square solution is when the two sets of 3D points have the same centroid. This allows for a simplification of the original problem to:

$$\Sigma^2 = \sum_{i=1}^N \|q_i' - \mathbf{R}q_i\|^2 \quad (2.26)$$

Where q_i is $p_i - p$ because the least square solution is when the two sets of points have the same centroid. This reduces the original problem to only find \mathbf{R} to minimize Σ^2 . The translation \mathbf{T} can then be found using:

$$\mathbf{T} = p' - \mathbf{R}p \quad (2.27)$$

The iterative loop of an ICP algorithm can be summed up to the following steps:

1. Select point correspondences between the two data sets.
2. Minimize the rotation \mathbf{R} and translation \mathbf{T} .
3. Iterate 1. and 2. until the error Σ^2 is within a user set threshold.

The ICP algorithm is commonly utilized as a final step when aligning two sets of point clouds. An initial, less accurate alignment is calculated first, then used as the start point for the iterative loop. This is done to reduce the number of ICP iterations.

Figure 2.36 shows two point clouds before and after Iterative Closest Point alignment.

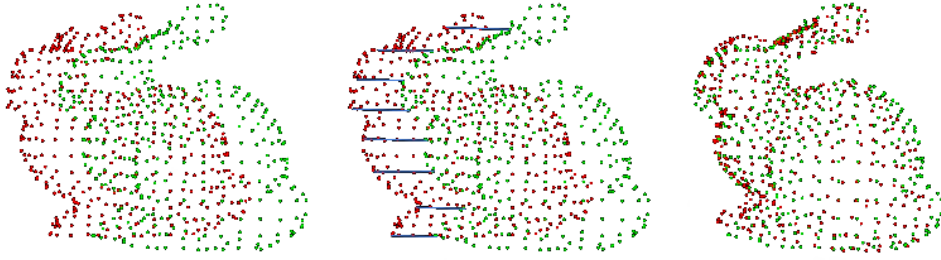


Figure 2.36: Illustrates two point separate point clouds (red and green) with correspondences (drawn as a line between the points of the two clouds). The right most figure is a result of the ICP algorithm. Figure from PCL (d).

2.5.3 Initial alignment

Initial alignment is done using local descriptor matching. This process is similar to the ICP approach. The main difference is that this step is utilized as an initial alignment, and is not meant to be highly accurate. Descriptor matching uses correspondences between the key points of a source and target point cloud to estimate a rigid transform (rotation \mathbf{R} and translation \mathbf{t}). This differs from the ICP method, since the ICP method estimates a rigid transform based on all points in the two point clouds. The number of correspondences is reduced using methods for bad correspondences rejection.

A typical descriptor matcher estimates the rigid transform between two point clouds based on the Random Sample Consensus principle.

2.5.4 Registration

Registration is a broad term in the context of 3D computer vision. It refers to the action of aligning two point clouds. The term is typically used when the goal of an alignment is to build a model using multiple point clouds, like mapping a room. This is commonly done using an implementation of some variant of the local pipeline (see section 2.5.1). Figure 2.37 shows a typical local pipeline applied to the registration task. This process takes two inputs, *point cloud A* and *point cloud B*, and returns a single output: *refined alignment*. The refined alignment represent a transformation \mathbf{T}_A^B applied to point cloud B in order to register it with point cloud A.

$$\mathbf{B} \text{ registered to } \mathbf{A} = \mathbf{B} \times \mathbf{T}_A^B$$

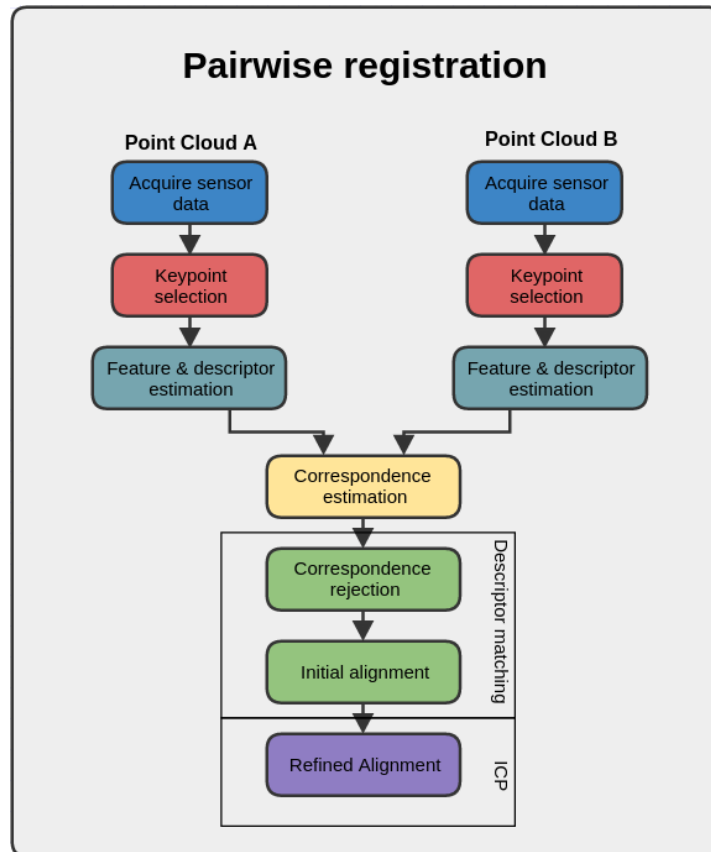


Figure 2.37: A typical local pipeline applied to the registration task.

Figure 2.39 shows the result of registration of multiple point clouds to build a model of a room (the multiple point clouds combined to the full model is shown in Figure 2.38).

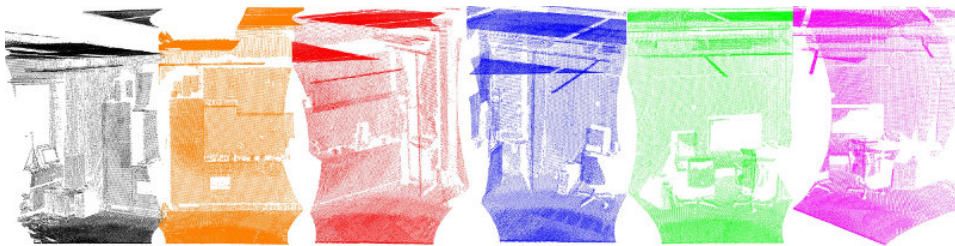


Figure 2.38: Multiple scenes of the same room taken from different viewpoints. Figure from Rusu (2009).

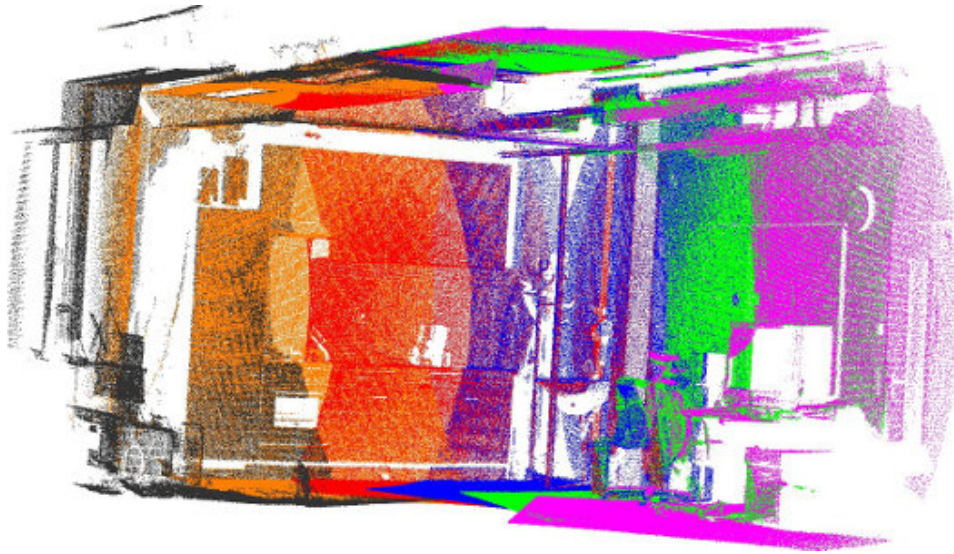


Figure 2.39: Multiple scenes registered to form a complete model of a room. Image from Rusu (2009).

2.5.5 Object detection

Object detection is the act of aligning a model of some object to a section of a scene. The model used for alignment is selected from a previously created training set. The goal of an object detection procedure is to identify the objects present in a scene, and estimate the pose of the object (both orientation and position). In the context of 3D computer vision, this is done by acquiring the correct model from a training set, and register the model to the object located in the scene. This task is preferably done using the global pipeline, since global descriptors contain data describing the complete object cluster (in other words, the task of matching a object cluster to a training model is simpler to fulfill using global descriptors). Figure 2.40 shows a typical global pipeline applied to the object detection task.

Figure 2.41 shows a scene containing multiple object. The result from a object detection on this scene is shown in Figure 2.42 where the training set model is registered onto the original scene, estimating the objects position and orientation.

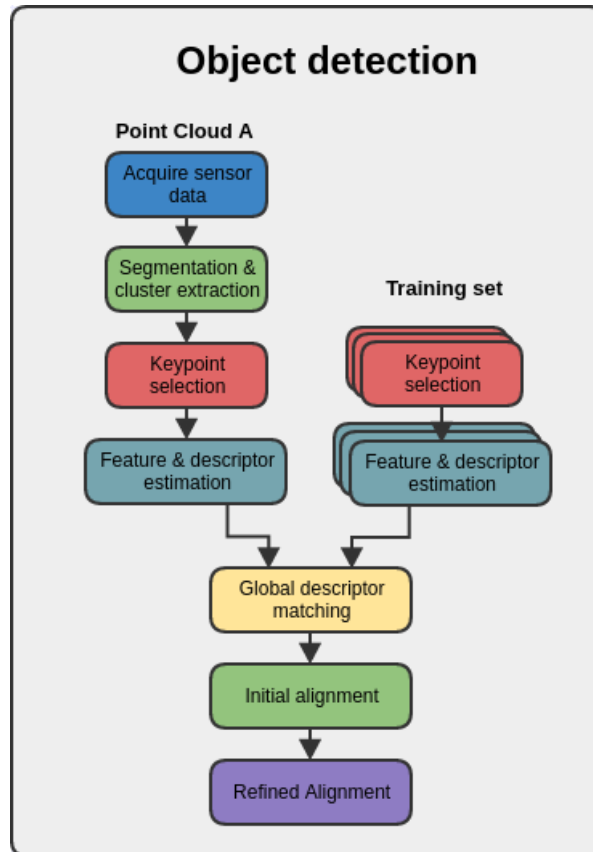


Figure 2.40: A typical global pipeline applied to the object detection task.



Figure 2.41: A scene with multiple objects placed on a table.

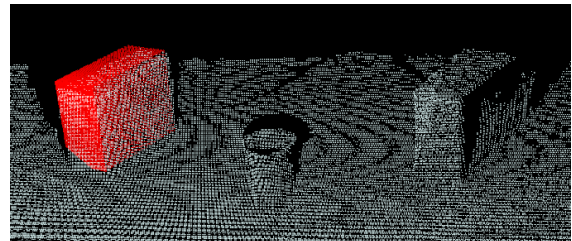


Figure 2.42: A model from a training set is registered onto a detected object, estimating its position and orientation.

2.6 2D computer vision

Computer vision concerns the science and technology of making machines able to see and automatically process visual data (sensed images) in the surrounding environment to recognize objects, track and recover their shape and spatial layout (Cipolla et al., 2010). The goal is to make useful decisions about real physical objects and scenes based on sensed images, as defined by Shapiro and Stockman (2001). Furthermore, Forsyth and Ponce (2003) describes computer vision as the act of extracting descriptions of the world from pictures or sequences of pictures.

The following chapter focuses on algorithms related to object recognition and explains the underlying mathematics and methods developed for different types of keypoint detectors, descriptor extractors and descriptor matching between images.

2.6.1 Scale invariant feature transform

Scale Invariant Feature Transform (SIFT) is the classic approach to image matching. It is considered to be the original detector and descriptor, inspiring development of several alternatives later on. It consists of four major stages of computation as stated in [Lowe \(2004\)](#):

1. Scale-space extrema detection
2. Keypoint localization
3. Orientation assignment
4. Keypoint descriptor

Scale-space extrema detection

Detection of the scale-space of an image is defined from the function, $L(x, y, \sigma)$. This function is produced from the convolution of a variable-scale Gaussian, $G(x, y, \sigma)$, with an input image, $I(x, y)$, and is expressed as:

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y), \quad (2.28)$$

where $*$ is the convolution operation in x and y , and

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2} \quad (2.29)$$

By using the scale-space extrema of a difference-of-Gaussian (DoG) function convolved with the image, it is possible to detect stable keypoint locations in scale space. The proposed function is chosen because every smoothed image, L , needs to be computed in any case for scale space feature description. The DoG function can therefore be computed by image subtraction. This function is defined as $D(x, y, \sigma)$, and can be computed from the difference of two nearby scales separated by a constant multiplicative factor k :

$$D(x, y, \sigma) = (G(x, y, k\sigma) - G(x, y, \sigma)) * I(x, y) = L(x, y, k\sigma) - L(x, y, \sigma) \quad (2.30)$$

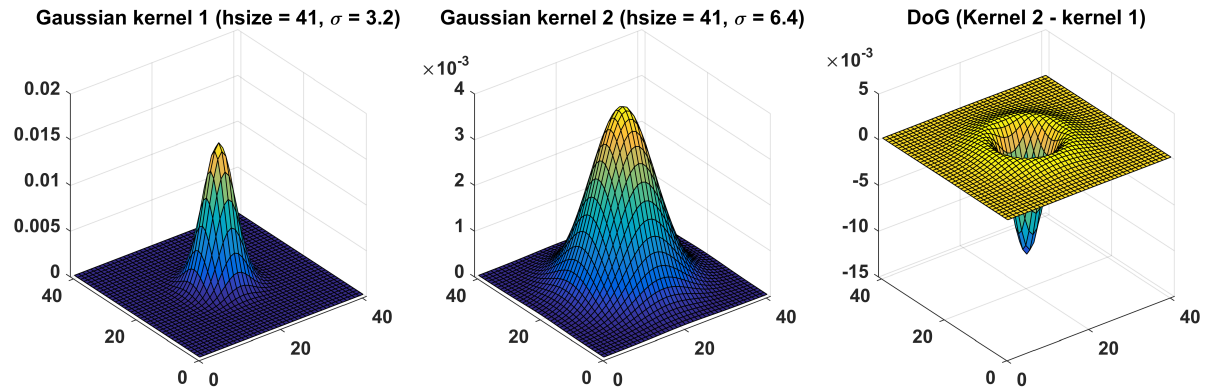


Figure 2.43: Two Gaussian kernels with window size 41×41 . Subtracting kernel one (left), which has $\sigma = 3.2$, from kernel two (middle), which has $\sigma = 6.4$, results in the DoG between them (right). Illustrations generated using MATLAB.

The result from convolving these kernels with an actual image is shown below in Figure 2.44.

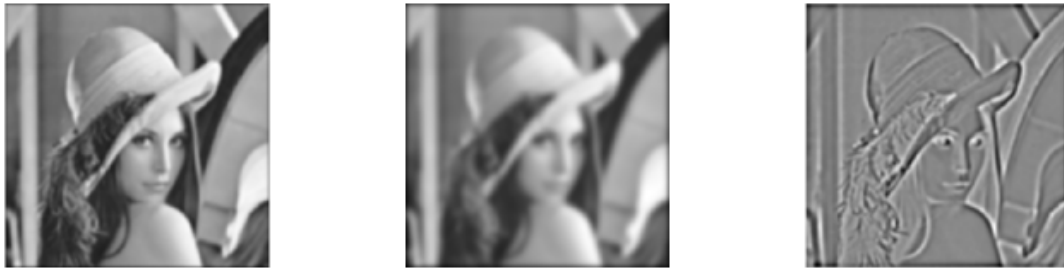


Figure 2.44: Convolution of an image with the kernels illustrated in Figure 2.43. Generated using MATLAB.

Another benefit from the function in equation 2.30 is the close approximation to the scale-normalized Laplacian of Gaussian (LoG), $\sigma^2 \nabla^2 G$. This is beneficial because the normalization of the Laplacian with the factor σ^2 is required for true scale invariance. Moreover, the maxima and minima of $\sigma^2 \nabla^2 G$ produce the most stable image features compared to the gradient, Hessian, or Harris corner function (Lowe, 2004).

To be able to understand the relation between D and $\sigma^2 \nabla^2 G$ the heat diffusion equation parametrized in terms of σ rather than $t = \sigma^2$ can be used:

$$\frac{\partial G}{\partial \sigma} = \sigma^2 \nabla^2 G \quad (2.31)$$

The finite difference approximation to $\partial G / \partial \sigma$ can then be used to compute $\nabla^2 G$, using the difference of nearby scales at $k\sigma$ and σ :

$$\sigma \nabla^2 G = \frac{\partial G}{\partial \sigma} \approx \frac{G(x, y, k\sigma) - G(x, y, \sigma)}{k\sigma - \sigma} \quad (2.32)$$

and therefore,

$$G(x, y, k\sigma) - G(x, y, \sigma) \approx (k - 1)\sigma^2 \nabla^2 G \quad (2.33)$$

The DoG function has scales differing by a constant factor. This implies that the function already incorporates the σ^2 scale normalization required for the scale-invariant Laplacian.

The approach proposed by Lowe (2004) in order to construct the DoG for all scales in every octave is illustrated in Figure 2.45. Each scale in every octave is repeatedly convolved with Gaussians to produce the set of scale space images as shown on the left. As seen on the right in Figure 2.45, the DoG images are computed from subtraction of adjacent Gaussian images. When the current octave has been finished, the Gaussian image is down-sampled by a factor of two, and the process is repeated.

The next step in order to detect the local maxima and minima of $D(x, y, \sigma)$, is to compare each sample point to its eight neighbours in the current image and nine neighbours in the adjacent scales. 3×3 regions at the current and adjacent scales are used. The sample point is selected as an extrema only if it is larger or smaller than all of its neighbours. See Figure 2.46.

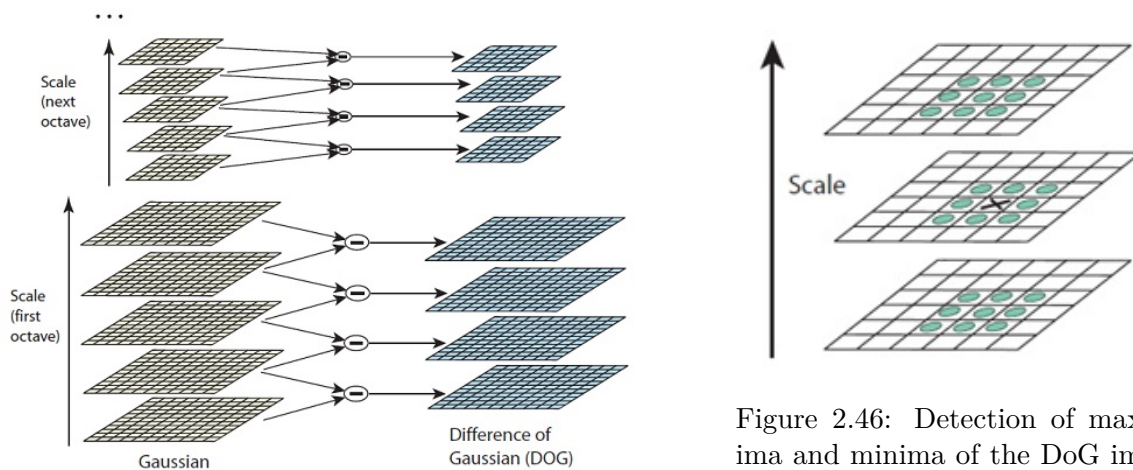


Figure 2.45: Illustration of the DoG from different scales and octaves. Image from Lowe (2004).

Figure 2.46: Detection of maxima and minima of the DoG images. Pixel marked X is the current sample point. Image from Lowe (2004).

Keypoint localization and rejection

The process of finding the minima and maxima determines which pixels that are candidates for keypoints. The next step allows the rejection of bad candidates, assuring that only stable keypoints are used. In Lowe (2004), the suggested approach uses the *Taylor expansion* of the scale-space function, $D(x, y, \sigma)$, shifted so that the origin is at the sample point. The Taylor expansion is expressed as:

$$D(\mathbf{x}) = D + \frac{\partial D}{\partial \mathbf{x}} \mathbf{x} + \frac{1}{2} \mathbf{x}^T \frac{\partial^2 D}{\partial \mathbf{x}^2} \mathbf{x} \quad (2.34)$$

The vector $\mathbf{x} = (x, y, \sigma)^T$ is the offset from the sample point. $D(\mathbf{x})$ and its derivatives are evaluated at the same sample point. The location of the extremum is of interest and can be expressed by taking the derivative of $D(\mathbf{x})$ with respect to \mathbf{x} and setting it to zero:

$$\hat{\mathbf{x}} = -\frac{\partial^2 D^{-1}}{\partial \mathbf{x}^2} \frac{\partial D}{\partial \mathbf{x}} \quad (2.35)$$

Finally the function value of the extremum is used for rejecting unstable extrema with low contrast. Substituting equation 2.35 into 2.34 results in:

$$D(\hat{\mathbf{x}}) = D + \frac{1}{2} \frac{\partial D^T}{\partial \mathbf{x}} \hat{\mathbf{x}} \quad (2.36)$$

Lowe (2004) proposed that all extrema with $D(\hat{\mathbf{x}})$ returning a value less than 0.03 should be discarded.

The rejection of extrema with low contrast alone is not sufficient for stability. Since the DoG function will have a strong response along edges, a method based on a 2×2 Hessian matrix is proposed:

$$\mathbf{H} = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix} \quad (2.37)$$

It is computed at the location and scale of the keypoint, and the derivatives are estimated by taking differences of neighbouring sample points. The ratio of the eigenvalues of \mathbf{H} is of interest. The eigenvalue with the largest magnitude is denoted α and the smaller one is denoted β :

$$\begin{aligned} \text{Tr}(\mathbf{H}) &= D_{xx} + D_{yy} = \alpha + \beta \\ \text{Det}(\mathbf{H}) &= D_{xx}D_{yy} - (D_{xy})^2 = \alpha\beta \end{aligned} \quad (2.38)$$

The sum of eigenvalues is given from the trace of \mathbf{H} and their product is given from the determinant as expressed in equation 2.38. The determinant may be negative, although it is unlikely. However, if this is the case, a point may be discarded as not being an extremum since the curvatures have different signs. As already mentioned the ratio between the eigenvalue with largest magnitude and the smaller one is of interest. It is expressed as $\alpha = r\beta$, where r is the ratio. Using this in relation to equation 2.38 we get:

$$\frac{\text{Tr}(\mathbf{H})^2}{\text{Det}(\mathbf{H})} = \frac{(\alpha + \beta)^2}{\alpha\beta} = \frac{(r\beta + \beta)^2}{r\beta^2} = \frac{(r + 1)^2}{r} \quad (2.39)$$

As evident from the above equation it is only dependable on the ratio of the eigenvalues and not their actual individual values. This permits a rather efficient check of the principal curvatures:

$$\frac{\text{Tr}(\mathbf{H})^2}{\text{Det}(\mathbf{H})} < \frac{(r + 1)^2}{r} \quad (2.40)$$

When the two eigenvalues are equal the expression $(r + 1)^2/r$ is at a minimum. The value increases with r . Lowe (2004) used a value of $r = 10$ in his paper. This means that the keypoints with a ratio between the principal curvatures greater than 10 will be rejected.

Keypoint orientation

This step is important to achieve the invariance to image rotation. It is dependent on a consistent orientation to each keypoint based on local image properties. The proposed approach in Lowe (2004) is applied to a Gaussian smoothed image, L . The smoothed image is chosen to have the closest scale to the scale of the keypoint. This is done to ensure that the computations give scale-invariant results. The gradient magnitude $m(x, y)$, and gradient orientation $\theta(x, y)$, is precomputed using pixel differences. It is computed at the given scale, for each image sample $L(x, y)$:

$$\begin{aligned} m(x, y) &= \sqrt{(L(x + 1, y) - L(x - 1, y))^2 + (L(x, y + 1) - L(x, y - 1))^2} \\ \theta(x, y) &= \tan^{-1} \left(\frac{L(x, y + 1) - L(x, y - 1)}{L(x + 1, y) - L(x - 1, y)} \right) \end{aligned} \quad (2.41)$$

The gradient orientations of sample points within a region around the keypoint is then used to generate an orientation histogram. The histogram is divided into 36 bins, one for every 10 degrees, covering the 360 degree range of orientations. To be able to determine the dominant directions of the local gradients, each sample point is weighted by its gradient magnitude. It is also weighted by a Gaussian-weighted circular window with a smoothing factor, $\sigma = 1.5\sigma_{keypoint}$.

Peaks in the orientation histogram corresponds to dominant directions of the local gradients. A histogram will in some cases have peaks of magnitude close to the dominant peak. If any local peak is within 80% of the dominant peak, an additional keypoint with that orientation is generated. As stated in Lowe (2004), this contributes significantly to the stability of matching. For improved accuracy, a parabola is fit to the 3 histogram values closest to each peak, interpolating the peak position.

Keypoint descriptor

At this final stage, a representation of the local image features is generated from the location, scale and orientation of each keypoint. The descriptor is designed to be highly distinctive, yet robust against significant levels of local shape distortion and change in illumination.

The keypoint descriptor is created by first computing the gradient magnitude and orientation at each image sample point in a region around the keypoint location as previously described. This is illustrated to the left in Figure 2.47 together with the Gaussian-weighted window, indicated by the overlaid circle. The samples are then accumulated into orientation histograms summarizing the contents over 4×4 subregions, with the length of each arrow corresponding to the sum of the gradient magnitudes near that direction within the region. Figure 2.47 illustrates a 2×2 descriptor computed from an 8×8 set of samples. The actual SIFT descriptor is a 4×4 array of histograms computed from a 16×16 sample array.

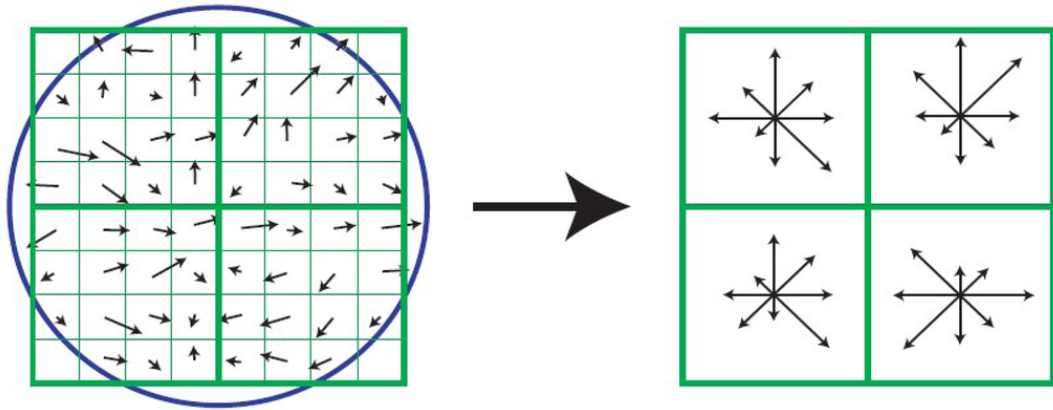


Figure 2.47: Illustration of SIFT descriptor computed from gradient magnitude and orientation. Image from [Lowe \(2004\)](#).

The descriptor is formed from a vector containing the values of all the orientation histogram entries, corresponding to the length of the arrows illustrated on the right side of Figure 2.47. Lowe concluded from his experiments that every subregion in the 4×4 array of histograms should have 8 orientation bins each. This results in a $4 \times 4 \times 8 = 128$ element feature vector for each keypoint.

2.6.2 Speeded-up robust features

Speeded-Up Robust Features (SURF) was introduced by [Bay et al. \(2006b\)](#) and thoroughly explained in [Bay et al. \(2008\)](#). It entered the field of keypoint detectors and descriptors with the goal to outperform the state-of-the-art alternatives, e.g. SIFT, both in terms of computational speed and performance. SURF is, as SIFT, both a detector and a descriptor. It consists of three steps:

1. Interest Point Detection
2. Interest Point Description
3. Matching between different images (see section 2.6.5)

Interest point detection

This step, also referred to as Fast-Hessian Detector ([Bay et al., 2006b](#)) is, as suggested from the name, an approach based on the Hessian matrix. It is however, a very basic Hessian-matrix approximation allowing the use of integral images, reducing the computational time drastically. What this really means is the ability to quickly compute box type convolution filters. The entry of an integral image $I_{\Sigma}(\mathbf{x})$ at a location $\mathbf{x} = (x, y)^T$ represents the sum of all pixels in the input image I within a rectangular region formed by the origin and \mathbf{x} .

$$I_{\Sigma}(\mathbf{x}) = \sum_{i=0}^x \sum_{j=0}^y I(i, j) \quad (2.42)$$

Looking at Figure 2.48, there are four rectangles. The corners of rectangle Σ are marked with letters. Each of these points in the image has a value formed from the sum of the pixel intensities inside a rectangular region. Point **D** has the sum of pixel intensities inside rectangle 1, **B** has the sum from 1 + 2, **C** is the sum of 1 + 3 and **A** is from 1 + 2 + 3 + Σ . The intensity inside rectangle Σ may then be calculated as: $\Sigma = \mathbf{A} - \mathbf{B} - \mathbf{C} + \mathbf{D}$.

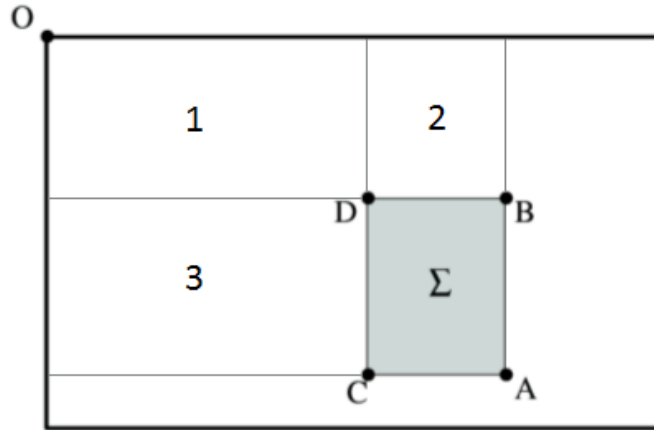


Figure 2.48: Illustration of an integral image. Image from Bay et al. (2008).

Once the integral image has been computed, it is possible to calculate the sum of the intensities over a rectangular area of any size, using only three additions and four memory accesses (reading the intensity at four given points). This implies that the computational time is independent of the size of the rectangular region.

As stated in Bay et al. (2008) the Hessian matrix approach for interest point detection was chosen because of its good performance in accuracy. Structures in the image are chosen as interest points at locations where the determinant of the Hessian (DoH) matrix is maximum. The suggested approach also relies on the determinant of the Hessian for the scale selection.

The Hessian matrix in point (x, y) at scale σ in an image I is expressed as follows:

$$\mathbf{H}(x, y, \sigma) = \begin{bmatrix} L_{xx}(x, y, \sigma) & L_{xy}(x, y, \sigma) \\ L_{xy}(x, y, \sigma) & L_{yy}(x, y, \sigma) \end{bmatrix} \quad (2.43)$$

$L_{xx}(x, y, \sigma)$ is the convolution of the Gaussian second order derivative $\frac{\partial^2}{\partial x^2}g(\sigma)$ with the image I in point (x, y) . Similarly for $L_{xy}(x, y, \sigma)$ and $L_{yy}(x, y, \sigma)$.

In comparison to the approximation to Laplacian-of-Gaussian by Difference-of-Gaussian as performed in SIFT, an approximation is also carried out for the Hessian matrix used in SURF. In addition, real filters are non-ideal in any case, introducing some limitations in practice. An example is the Gaussian second order derivative, which has to be discretized and cropped, leading to a loss in repeatability under image rotations around odd multiples of $\frac{\pi}{4}$. This weakness is valid for Hessian-based detectors in general.

As evident from Bay et al. (2008) the performance of the approximation, using box filters, is comparable to, or better than the performance with discretized and cropped Gaussians. Furthermore, this approximation of the second order Gaussian derivatives is very computationally efficient because of the use of integral images.

Figure 2.49 illustrates the Gaussian second order partial derivative filters in comparison to the approximation using box filters. The illustrated box filters are of size 9×9 and approximates a Gaussian with $\sigma = 1.2$, which represents the lowest scale, used in SURF, for determining the location of interesting structures. This is called a blob response map (Bay et al., 2008). The approximations are denoted by D_{xx} for L_{xx} , D_{yy} for L_{yy} and D_{xy} for L_{xy} .

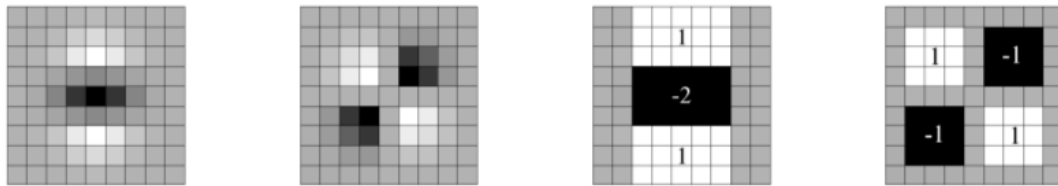


Figure 2.49: The left half shows the discretized and cropped Gaussian second order partial derivative in y - (L_{yy}) and xy -direction (L_{xy}). The right half shows the approximation for the second order Gaussian partial derivative in y - (D_{yy}) and xy -direction (D_{xy}), using box filters. Image from Bay et al. (2008).

The determinant of the approximated Hessian matrix is expressed as:

$$\det(\mathbf{H}_{\text{approx}}) = D_{xx}D_{yy} - (wD_{xy})^2, \quad (2.44)$$

where $w \simeq 0.9$ is a relative weight of the filter responses used to balance the expression for the determinant of the Hessian. This value is kept constant, despite the theoretical incorrectness of doing so. See Bay et al. (2008) for details.

The search of correspondences often requires their comparison in images where they are seen at different scales. This implies that the interest points need to be found at different scales. Scale spaces are typically implemented as an image pyramid, however, the implementation of this in SURF differs from SIFT. As explained in Lowe (2004), the images are repeatedly convolved with a Gaussian kernel for the current pyramid octave, then the image is down-sampled and the process is repeated for the new octave. In SURF, the scale space is analysed by up-sampling the filter size rather than iteratively reducing the image size, as illustrated in Figure 2.50. The latter approach is possible because of the use of integral images, and was chosen because of its computational efficiency. The computation time is constant independent of filter size (Bay et al., 2008).

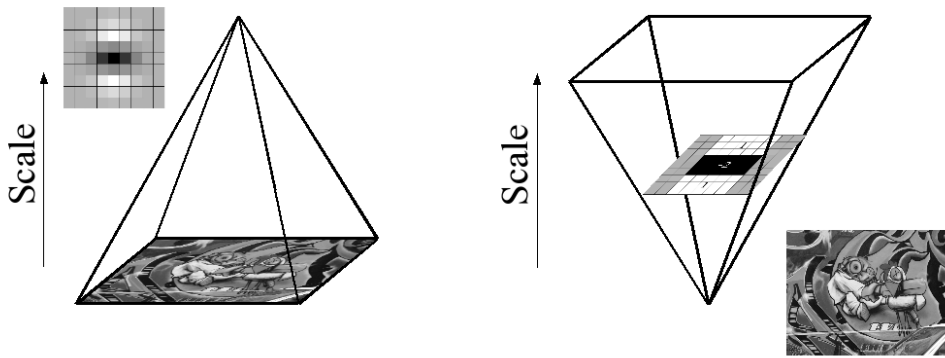


Figure 2.50: Iteratively reducing the image size as in SIFT (left). The use of integral images allows the up-sampling of the filter (right). Image from Bay et al. (2008).

As already mentioned, the initial scale layer of the scale space is the output of the 9×9 filter, approximating Gaussian derivatives with $\sigma = 1.2$, hereby referred to as scale $s = 1.2$ for the approximation. Furthermore, the scale space is divided into octaves, consisting of a series of filter response maps obtained from convolving the same input image with a filter of increasing size. To ensure the existence of the central pixel, the filter mask size must increase by a total of 6 pixels from one layer to the next as illustrated in Figure 2.51 (Bay et al., 2008). The first octave therefore consists of images filtered with mask sizes 9×9 , 15×15 , 21×21 and 27×27 . However, for each new octave, the filter size increase is doubled, going from 6 to 12 to 24 to 48. The filter sizes in three successive octaves are illustrated in Figure 2.52. See Bay et al. (2008) for more details.

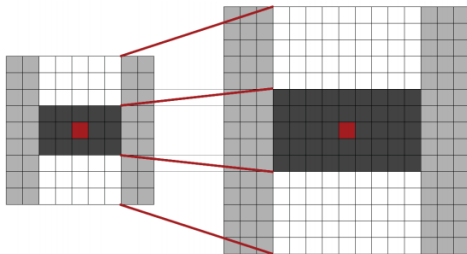


Figure 2.51: The length of the dark lobe can only be increased by an even number of pixels to guarantee the presence of the central pixel. Mask size 9×9 (left) and 15×15 (right). Image from Bay et al. (2006a).

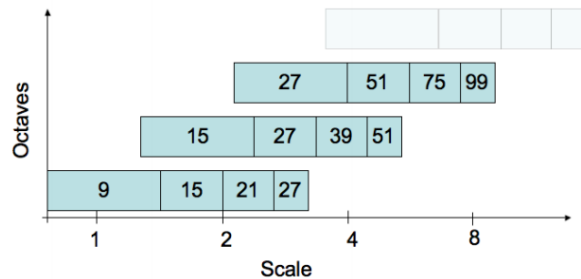


Figure 2.52: Graphical representation of the filter side lengths for three successive octaves. The octaves are overlapping in order to cover all possible scales seamlessly. Image from Bay et al. (2008).

With the complete scale space in place, the interest points can be localized by applying a non-maximum suppression in a $3 \times 3 \times 3$ neighbourhood as illustrated in Figure 2.46 in section 2.6.1. The maximum of the determinant of the Hessian matrix are then interpolated in scale and image space due to the relatively large difference in scale between the first layer of every octave (Bay et al., 2008). The interpolated location of the interest point is computed in the

same way as in SIFT using equation 2.34 and 2.35 where $\mathbf{x} = (x, y, s)$. See Bay et al. (2006a) for more details.

Interest point description

The proposed descriptor, Speeded-Up Robust Features (SURF), describes the distribution of the intensity content within the interest point neighbourhood. This is similar to the gradient information extracted by SIFT, however, the descriptor is built on the distribution of first order Haar wavelet responses in x and y rather than the gradient (Bay et al., 2008).

The Haar wavelet responses within a circular neighbourhood of $6s$ (s is the scale of the approximate Gaussian filter) around the interest point is used to identify a reproducible orientation of the point. This has to be done in order to make the descriptor invariant to image rotation. Note that s is the scale at which the interest point was detected. The sampling step is scale dependent and chosen to be equal to s . The size of the wavelets are also scale dependent and set to a side length of $4s$. Again, the use of integral images for fast filtering is possible, fulfilling the goal of keeping the computational time low compared to previously proposed schemes.

Calculation of the wavelet responses in x and y direction is performed by filtering the images, using Haar wavelet filters illustrated in Figure 2.53. The responses are smoothed with a Gaussian $\sigma = 2s$ around the interest point and represented as points in space with horizontal and vertical response strength. A sliding orientation window of size $\frac{\pi}{3}$ is used to find the dominant orientation of the interest point. From each circle segment the (x, y) components are summed up, yielding a local orientation vector as illustrated in Figure 2.54.

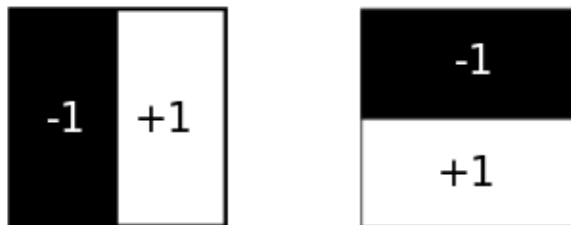


Figure 2.53: Haar wavelet filters to compute the responses in x (left) and y (right) direction. The weights of the dark and bright parts are illustrated.

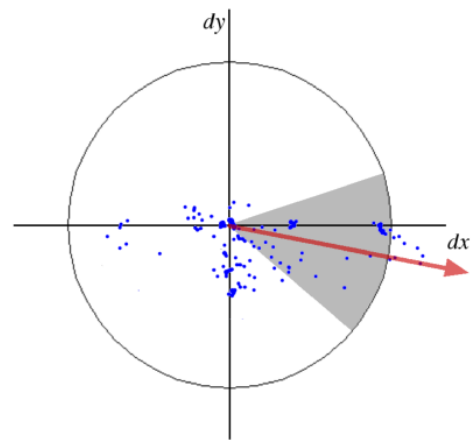


Figure 2.54: The dominant orientation of the Gaussian weighted Haar wavelet response is detected within a sliding orientation window. Image from Bay et al. (2008).

At this point the descriptor can be extracted. A square region is centred around the interest point oriented along the computed dominant orientation of the interest point. This window has a size of $20s$. This region is then split up regularly into $4 \times 4 = 16$ square sub-region, as illustrated in Figure 2.55. For each sub-region at 5×5 regularly spaced sample points,

Haar wavelet responses are computed. The Haar wavelet responses in horizontal and vertical direction, in relation to the selected interest point orientation, is denoted d_x and d_y . To increase robustness the responses are first weighted with a Gaussian $\sigma = 3.3s$ centred at the interest point. The wavelet responses, d_x and d_y , are then summed up over each sub-region to form the first set of entries in the feature vector. Each sub-region has a four-dimensional descriptor vector expressed as $\mathbf{v} = (\sum d_x, \sum d_y, \sum |d_x|, \sum |d_y|)$. This applies to all 4×4 sub-regions, resulting in a vector of length $4 \times 4 \times 4 = 64$. The remaining two vector elements, $|d_x|$ and $|d_y|$, is the sum of the absolute values of the responses. These entries are needed in order to include information about the polarity of the intensity changes. See Figure 2.56.

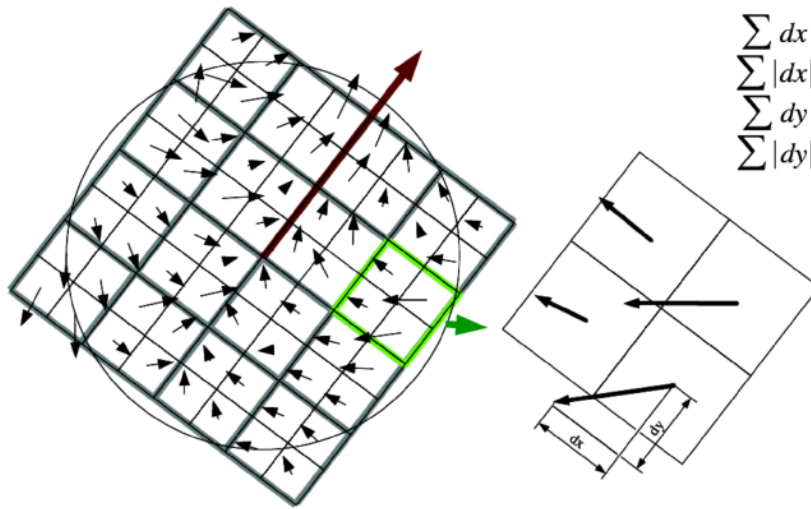


Figure 2.55: Left: An oriented quadratic grid with 4×4 square sub-regions centred around the interest point. Right: The actual fields of the descriptor, the sums d_x , d_y , $|d_x|$ and $|d_y|$, are computed for each sub-region relative to the orientation of the grid. The sub-regions in this figure are 2×2 instead of 5×5 for reasons of illustration. Image from Bay et al. (2008).



Figure 2.56: Illustration of the nature of a SURF descriptor. Left: A homogeneous region will make all values relatively low. Middle: Frequencies in x direction will make the value of $\sum |d_x|$ high, but all others remain low. Right: In the case of a gradually increasing intensity in x direction, both values $\sum d_x$ and $\sum |d_x|$ are high. Image from Bay et al. (2008).

2.6.3 Binary robust invariant scalable keypoints

Binary Robust Invariant Scalable Keypoints (BRISK) is a relatively new keypoint detector and descriptor. It was proposed at the International Conference on Computer Vision (ICCV) in 2011 by [Leutenegger et al. \(2011\)](#). As SURF was developed to outperform SIFT in terms of computational cost and performance, BRISK seeks to improve on the computational time needed and still deliver high performance under a variety of image transformations. Evaluation on benchmarks show that BRISK can be computed at an order of magnitude faster than SURF at comparable matching performance in some cases ([Leutenegger et al., 2011](#)). The contents of this section will focus on the theory and methods behind the detection and description of BRISK keypoints, divided in two steps:

1. Scale-space keypoint detection
2. Keypoint description

Scale-space keypoint detection

[Leutenegger et al. \(2011\)](#) proposed a detection methodology with the goal of achieving an efficient computation of keypoints. It is inspired by a detector by [Mair et al. \(2010\)](#) called *Adaptive and Generic Corner Detection Based on the Accelerated Segment Test* (AGAST). This is an extension for accelerated performance of a detector called *Features from Accelerated Segment Test* (FAST) by [Rosten and Drummond \(2006\)](#). Scale invariance is crucial for high-quality keypoints. However, FAST and AGAST is not invariant to scale. To overcome this drawback [Leutenegger et al. \(2011\)](#) introduces the search for maxima not only in the image plane, but also in scale-space using the FAST score s as a measure for saliency.

The scale-space pyramid layers consist of the following for $i = \{0, 1, \dots, n - 1\}$ and typically $n = 4$:

- n octaves c_i , which are formed by progressively half-sampling the original image c_0
- n intra-octaves d_i , which are located in-between layers c_i and c_{i+1}

By down-sampling the original image c_0 by a factor of 1.5 the first intra-octave d_0 is obtained. The rest of the intra-octaves are derived by successive half-sampling. Therefore, if t denotes scale then $t(c_i) = 2^i$ and $t(d_i) = 2^i \cdot 1.5$. See Figure 2.58 for an illustration of the octaves and intra-octaves.

As already mentioned, the BRISK detector is based on the ideas from FAST and inspired by the computational efficiency of AGAST. In both these detectors, corners are detected by checking the intensity of pixels in a circle around a current sample pixel p illustrated in Figure 2.57. A typical mask is the FAST 9-16 mask, which requires at least 9 of the 16 pixels in the circle to be either brighter or darker than pixel p by a given threshold. This mask provides the best performance according to [Rosten and Drummond \(2006\)](#), and is the mask used for most of the detection in BRISK ([Leutenegger et al., 2011](#)). The exception is for detection of interest points below the first octave. In this case the FAST 5-8 mask is used to obtain FAST scores as a *virtual* layer below this octave. These scores are only needed in order to fit a parabola for scale refinement ([Fan et al., 2015](#)), not for non-maxima suppression in the first octave.

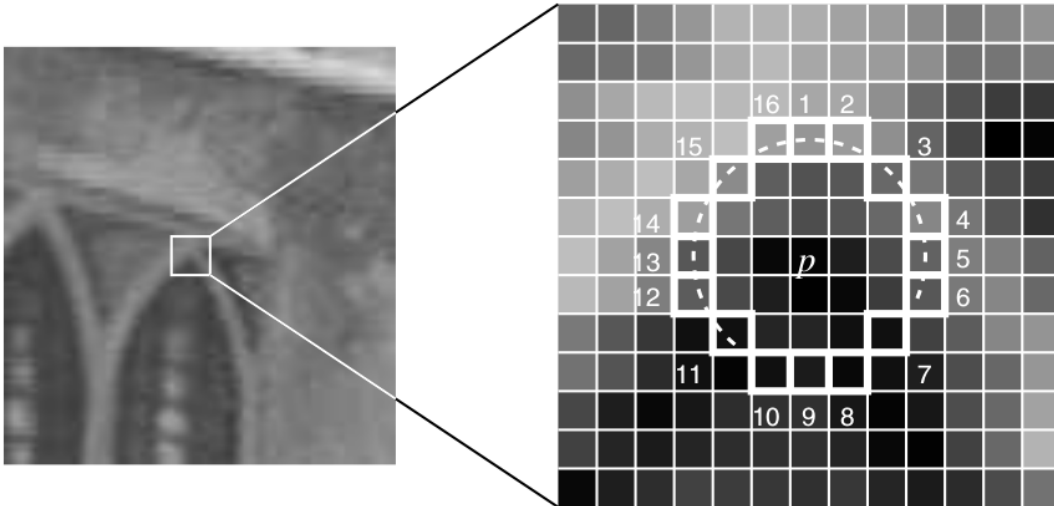


Figure 2.57: Illustrates a segment test corner detection using a 12-16 mask. The dotted arc indicates 12 pixels which are brighter than pixel p by more than a given threshold. Image from Rosten and Drummond (2006).

To detect potential regions of interest, each octave and intra-octave is processed with the FAST 9-16 detector separately using the same threshold T . The points belonging to these regions are then evaluated by applying a non-maxima suppression in scale-space. This means that the FAST score s of the current sample point needs to be larger than the FAST score of its 8 neighbouring points in the same layer. In addition the FAST scores in the layer above and below needs to be lower than the current sample point. See Rosten and Drummond (2006) for an in depth explanation of the FAST score. The check is performed inside equally sized square patches of 2 pixels side-length in the layer with the suspected maximum. The neighbouring layers are discretized differently, which is dealt with by interpolation at the boundaries of the patch. See Figure 2.58. Furthermore, to determine the true scale of the keypoint, the local saliency maximum in all three layers of interest is sub-pixel refined (by fitting a 2D quadratic function in the least-squares sense to each of the three score-patches) before a 1D parabola is fitted along the scale-axis. This is illustrated in Figure 2.58. Finally, the location of the keypoint is re-interpolated between the patch maxima closest to the determined scale (Leutenegger et al., 2011).

Keypoint description

Compared to SIFT and SURF, this descriptor is different, especially in terms of matching. Matching will be explained in section 2.6.5. The difference is present mainly because the descriptor is composed as a binary string. Given a set of keypoints, the string is generated by concatenating the results of simple brightness comparison tests. The approach is inspired by a descriptor called Binary Robust Independent Elementary Features (BRIF) by Calonder et al. (2010), which is efficient to compute, but not invariant to scale or rotation.

The detected keypoints as detected in scale-space described in section 2.6.3 needs to be processed before building the descriptor bit-string. A sampling pattern with N locations equally spaced on circles concentric with the keypoint is used. Each of these locations are points \mathbf{p}_i

in the pattern. Every point \mathbf{p}_i is smoothed with a Gaussian σ_i proportional to the distance between the points on the respective circle. This smoothing is performed to avoid aliasing effects when sampling the image intensity of a point (Fan et al., 2015). Figure 2.59 illustrates the sampling pattern.

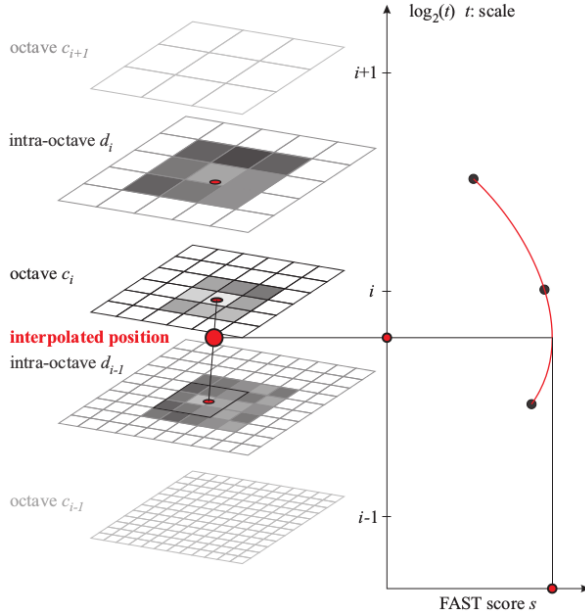


Figure 2.58: Scale-space interest point detection illustrated. The 1D parabola is fitted along the scale-axis to determine the true (interpolated) scale of the keypoint. Image from Leutenegger et al. (2011).

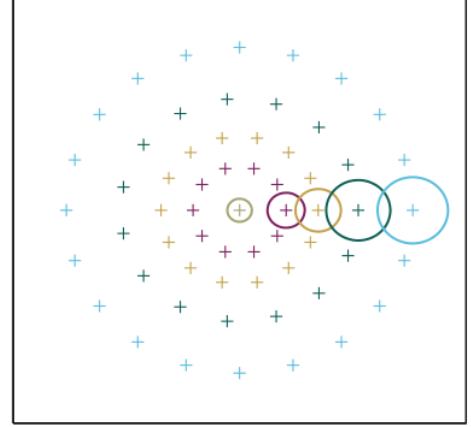


Figure 2.59: A sampling pattern with $N = 60$ sampling points including the center point, equally spaced on four concentric circles around the keypoint. For clarity, only one point in each circle is marked with a circle denoting the radius σ of the Gaussian kernel used to smooth the intensity values of the sampling points. Image from Fan et al. (2015).

For a keypoint k in the image, we consider one of the $N \cdot (N - 1) / 2 = 1770$ sampling-point pairs $(\mathbf{p}_i, \mathbf{p}_j)$. The local gradient is expressed as:

$$\mathbf{g}(\mathbf{p}_i, \mathbf{p}_j) = (\mathbf{p}_j - \mathbf{p}_i) \cdot \frac{I(\mathbf{p}_j, \sigma_j) - I(\mathbf{p}_i, \sigma_i)}{\|\mathbf{p}_j - \mathbf{p}_i\|^2} \quad (2.45)$$

where $I(\mathbf{p}_i, \sigma_i)$ and $I(\mathbf{p}_j, \sigma_j)$ are the smoothed intensity values at the points \mathbf{p}_i and \mathbf{p}_j . A set of all sampling-point pairs is expressed as:

$$\mathcal{A} = \{(\mathbf{p}_i, \mathbf{p}_j) \in \mathbb{R}^2 \times \mathbb{R}^2 \mid i < N \wedge j < i \wedge i, j \in \mathbb{N}\} \quad (2.46)$$

From set \mathcal{A} a subset of short-distance pairings \mathcal{S} and another subset of L long-distance pairings \mathcal{L} is defined. The threshold distances determining which subset a sampling-point pair belongs to is set to $\delta_{max} = 9.75t$ and $\delta_{min} = 13.67t$ where t is the scale of keypoint k . The subsets are

illustrated in Figure 2.60 and expressed mathematically as:

$$\begin{aligned}\mathcal{S} &= \{(\mathbf{p}_i, \mathbf{p}_j) \in \mathcal{A} \mid \|\mathbf{p}_j - \mathbf{p}_i\| < \delta_{max}\} \subseteq \mathcal{A} \\ \mathcal{L} &= \{(\mathbf{p}_i, \mathbf{p}_j) \in \mathcal{A} \mid \|\mathbf{p}_j - \mathbf{p}_i\| < \delta_{min}\} \subseteq \mathcal{A}\end{aligned}\quad (2.47)$$

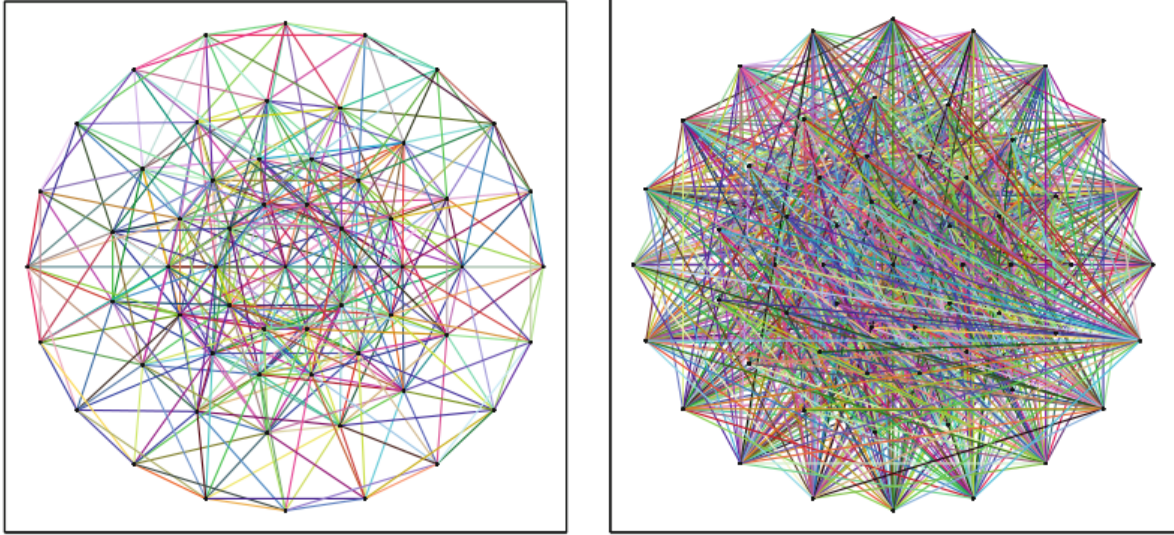


Figure 2.60: The set of short-distance pairs, \mathcal{S} , of sampling points used for constructing the descriptor is illustrated to the left. The set of long-distance pairs, \mathcal{L} , of sampling points used for computing orientation is illustrated to the right. Each colour indicates a pair. Image from Fan et al. (2015).

The subset of L long-distance point pairs are then used to determine the overall characteristic pattern direction of the keypoint k . The computation is done by iterating through subset \mathcal{L} , and is expressed as:

$$\mathbf{g} = \begin{pmatrix} g_x \\ g_y \end{pmatrix} = \frac{1}{L} \cdot \sum_{(\mathbf{p}_i, \mathbf{p}_j) \in \mathcal{L}} \mathbf{g}(\mathbf{p}_i, \mathbf{p}_j) \quad (2.48)$$

In order to build up the descriptor, the sampling pattern as explained above is applied with a rotation by $\alpha = \text{atan2}(g_y, g_x)$ around the keypoint k . Then the bit-vector descriptor d_k is computed by processing all the short-distance intensity comparisons of point pairs in the rotated pattern $(\mathbf{p}_i^\alpha, \mathbf{p}_j^\alpha) \in \mathcal{S}$. Determining the state of each bit is performed as follows

$$b = \begin{cases} 1, & I(\mathbf{p}_j^\alpha, \sigma_j) > I(\mathbf{p}_i^\alpha, \sigma_i) \\ 0, & \text{otherwise} \end{cases} \quad (2.49)$$

where:

$$\forall (\mathbf{p}_i^\alpha, \mathbf{p}_j^\alpha) \in \mathcal{S}$$

The above equations (2.45-2.49) are used to generate a BRISK descriptor, differing from BRIEF by being both scale- and rotation invariant. The more in depth differences are explained in the original paper (Leutenegger et al., 2011). Usage of the method as explained in this section yields a descriptor bit-string of length 512.

2.6.4 Oriented FAST and rotated BRIEF

Oriented FAST and Rotated BRIEF (ORB) was proposed at ICCV in 2011 by Rublee et al. (2011). Like BRISK, this is also a detector and descriptor developed to outperform SIFT in terms of computational cost. The authors claim that ORB is an efficient alternative to SIFT or SURF. It is evident from Rublee et al. (2011) that the computational time is over two orders of magnitude faster than SIFT. This boost in processing speed is a result of using FAST and BRIEF as base for detection and description, respectively. ORB is, like BRISK, invariant to scale and rotation. Scale-invariance is achieved by employing a scale pyramid of the image. Rotation invariance is achieved by using a measure of corner orientation called the *intensity centroid*, originally presented in Rosin (1999). Furthermore, the binary descriptor is built by comparing intensities between two sampling patterns, similar to BRISK. Moreover, ORB does this by using a different sampling and feature selection strategy. This section will present the theory and methods behind this strategy, in two parts:

1. FAST Keypoint Orientation (oFAST)
2. Rotation-Aware BRIEF (rBRIEF)

FAST keypoint orientation (oFAST)

Features from Accelerated Segment Test (FAST) as proposed by Rosten and Drummond (2006) is the method of choice for finding keypoints with minimal computational cost. By evaluating a circle of 16 pixels around a center pixel, it can be determined if this center pixel is a corner or not. If the intensities of the pixels in the circle are brighter or darker compared to the central pixel, by a given threshold, it is considered to be a corner. See Figure 2.57. Typically, at least 9 or 12 of these pixels must fulfill this test, referred to as FAST 9-16 and FAST 12-16 respectively.

As already mentioned, FAST is the base for the ORB detector, and in order to acquire invariance to scale, a simple scale pyramid is used. Depending on the implementation, this pyramid may vary in number of levels and scale factors between each level. For example n levels with a scale factor equal to 1.2 will result in a pyramid where the original image is first down-scaled by a factor of 1.2, then the result from this is down-scaled by a factor of 1.2 and so on until the pyramid has been filled with n images, as illustrated in Figure 2.61. For each level of the pyramid, FAST features are produced and then filtered. The detector of choice in ORB is the FAST 9-16. To filter out unstable features, a Harris corner measure is employed (Harris and Stephens, 1988). This is done by setting a threshold of N keypoints. The threshold must be set low enough so that N is lower than the total amount of keypoints in the image. The detected keypoints are ranked and ordered according to the Harris measure, and the top N points are retained. Figure 2.62 shows an image that has been processed with the ORB detector over a 5 level pyramid with scale factor 1.2. The detected keypoints were then filtered with a threshold $N = 200$.

The proposed approach to assign an orientation to each keypoint uses a measure of corner orientation called the *intensity centroid*. This measure assumes that the intensity of a corner is offset from its center. A vector from corner center to this offset point may be used to assign an orientation. The moments of an image patch is expressed as presented in Rosin (1999):

$$m_{pq} = \sum_{x,y} x^p y^q I(x, y), \quad (2.50)$$

which is used to define the centroid as:

$$\mathbf{C} = \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}} \right) \quad (2.51)$$

A vector from the corner keypoint center \mathbf{O} , to the centroid \mathbf{C} , is then denoted $\overrightarrow{\mathbf{OC}}$. The orientation of the patch is:

$$\theta = \text{atan2}(m_{01}, m_{10}) \quad (2.52)$$

To further improve the rotation invariance of the above measure, Rublee et al. (2011) proposed to compute the moments within a circular region of radius r corresponding to the patch size. This means that x and y run from $[-r, r]$. The orientation of keypoints is illustrated in Figure 2.62.

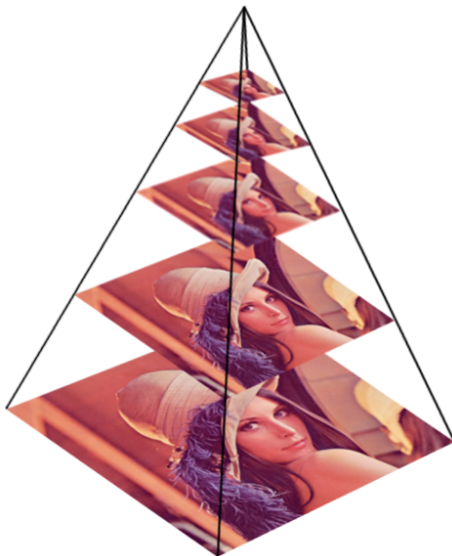


Figure 2.61: Pyramid with 5 levels. A scale factor close to 1 will need more pyramid levels to cover a large scale range, thus increasing the computational cost. A large scale factor will on the other hand weaken the invariance to scale.

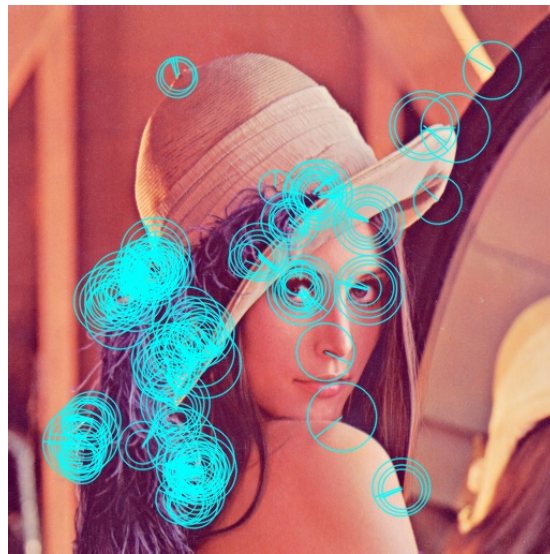


Figure 2.62: Keypoints detected with the ORB detector. The cyan rings denote keypoints with its respective orientation. Notice that some keypoints of differing scale overlaps each other in a concentric manner, which means they are detected from different levels of the pyramid.

Rotation-aware BRIEF (rBRIEF)

As mentioned in the introduction to this section, BRIEF is the base for description in ORB. Providing a set of keypoints with detected scale and rotation as previously explained, the ORB descriptor can be computed by first extracting a scale and rotation normalized local patch. The descriptor is then computed on the patch. The standard way of computing a BRIEF descriptor is by randomly selecting 256 test pairs in a smoothed image patch. The intensity of the two pixels in a pair is then compared to each other yielding the bit value of that test pair. However, this approach is not a good choice for ORB. As explained in [Fan et al. \(2015\)](#), the orientation of ORB keypoints is computed based on the intensities of the described patch. Therefore, the intensity relationship between the rotated pairs of positions used in BRIEF will move toward some fixed pattern. This implies that there are correlations among these position pairs that are used for computing the binary descriptor. To reduce the correlations among the binary tests, [Ruble et al. \(2011\)](#) has developed a learning method for choosing a good subset of binary tests.

Given an extracted local image patch of size $m \times m$. All possible tests from the patch is a pair of $w \times w$ sub-windows of the patch. The number of possible sub-windows is then given by $N = (m - w)^2$. Typically, $m = 31$ and $w = 5$. Pairs of two are selected from these sub-windows, giving $\binom{N}{2}$ binary tests. Tests that overlap are eliminated, yielding a final set of candidate bit features. It is important to smooth the image before performing the tests ([Ruble et al., 2011](#)). Based on a training set, ORB selects at most 256 bits according to the following algorithm:

1. Run each test against all training patches.
2. Order the tests by their distance from a mean of 0.5, forming the vector T.
3. Greedy search:
 - (a) Put the first test into the result vector R and remove it from T.
 - (b) Take the next test from T, and compare it against all tests in R. If its absolute correlation is greater than a threshold, discard it; else add it to R.
 - (c) Repeat the previous step until there are 256 tests in R. If there are fewer than 256, raise the threshold and try again.

The algorithm is a greedy search for a set of uncorrelated tests with means near 0.5. The result is called Rotation-Aware BRIEF (rBRIEF). As evident in the paper by [Ruble et al. \(2011\)](#) this algorithm clearly reduces the correlation between tests making each test contribute to the result. It also raises the variance of binary tests yielding a more discriminative descriptor.

2.6.5 Descriptor matching

In terms of detection of an object in a scene using keypoints and descriptors, a matching procedure is a must. This usually happens by comparing the descriptors from a query image (object) with the descriptors from a training image (scene). Typically, some sort of distance measurement of the descriptors is used for comparison. As previously presented in this thesis, there are descriptors expressed as a string of bits generated from a pixel intensity test, referred to as binary descriptors. The other type is descriptors expressed as a feature vector built from

e.g. an orientation histogram or sums of Haar wavelet responses, referred to as real-valued descriptors. The binary descriptors are those based on BRIEF, i.e. BRISK and ORB. The real-valued descriptors are SIFT and SURF. The method of distance measurement used for these two groups of descriptors differ. This thesis will not cover the details of these methods, however they can be summarized as:

- Binary
 - **Hamming distance** - Checks the amount of symbols that are different at corresponding positions in two strings of equal length. As an example, the hamming distance between the two bit-strings 01011100 and 01010101 is equal to 2.
- Real-valued
 - **Manhattan distance (L_1 norm)** - Also known as *Taxicab distance*. It is a measure of distance between two points in a rectilinear system. For a 2D plane the distance between two points is therefore the distance in x direction added to the distance in y direction. Considering descriptors, these points are actually vectors. Two descriptor vectors describe a feature \mathbf{q} in the query image and a feature \mathbf{t} in the training image. As expressed in [Nixon and Aguado \(2012\)](#), the Manhattan distance is the sum of the modulus of the differences between the n element descriptor of \mathbf{q} and \mathbf{t} :

$$d_M = \sum_{i=1}^n |\mathbf{q}_i - \mathbf{t}_i| \quad (2.53)$$

This method is computationally more efficient than the *Euclidean distance*.

- **Euclidean distance (L_2 norm)** - An alternative to the *Manhattan distance*. It measures the straight line between two points and yields only one solution to the shortest path. Considering two n element descriptors of feature \mathbf{q} and \mathbf{t} , the difference d between the descriptors is expressed as in [Nixon and Aguado \(2012\)](#):

$$d_E = \sqrt{\sum_{i=1}^n (\mathbf{q}_i - \mathbf{t}_i)^2} \quad (2.54)$$

This method is computationally more costly than the *Manhattan distance*.

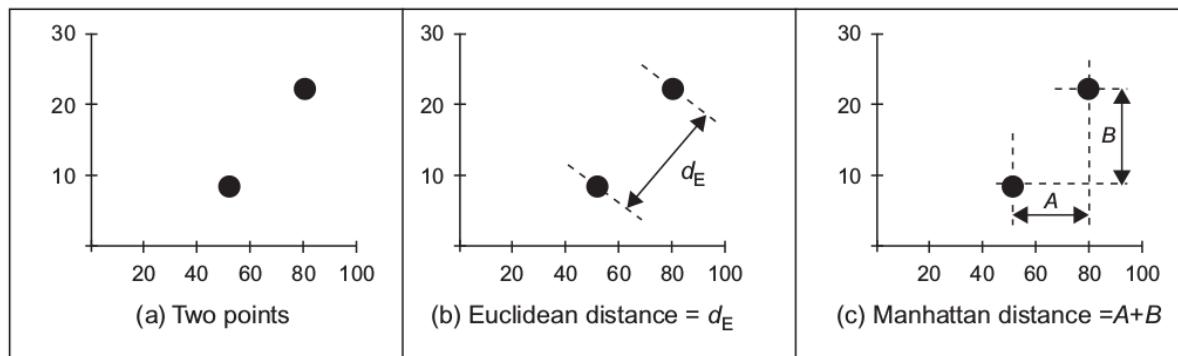


Figure 2.63: Distance measure in a 2D plane. Image from [Nixon and Aguado \(2012\)](#).

Brute-force

The idea of brute-force matching of descriptors is simply to match one feature in the query image with all other features in the training image using one of the distance measurements as described in the previous section. The closest one is returned as a match. A lower distance means better match. However, this matching approach may accept some false positives. The result can be improved by sorting the matches by distance ([OpenCV, 2015a](#)).

Brute-force matching can also return more than one match if that is desirable. The result is then processed with a ratio test of k best matches as explained by [Lowe \(2004\)](#). Considering $k = 2$, the two closest descriptors are returned as candidate matches based on their measured distance. Given a distance ratio, the two candidates can be compared to one another. If the measured distance is low for the best candidate, but much larger for the second best candidate, the best candidate is accepted as a match. Both the candidates are rejected if the measured distance is similar. [Lowe \(2004\)](#) proposed to reject all candidates in which the distance ratio is greater than 0.8, which means that the distance of the second best candidate can not be closer than 80% of the best candidate. This leads to an elimination of 90% of the false positives while only discarding up to 5% of the correct matches. Figure 2.64 shows the difference between brute-force matching with a distance ratio of 0.9 and 0.7. There are clearly fewer false positives with a lower distance ratio.

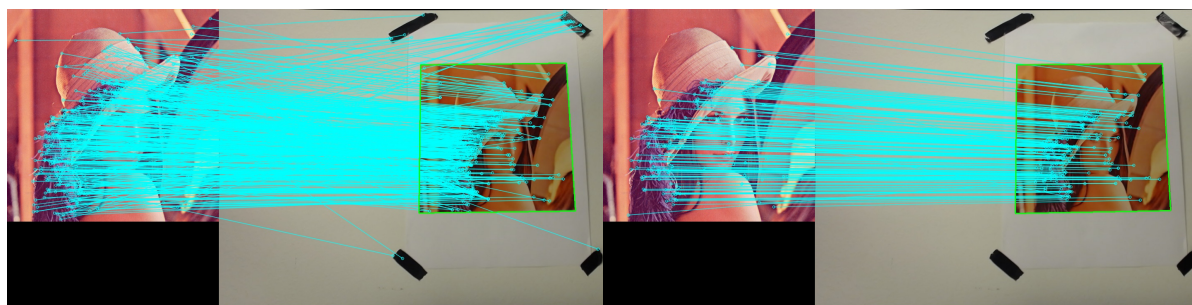


Figure 2.64: Brute-force matching of SIFT descriptors using a distance ratio of 0.9 (left) and 0.7 (right). The cyan lines denote a match between the object and the scene. Notice that some of the lines represents false positives as shown in the left half of the figure, while there are no false positives clearly represented in the right half.

FLANN

An alternative to brute-force matching is Fast Library for Approximate Nearest Neighbours (FLANN). This library contains a collection of algorithms optimized for fast nearest neighbour search in large data sets and for high dimensional features. FLANN is faster than brute-force for matching across large data sets (OpenCV, 2015a). For matching between two images, brute-force may in most cases be the best choice, however in case of a large database of descriptors from numerous images to match across, FLANN is the clear choice. Just as with brute-force matching returning k best matches, the best matches from a FLANN matching procedure may also be processed with a ratio test.

2.6.6 Planar homography

In short, planar homography can be described as a projective transformation between the corresponding points in two planes. The two planes may for example be a set of points from an image of the same object, but with different perspective, or position of the camera. This means that there are world points or features corresponding in the two different camera projections (Corke, 2013). Typically, homographies are computed by matching features between two images. The matched features of each image are then fitted to a plane using e.g. RANSAC, and the homography is then the projective transformation between the two planes. Considering a set of points in the two planes as ${}^1\mathbf{p}_i$ and ${}^2\mathbf{p}_i$. The relationship between them are then expressed as

$${}^2\tilde{\mathbf{p}}_i \simeq \mathbf{H} {}^1\tilde{\mathbf{p}}_i \quad (2.55)$$

where ${}^2\tilde{\mathbf{p}}_i$ and ${}^1\tilde{\mathbf{p}}_i$ is on the form $(x, y, 1)^T$ and $(x', y', 1)^T$ respectively. The homography matrix is a non-singular 3×3 matrix expressed as

$$\mathbf{H} = \begin{pmatrix} H_{11} & H_{12} & H_{13} \\ H_{21} & H_{22} & H_{23} \\ H_{31} & H_{32} & 1 \end{pmatrix} \quad (2.56)$$

The above matrix has 8 unknowns, which can be estimated from 4 world points and their corresponding image points in the two planes (Corke, 2013).

2.7 Robot operating system

Robot Operating System or ROS for short, is a large, community developed framework that works to make writing code for robots easier. ROS is a collection of tools, libraries and conventions that is put together to aid the development of robot software. The goal of the ROS project is to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms. In order to fully grasp the concept of ROS, a couple of key elements needs to be explained further. Full documentation of ROS is available at (ROS.org, 2016a).

2.7.1 The ROS architecture

The *Robot Operating System* project is, as the name implies, structured as an operating system. It runs on a wide variety of Linux distributions, and let developers run self written programs. The ROS framework implements conventions for:

- How programs should be written to run on ROS
- How different programs can communicate with each other
- How programs should log and report error messages

These conventions make it trivial to communicate cross-application, which is valuable when programming complex systems. This means that a system can consist of smaller and self sufficient programs instead of one large program. This has benefits both in the development phase (the developer is allowed to focus on a single task, instead of trying to implement a complete software solution for the entire system), and debugging phase.

Figure 2.65 attempts to illustrate the ROS node architecture. It illustrates how ROS is communicating with the robotic manipulator through a PLC (Wikipedia, 2015) and how different nodes can communicate with each other and external input.

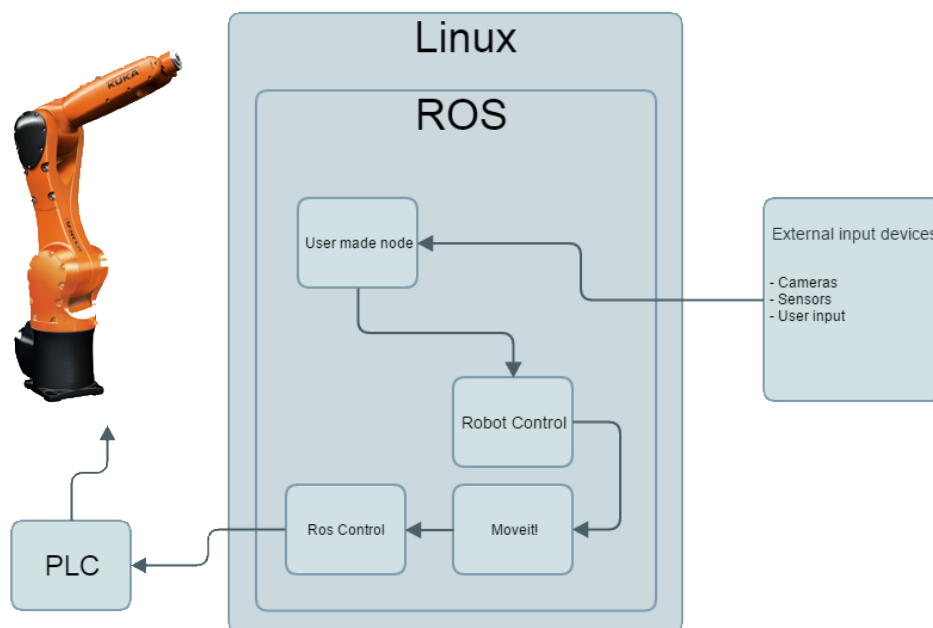


Figure 2.65: A simple illustration of the ROS architecture.

2.7.2 Nodes

An individual program (or executable) is referred to as a *node* in the ROS context. ROS allows users to run many nodes simultaneously, and handles the communication between nodes internally. This means that a complex robot setup can be run by several individual nodes cooperating to achieve a wanted behavior. In order to run a node in ROS, the following command is used:

```
roslaunch [package name] [node name]
```

2.7.3 Services

Services in the ROS context are comparable to program methods or functions. The key feature of the ROS service is that ROS allows for publication of available services for a given node. These services can then be called remotely from other nodes. The communication interaction between nodes is illustrated in Figure 2.66. This allows for simple interaction between different nodes. One important fact is that ROS handles all the cross-application communication, making the process of communication between two applications extremely simple. The data exchange in a service call (request and response) is predefined by the developer. This is done using a file with the *.srv* extension. The content of the file is simple, and is structured as *request* (*function argument*) and *response* (*function return value*). The request and response are separated with a line containing the text "- - -". A simple service might be defined as following.

```
1 | int request;
2 | ---
3 | int response;
```

Paragraph *Publishing a service* and *Calling a service* in section 3.2.7 shows an example of how to publish and call a service.

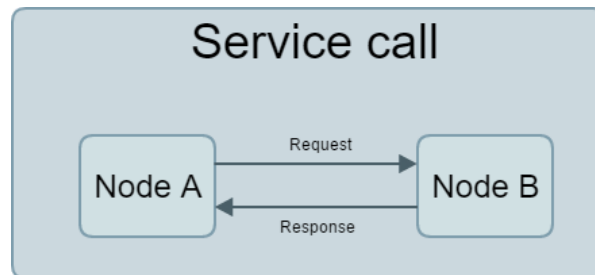


Figure 2.66: Illustrates the interaction between two nodes in a service call.

2.7.4 Topics

Topics in the ROS context are comparable to data streams. Like services, topics are methods used for communication between different applications. ROS allows for both publication and subscription of topics. The mechanics of this convention is that a node can publish data to a topic, which is automatically sent to all nodes subscribed to that particular topic. This mechanism is illustrated in Figure 2.67. Topics often see a different use case than services, and is more suited for communication that is meant to be continuous (like sensor input data, actuator control data, etc.). The data contained in a topic is defined in the message type of that given topic (see section 2.7.5 for more info about messages)

Paragraph *Publishing a topic* and *Subscribing to a topic* in section 3.2.7 shows an example of how to publish and subscribe to a topic.

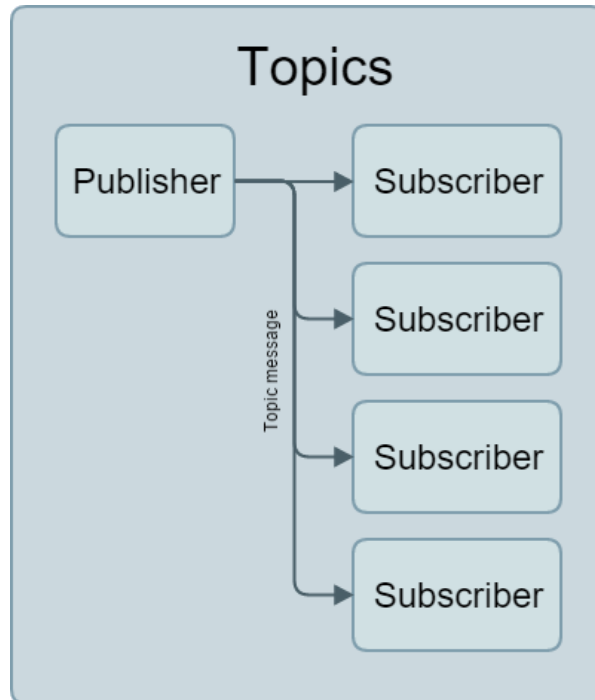


Figure 2.67: Illustrates the interaction between nodes when communication using topics.

2.7.5 Messages

Messages in the ROS context are comparable to *structs* in the C programming language. Like structs, messages are developer defined data types often consisting of multiple variables of different type. This means that a single message can contain many different variables with different types. Like services, messages are defined in the ROS framework by the use of a file. The file extension defining a message is *.msgs*. See paragraph ***Defining a message*** in section 3.2.7 for an example of how to define a message in ROS.

Chapter 3: Method

3.1 Physical setup

3.1.1 Robotic cell setup

The robotic cell used for this assembly task consists of the following hardware:

- Two KUKA KR 6 R900 sixx (KR AGILUS) six axis robotic manipulator.
- Two KUKA KR C4 compact robotic controllers.
- Microsoft Kinect™ One 3D depth sensor.
- Logitech C930e web camera.
- Schunk PSH 22-1 pneumatic linear gripper.
- ROS Master computer.
- Intel NUC NUC5i5RYH mini computer.

Figure 3.1 shows a simulated view of the robotic cell setup. The actual robotic cell is shown in Figure 3.2.

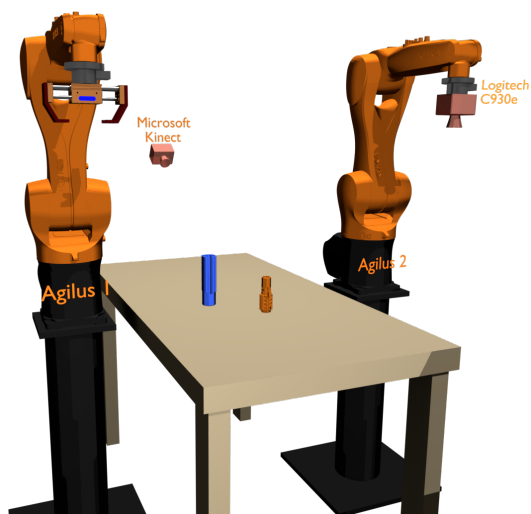


Figure 3.1: Shows a simulated view of the robotic cell.



Figure 3.2: Picture of the robotic cell.

As illustrated by figure 3.1 and 3.2, the Schunk PSH 22-1 pneumatic linear gripper is mounted on the left most robotic manipulator (hereby referred to as *Agilus 1*), and the Logitech web camera is mounted on the right most robotic manipulator (hereby referred to as *Agilus 2*). The

Microsoft Kinect™ depth sensor is located above the table, and behind the two robots. This position was selected in order to produce a 3D point cloud where the whole table is clearly visible without placing the 3D camera far away from the table. This is specifically important when using the Microsoft Kinect™ depth sensor in order to keep the accuracy of the sensor as high as possible because, as shown by [Khoshelham and Elberink \(2012\)](#), the accuracy of the sensor is proportional to the distance between the camera and the object of interest.

Agilus 1 is used to manipulate the parts that is to be assembled using the linear pneumatic gripper and *Agilus 2* is used to position the Logitech web camera (called an *eye-in-hand* setup). The flexible position of the web camera allows for a highly dynamic assembly setup, where the initial position of the parts can be chosen at random.

In order to coordinate information from both cameras together with the two robots, four reference frames have been established with known positions. These reference frames are shown in Figure 3.3. The frames located at the tool of the two robots are fixed to the robots, and move accordingly. The two remaining reference frames are fixed in space. The main origin of the robotic cell is defined by the reference frame located at floor level between the two robots. All movements, and object positions retrieved using object detection are transformed into this reference frame, effectively making it the global origin of the robotic cell.

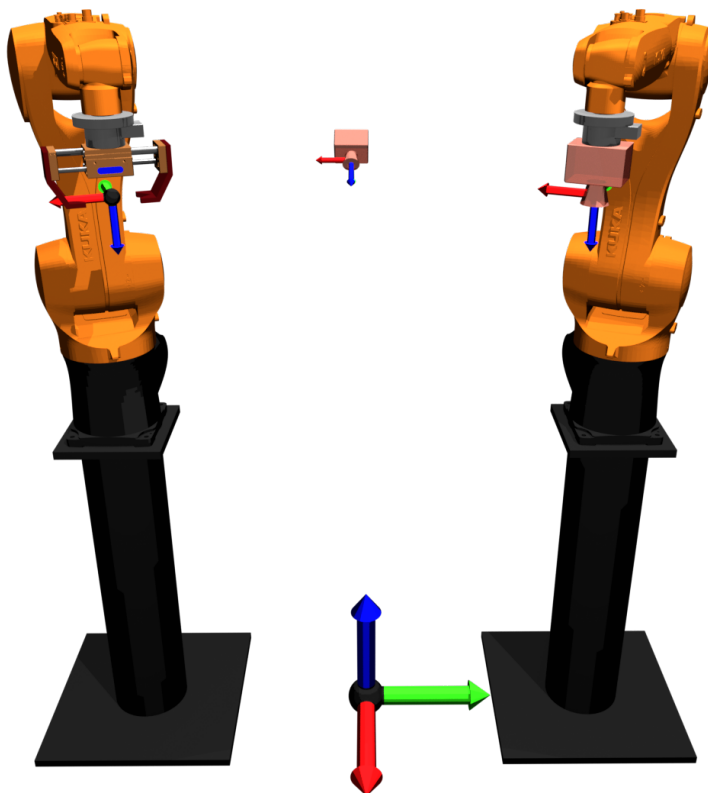


Figure 3.3: Illustrates the global origin of the robotic cell, as well as the tool and optical reference frames for the robots and cameras.

A simulated view of the scene produced by the 3D depth sensor, and the web camera is shown in figure 3.4 and 3.5. Note that the simulated image from the web camera is taken with the

robot positioned above the table, and not in its home position.

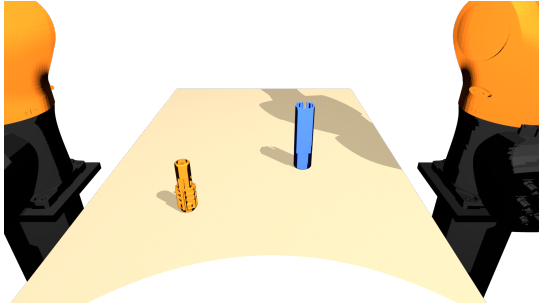


Figure 3.4: A simulated view of the table as seen from the 3D camera.



Figure 3.5: A simulated view of the table as seen from the 2D web camera.

The purpose of the Intel NUC is to serve as a 3D camera data acquisition server. The Kinect depth sensor is connected to the Intel NUC, and the data acquired from the depth sensor is published on a ROS topic in order to access it on the main ROS computer. This was done because of physical limitations, where the positions of the depth sensor and the ROS master computer is too far apart to be able to connect them directly. The data is streamed over the network by TCP/IP using the ROS framework.

3.1.2 Calibrating 3D camera position

In order to obtain usable information from the 3D depth sensor, its position in space needs to be known. In order to calibrate the 3D camera's position in space in relation to the world frame (robotic cell origin) an *augmented reality* tag with a known position is used. Figure 3.6 shows a typical *augmented reality* tag.

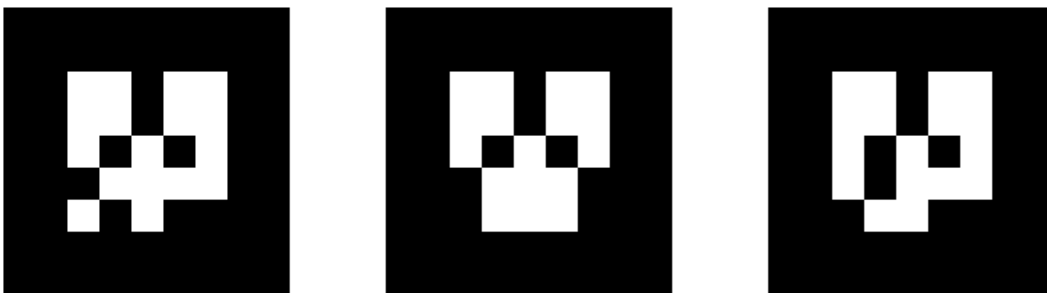


Figure 3.6: Shows three typical augmented reality tags. Image from [Liebhardt \(2016\)](#).

An *augmented reality* tag was placed on the table, directly above the world reference frame

of the robotic cell. By measuring the height of the table z_{table} , a rigid transform between the world reference frame and the *augmented reality* tag is defined as:

$$\mathbf{T}_{ar-tag}^{world} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & z_{table} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.1)$$

The position of the *augmented reality* tag in relation to the 3D camera is found using a publicly available *ROS* node called *ar_track_alvar* (Liebhardt, 2016). The *ar_track_alvar* program runs in *ROS*, taking the 3D depth data as input, and returning the position and orientation of the *ar-tag* in relation to the 3D camera reference frame. This output is in the form of a homogeneous transformation matrix T_t^c .

The final step is to use the information acquired from the *ar-tags* position in the 3D camera reference frame to obtain position of the 3D camera in relation to the world reference frame. This is done using the following equations:

$$T_c^w = T_t^w \times T_c^t, \quad \text{where } T_c^t = (T_t^c)^{-1} \quad (3.2)$$

The annotations used in equation 3.2 relate to the following

w - World reference frame.

t - Augmented reality tag reference frame.

c - 3D camera reference frame

3.1.3 Parts used for assembly

The automated assembly described in this thesis is meant to perform an assembly task of two given parts, *part A* and *part B* as shown in Figure 3.7 and 3.8. The way these parts are assembled is with *part A* placed into *part B* from a specific direction. The positional accuracy needed for a successful assembly is approximately $\pm 1\text{mm}$ in both the x and y axes. The assembly tolerance when it comes to orientation is approximately $\pm 1.5^\circ$.

The two parts used will be denoted, from this point on, as *part A* and *part B* in this thesis.

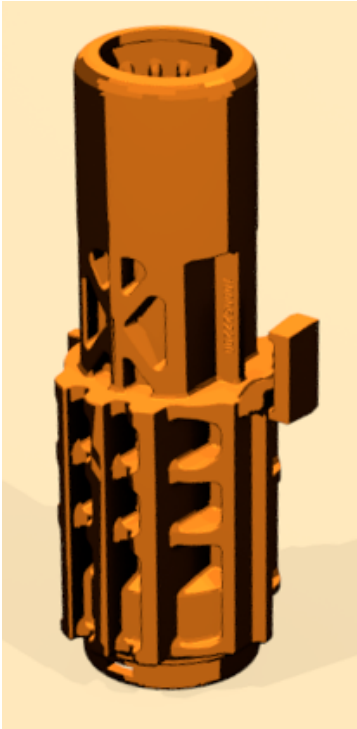


Figure 3.7: A 3D model of *part A*.

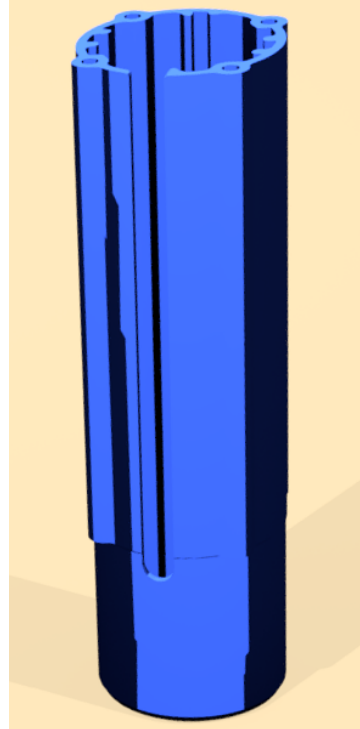


Figure 3.8: A 3D model of *part B*.

3.1.4 Calibrating 2D intrinsic parameters

Calibration of a 2D camera, also referred to as *camera resectioning* (The MathWorks, 2016), is used in robotics for accurate computation of the position of objects in the image. As explained in section 2.1.6, the camera parameter matrix is expressed as:

$$\mathbf{K} = \begin{pmatrix} \frac{f}{\rho_w} & 0 & u_0 \\ 0 & \frac{f}{\rho_h} & v_0 \\ 0 & 0 & 1 \end{pmatrix}$$

and is actually the intrinsic parameters of the camera, defined by the focal length, pixel size and the optical center in pixels. This matrix allows the computation of image coordinates from pixel coordinates. This is needed to express the position of objects detected in the image.

Camera calibration will also allow correction for lens distortion. As modern day cameras use lenses to create brighter images and allow focusing, they also introduce radial distortion of the image. In order to flatten the image, representing the scene as it actually is, camera calibration can be used to estimate the parameters of the lens and image sensor of the camera. Note that the distortion is larger close to the image edges, and very small at the optical center.

Such calibration algorithms are available in numerous image processing toolboxes, e.g. MATLAB and OpenCV. The approach in this thesis is implemented in OpenCV using C++. The following list describes what is needed to perform the calibration:

- A compatible camera (Logitech C930e)
- A chessboard of size e.g. 5×7 printed on paper
- Compiled executable for calibration (available in the digital appendix as described in Appendix D)

The code used allows the input of the camera resolution, chessboard size, path to parameter storage after calibration and some other options like number of calibration images and delay between image capture. The chessboard is then held in front of the camera with different orientations and moved around to cover the whole field of view. Loading a set of already captured images is also an option. The output is a file of type XML or YAML including the following parameters:

- Camera intrinsic parameters
- Camera extrinsic parameters
- Camera distortion coefficients

This file can be loaded into other applications and used for correction of lens distortion and computation of image coordinates.

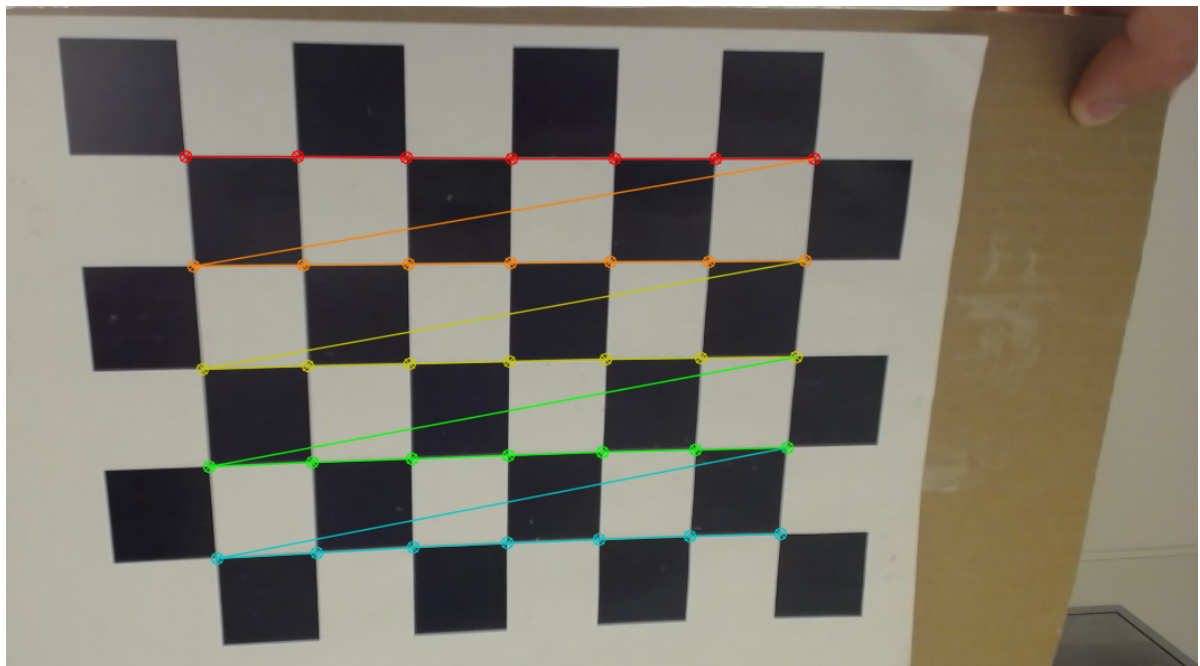


Figure 3.9: Calibration procedure of camera parameters using a chessboard of size 5×7 .

3.1.5 Calibrating eye-in-hand transform

In order to perform correct vision-based robot control, the *eye-in-hand* system had to be calibrated. This is crucial since the chosen approach for object assembly detects the center of an object in the images captured with the *eye-in-hand* system. The robot is then moved so that the optical center of the camera overlays the object center, thus minimizing the error

between a current center point to a desired point. This approach is called *Image-Based Visual Servoing* (IBVS) as described in Corke (2013). However, our approach is simplified in terms of the following assumptions:

- The objects are detected with a measured distance from the camera lens.
- The camera orientation is fixed and always perpendicular to the table the object is placed on.

This means that the distance in the xy -plane between the optical center of the camera and a detected object is the main output from the detection algorithm presented in section 3.2.4. Since the world frame and the camera frame in the robotic cell is orientated as illustrated in Figure 3.10, a rigid transform between them must be established to represent the position of objects detected in the camera frame relative to the world frame.

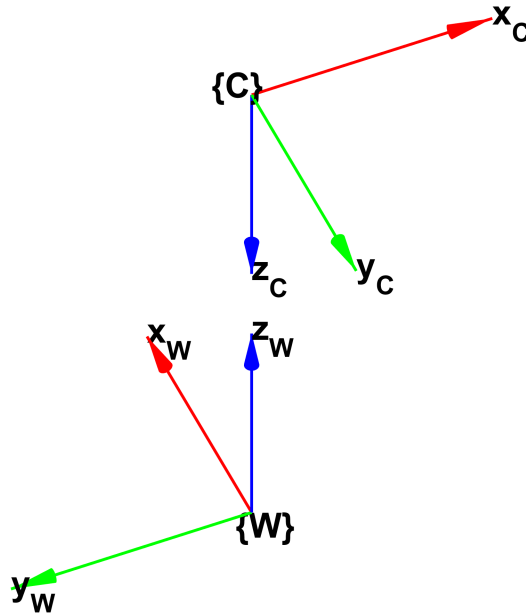


Figure 3.10: Illustration of the orientations of the camera frame \mathcal{C} and world frame \mathcal{W} .

The homogeneous transformation matrix (as described in section 2.1.2) of the camera frame \mathcal{C} relative to the world frame \mathcal{W} is given from

$$\mathbf{T}_C^{\mathcal{W}} = \begin{bmatrix} \mathbf{R}_C^{\mathcal{W}} & \mathbf{t}_C^{\mathcal{W}} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.3)$$

where a typical translation between the frames is $\mathbf{t}_C^{\mathcal{W}} = (x \ y \ z)^T$ and the rotation is given

from: $\mathbf{R}_C^{\mathcal{W}} = \mathbf{R}_y(\pi)\mathbf{R}_z(-\frac{\pi}{2})$, which results in:

$$\mathbf{T}_C^{\mathcal{W}} = \begin{bmatrix} 0 & -1 & 0 & x \\ -1 & 0 & 0 & y \\ 0 & 0 & -1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.4)$$

From the camera calibration method as explained in section 3.1.4, the camera intrinsic parameters K are also known. Lets say the camera parameters are given as:

$$K = \begin{pmatrix} 750 & 0 & 640 \\ 0 & 750 & 360 \\ 0 & 0 & 1 \end{pmatrix} \quad (3.5)$$

For an object detected in the image plane at pixel coordinate $\tilde{\mathbf{p}} = (720 \ 400 \ 1)^T$, the normalized image coordinates are given as explained in section 2.1.6:

$$\tilde{\mathbf{s}} = K^{-1}\tilde{\mathbf{p}} = \begin{pmatrix} \frac{1}{750} & 0 & -640/750 \\ 0 & \frac{1}{750} & -360/750 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 720 \\ 400 \\ 1 \end{pmatrix} = \begin{pmatrix} 0.1067 \\ 0.0533 \\ 1 \end{pmatrix} \quad (3.6)$$

We denote an object frame \mathcal{O} with the same orientation as the world frame, fixed to the detected object center as illustrated in Figure 3.11. If the distance along the camera optical axis to the detected object is $\lambda = 0.2$, the position $\tilde{\mathbf{t}}_{\mathcal{CO}}^{\mathcal{C}}$ of the object in the camera frame is:

$$\tilde{\mathbf{t}}_{\mathcal{CO}}^{\mathcal{C}} = \begin{bmatrix} \mathbf{t}_x^{\mathcal{C}} \\ \mathbf{t}_y^{\mathcal{C}} \\ \mathbf{t}_z^{\mathcal{C}} \\ 1 \end{bmatrix} = \begin{bmatrix} \lambda\tilde{\mathbf{s}} \\ 1 \end{bmatrix} = \begin{bmatrix} 0.0213 \\ 0.0107 \\ 0.2 \\ 1 \end{bmatrix} \quad (3.7)$$

If we now include the homogeneous transformation matrix $\mathbf{T}_C^{\mathcal{W}}$, we will get the position of \mathcal{O} in the coordinates of the camera frame \mathcal{C} relative to the world frame \mathcal{W} expressed as:

$$\tilde{\mathbf{t}}_{\mathcal{CO}}^{\mathcal{W}} = \mathbf{T}_C^{\mathcal{W}}\tilde{\mathbf{t}}_{\mathcal{CO}}^{\mathcal{C}} = \begin{bmatrix} 0 & -1 & 0 & x \\ -1 & 0 & 0 & y \\ 0 & 0 & -1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{t}_x^{\mathcal{C}} \\ \mathbf{t}_y^{\mathcal{C}} \\ \mathbf{t}_z^{\mathcal{C}} \\ 1 \end{bmatrix} = \begin{bmatrix} x - \mathbf{t}_y^{\mathcal{C}} \\ y - \mathbf{t}_x^{\mathcal{C}} \\ z - \mathbf{t}_z^{\mathcal{C}} \\ 1 \end{bmatrix} = \begin{bmatrix} x - 0.0107 \\ y - 0.0213 \\ z - 0.2 \\ 1 \end{bmatrix} \quad (3.8)$$

The variables x , y and z denote the position of \mathcal{C} relative to \mathcal{W} . This is actually the position of the camera lens mounted on the manipulator end-effector. The relative movement from current end-effector pose to the detected object in world coordinates is expressed in equation 3.8. If only the movement in the xy -plane is executed, the camera optical axis will be lined up with the detected object center. The true position of the object in the world xy -plane can then be retrieved by acquiring the manipulator pose from a move group as explained in 3.2.3.

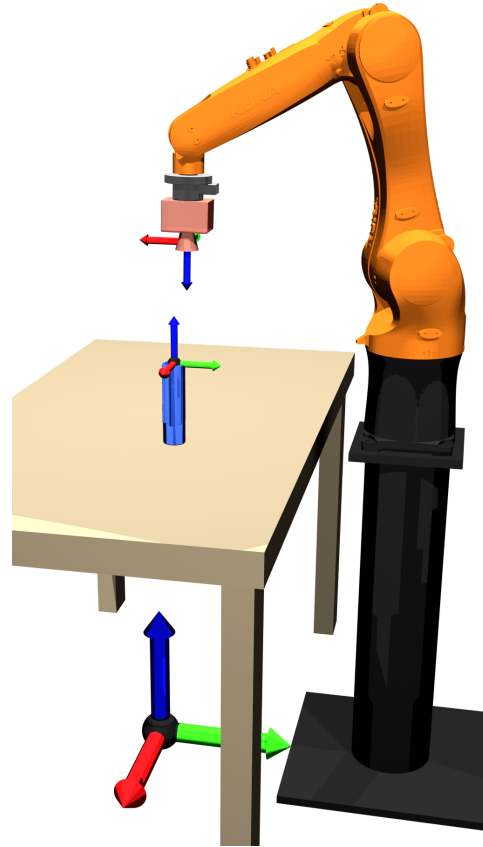


Figure 3.11: Shows the world frame \mathcal{W} (floor), the object frame \mathcal{O} (object center) and the camera frame \mathcal{C} (camera lens).

Furthermore, since the camera used to detect objects is mounted on a bracket at the manipulator end-effector, the optical axis of the camera is most likely not properly lined up with the z -axis of the manipulator end-effector. This is a problem when the second manipulator is going to pick up the object at a detected position. To overcome this offset, the following steps were performed:

1. An object was placed on the table in the robotic cell. The *eye-in-hand* (*Agilus 2*) robot was then moved above the object and the detection algorithm was activated.
2. Based on the above method in equation 3.8, a relative movement of the robot to the center of the object was computed and the robot moved accordingly in the xy -plane.
3. The camera bracket was then replaced with a *calibration tool* (a rod with a fine point) for tool center point. With the new tool the robot was jogged close to the object along the z -axis keeping the same xy -coordinates.
4. By iteratively moving the robot in x - and y -axes, the offset from the camera optical axis to the end-effector z -axis was detected in relation to the world frame.

By adding the offset values to the detected coordinates of the object in the world frame, the robotic manipulator with the gripper (*Agilus 1*) can accurately pick up an object detected

from the *eye-in-hand* system.

3.2 Software development

The software used to process both 2D and 3D images, as well as control the robotic manipulators is written using the C++ programming language. The actual development was done using both the *QT Creator* and the *CLion* integrated development environment (IDE). The following sections describes the different aspects of the software development in detail.

3.2.1 Acquiring 3D point clouds

The 3D point cloud produced by the Microsoft Kinect™ can be acquired and published through a publicly available ROS node named *kinect2_bridge* (Wiedemeyer, 2016). This program utilizes a data stream acquisition program called *libfreenect2* (Xiang et al., 2016) to acquire the data published by the 3D camera. The data stream is then converted to a ROS message. The message used is *sensor_msgs/pointcloud2* (ROS.org, 2016b). This message is made available in ROS through a topic.

3.2.2 Acquiring 2D images

Setting up the video stream for 2D object detection was carried out using OpenCV (Open Source Computer Vision) for C++ in ROS. To enable video capturing, one simply has to instantiate an object of class *cv::VideoCapture*. This is a C++ API enabling video capture from cameras. The class has numerous properties that can be tweaked, for instance the desired resolution of the captured image frames. Once the class has been configured to match the desired video properties a method called *open(int index)* is called. As suggested by the method name, it will open a connection to the camera.

In order to actually acquire an image that can be processed, the object of *cv::VideoCapture* is used to store the image data in an object of class *cv::Mat*. This class represents an n-dimensional dense numerical single-channel or multi-channel array. It can be used to store real or complex-valued vectors and matrices, grayscale or colour images, voxel volumes, vector fields, point clouds, tensors or histograms (OpenCV, 2015b). The acquirement of an 2D image is the first action in every iteration of the image processing ROS node at a given loop rate. The image can then be processed as explained in section 3.2.4 and published as a ROS message on a given topic using *cv_bridge*. *Cv_bridge* is an interface used to encode OpenCV images into ROS image messages (Mihelich and Bowman, 2010). Any ROS node can now subscribe to the topic and visualize the image.

3.2.3 Control the robotic manipulator

ROS is, as described in section 2.7, used for control of the two KR AGILUS manipulators in the robotic cell. Included in ROS is a software package called MoveIt! (Sucan and Chitta, 2016). This package has an inverse kinematic solver making it possible to control the manipulators based on input of e.g. the end-effector pose. The robotic cell is already configured for use with MoveIt! and typically, the MoveIt! Rviz (a visualization framework available in ROS) plugin, a graphical user interface for manipulator control, is used to move the manipulators by drag-and-drop of the end-effector. However, the MoveIt! software can also be used directly in

any ROS node through a C++ API called *move_group_interface* (Sucan and Chitta, 2013). It allows trajectory planning and current pose acquirement of *move groups* and is a powerful tool when the goal is to control a manipulator from other ROS nodes.

In MoveIt! a *move group* consists of a given number of connected joints. Each group has a defined name. In the case of this project they are called *agilus1* and *agilus2*. One approach to manipulator control is to create two objects of class *moveit::planning_interface::MoveGroup*, one for each group, and interface directly with these groups in a control node. However, service controlled manipulator movements were considered a "nice to have" functionality and chosen as the desired approach. Therefore, the use of the *move_group_interface* was programmed in a stand-alone ROS node. This node advertise two different services called *plan_pose* and *go_to_pose*.

Through these services, any ROS node can plan a trajectory or move the manipulators by calling the appropriate service. By specifying the goal pose of the end-effector a linearly interpolated trajectory from start pose to goal pose is generated.

The following is a code example showing the use of the *go_to_pose* services. This service is called using the *Pose.srv* service object (the *Pose.srv* service object definition is available in Appendix C).

```

1 // The service client and service object is created.
2 goToClient = n.serviceClient<agilus_planner::Pose>("/robot_service_ag1/
   go_to_pose");
3 agilus_planner::Pose pose_service;
4
5 // The content of the service object is populated with the home position of
   Agilus 1.
6 pose_service.request.header.frame_id = "/world";
7 pose_service.request.relative = false;
8 pose_service.request.set_position = true;
9 pose_service.request.position_x = 0.445;
10 pose_service.request.position_y = -0.6025;
11 pose_service.request.position_z = 1.66;
12 pose_service.request.set_orientation = true;
13 pose_service.request.orientation_r = 0.0;
14 pose_service.request.orientation_p = 3.1415;
15 pose_service.request.orientation_y = 0.0;
16
17 // The service client is called with the populated service object.
18 goToClient.call(pose_service);

```

3.2.4 2D object detection

As already mentioned in section 3.2.2, OpenCV for C++ is used to acquire images. Before these images add any value to the vision solution, they must be processed in terms of feature detection. OpenCV is also used for this task as it implements very useful functionality for 2D object detection in a way that makes abstraction of code possible. The functionality of interest in OpenCV is mainly the detection, description and matching of image features. This can be done in numerous ways. However, based on theory about this type of object detection, the following four algorithms are the most promising in order to solve the object detection problem:

SIFT The keypoint detector and descriptor extractor called Scale-Invariant Feature Transform as presented in section 2.6.1 is implemented and available in OpenCV as a class named `cv::xfeatures2d::SIFT`.

SURF Speeded-Up Robust Features as presented in 2.6.2 is also available in OpenCV. The implemented class is named `cv::xfeatures2d::SURF`.

BRISK Binary Robust Invariant Scalable Keypoints as presented in 2.6.3 is a third alternative for object detection. It is implemented in OpenCV named `cv::BRISK`.

ORB Oriented FAST and Rotated BRIEF is similar to BRISK as explained in 2.6.4. The algorithm is implemented in OpenCV as `cv::ORB`.

All of the above algorithms are invariant to scale and rotation. The usage of each one is similar. An object detection procedure consists of the following steps:

1. Load a query image of the object to be detected into an instantiated object of `cv::Mat`.
2. In the query image:
 - a. Detect keypoints and store them in an object of `std::vector<cv::KeyPoint>`.
 - b. Extract descriptors and store them in an object of `cv::Mat`.
3. Acquire a training image as described in 3.2.2.
4. For each training image obtained:
 - a. Detect keypoints and store them in an object of `std::vector<cv::KeyPoint>`.
 - b. Extract descriptors and store them in an object of `cv::Mat`.
 - c. Match the query descriptors with the training descriptors using either brute-force or FLANN. Store the matches in a vector `std::vector<cv::DMatch>` and sort them using the distance ratio-test as explained in section 2.6.5.
 - d. Compute the homography between the matched points in the query and training image using RANSAC as described in 2.6.6. Transform the query object plane using the homography and surround the detected object with four corner points in the training image, supposedly as a rectangular box when lines are drawn between them.
 - e. Check that the inner angles of this box is close to 90° compared to an allowed deviation.
 - f. If the box is not rectangular:
 - i. Cancel further processing and proceed with the next image.
 - g. If the homography transform is accepted:
 - i. Mark the object in the training image and publish the processed image using `cv_bridge` as described in 3.2.2.
 - ii. Compute the image coordinates of the detected object center using the pixel coordinates of the intersection between the object box diagonals as input.
 - iii. Compute the orientation of the detected object.

- iv. Publish the image coordinates and orientation data as a ROS message of type `geometry_msgs/Pose2D.msg`.

Testing and evaluation of the available keypoint detectors and descriptor extractors is possible by implementing this procedure in a C++ ROS node. Test procedures are presented in section 3.3.

Acquiring object orientation

As previously mentioned, if the homography between the query image and the match in the training image results in a rectangular box surrounding the object, the match is good for further processing. In order to successfully assemble *part A* into *part B*, the orientation acquirement of both parts needs to be accurate. Section 3.1.5 points out that the orientation of the *eye-in-hand* system is always fixed and perpendicular to the table where the objects are placed, thus always perpendicular to the object seen from above.

With a known and fixed orientation of the camera frame, it is possible to accurately compute the orientation of a given shape in the image using basic geometry. This shape is always a simple square or rectangle because of the matching algorithm expressed above. An object is detected at pixel coordinates $\mathbf{p} = (u, v)$ in the image plane as illustrated in Figure 3.12. The pixel coordinates of the corners 0, 1, 2 and 3 is denoted $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2$ and \mathbf{p}_3 . The image center is denoted $\mathbf{p}_c = (u_0, v_0)$. Given a rotation of the object box, there will be a right triangle with hypotenuse between corner 0 and 1 as shown in the figure below.

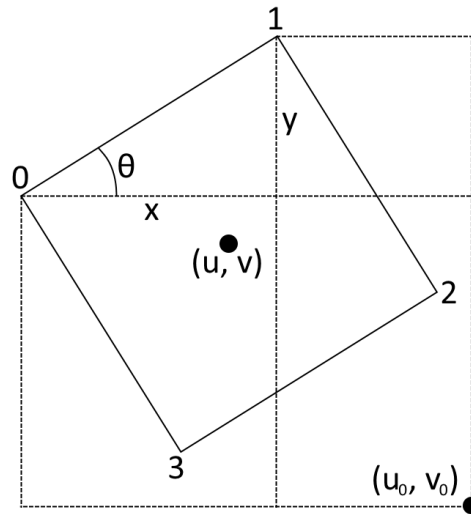


Figure 3.12: Illustrates the simple computation of the objects orientation.

The angle of the object is then simply expressed as:

$$\theta = \text{atan2}\left(\frac{y}{x}\right) \quad (3.9)$$

where the horizontal pixel length x and vertical pixel length y of the triangle is:

$$\begin{aligned}x &= \mathbf{p}_1(u, 0) - \mathbf{p}_0(u, 0) \\y &= \mathbf{p}_0(0, v) - \mathbf{p}_1(0, v)\end{aligned}$$

A testing procedure for the stability of this method is presented in section 3.3.5.

3.2.5 3D object detection

The 3D object detection used in this assembly task is implemented using the C++ programming language in conjunction with the *Point Cloud Library* (PCL) (PCL, 2016). PCL implements functionality useful to perform 3D object detection. The following is an explanation of the steps performed in sequence in order to detect a wanted object using the 3D camera, and a description of the PCL classes used to perform them. This is an implementation of a global pipeline.

Passthrough filtering The first step in the object detection process is *passthrough filtering*. This is done to reduce the number of data points in the point cloud, but also to remove any unwanted parts of the 3D scene. Passthrough filtering is explained in detail in section 2.3.1. The PCL class `pcl::PassThrough` is used to perform a passthrough filtering task.

Voxel grid filtering The *voxel grid filtering* is performed to ensure that the point cloud is uniformly sampled. This is important in order to estimate accurate descriptors of the scene that is comparable to the training set (the point clouds in the training set are all sampled uniformly). Voxel grid filtering is explained in detail in section 2.3.2. The PCL class `pcl::VoxelGrid` is used to perform a voxel grid filtering task.

Plane model segmentation Given that it is known that the parts that is to be assembled will be located on a table surface, a model segmentation is performed. This is done in order to remove the part of the point cloud that corresponds with the table surface. Model segmentation is explained in detail in section 2.3.5. The model segmentation is done using the PCL class `pcl::SACSegmentation`. In order to segment a plane, the RANSAC model `pcl::SACMODEL_PLANE` is used.

Cluster extraction At this point in the process, the only points left in the point cloud will correspond with the parts that is to be assembled and some random, scattered noise. The purpose of the cluster extraction step is to separate the different objects in the scene into their own individual point clouds. This step is critical in order to use global descriptors. This is explained in detail in section 2.3.6. The cluster extraction is performed by the PCL class `pcl::EuclideanClusterExtraction`. All steps following the *cluster extraction* is performed for all clusters extracted in this step.

Normal estimation Surface normals are estimated. The surface normals are instrumental to the estimation of both local and global descriptors. The method used for estimating surface normals is described in section 2.4.1. Normal estimation is performed using the PCL class `pcl::NormalEstimation`.

Keypoint selection Keypoints used for local descriptor estimation is selected using SIFT3D. As described in section 2.4.2, keypoints are selected to be points of interest that contain

more information than its neighbouring points. Keypoints is explained in section 2.4.2. The selection is done using the PCL class *pcl::SIFTKeypoint*.

Local descriptor estimation Local descriptors are estimated using the *FPFH* descriptor. The local descriptors will be used to calculate an initial alignment at a later step. Local descriptors is explained in section 2.4.3. The estimation is performed using the PCL class *pcl::FPFHEstimation*.

Global descriptor estimation Global descriptors are estimated for all clusters extracted in the cluster extraction step. As described in section 2.4.4, the global descriptor holds information that describes a cluster of points. This descriptor is used for viewpoint matching. The global descriptor estimation is done using the VFH (see section 2.4.4) global descriptor which is implemented in the PCL class *pcl::VFHEstimation*.

Viewpoint matching In order to select the model from the training set with the correct viewpoint of the part (that matches the viewpoint of the part captured by the 3D camera), the global descriptors of all viewpoints in the training set is compared to the object cluster using a nearest neighbour search. The best match is selected as the model that is used for alignment. The nearest neighbour search is applied on a Kd-tree data structure which is generated using the PCL class *pcl::KdTreeFLANN*. This class also implements nearest neighbour search.

Initial alignment Initial alignment is done using the model found to be the best match (previous step). The output of this step is a rigid transform that is close to registering the model from the training set to the object cluster in the scene. The approach used for initial alignment estimation is the sample consensus approach. A brief explanation is available in section 2.5.3. This is implemented in the PCL class *pcl::SampleConsensusInitialAlignment*.

Final alignment The final alignment is estimated using ICP (see section 2.5.2) with the rigid transform estimated in the previous step as the starting point. The rigid transform produced by the ICP algorithm is used in conjunction with the training set information regarding the pose of the model as an estimate of the pose for the wanted object in relation to the 3D camera reference frame T_o^c . This pose is transformed to the world reference frame T_o^w using the rigid transform from the world reference frame to the 3D camera T_c^w as follows:

$$T_o^w = T_c^w \times T_o^c$$

The position of the part found using 3D object detection is the final output of this sequence. The ICP algorithm is implemented in the PCL class *pcl::IterativeClosestPoint*.

3.2.6 Creating training sets

The training set used for 3D object detection was created using the virtual approach described in section 2.4.5. A virtual tessellated sphere is used to position a virtual 3D depth sensor. The sphere used when creating the training set produces 42 individual 3D point clouds, rendered with a resolution of 200×200 pixels. Using this resolution, the rendered scene produced is comparable to the 3D point cloud of an object captured by a 3D camera.

In order to reduce the computation time for 3D object detection, a full set of features (key-points, surface normals, local descriptors and global descriptor) are calculated for each of the 42 point clouds produced for a model. The point cloud, with its corresponding features are saved to file on the computer, allowing for fast processing times during the 3D object detection process.

3.2.7 ROS communication

This section demonstrates how the main communication framework available through ROS are implemented and used.

Publishing a service

As mentioned in section 2.7.3, a service is defined in a separate file with the *.srv* extension. In addition, the node publishing the service needs a callback method for the particular service. The following code defines a callback for the service named *test_service*:

```

1 |     bool test(package_name::test_service::Request &req,
2 |               package_name::test_service::Response &res)
3 |     {
4 |         // This service adds value a and b from the service request together and
5 |         // returns the sum in the service response. This action is performed
6 |         // whenever this service is called.
7 |         res.sum = req.a + req.b;
8 |         return true;
9 |     }

```

Once the service callback is defined, the node can publish the service. This is done in the following way.

```

1 |     ros::ServiceServer service = node_handler.advertiseService("test_service",
2 |                                                               test);

```

The *ServiceServer* is defined with the service name and callback method. At this point, the service is published and can be called from other nodes on the ROS system.

Calling a service

In order to call a service from a remote node, we first define a service client.

```

1 |     ros::ServiceClient client = node_handler.serviceClient<package_name::
2 |     test_service>("test_service");

```

Next, a service object is made, and the request is filled with data.

```

1 |     package_name::test_service srv;
2 |     srv.request.a = 1;
3 |     srv.request.b = 2;

```

Finally, the service is called using the service client object.

```

1 |     client.call(srv);

```


Publishing a topic

Publishing a topic is done through the *Node Handler*. Each ROS node has a node handler, which is used for controlling the node, and communicate with other nodes. In order to publish a topic, a *Publisher* object is retrieved from the *Node Handler*.

```
1 | ros::Publisher topic_publisher = node_handler.advertise<std_msgs::String>("
  | topic_name", 1000);
```

The publisher is defined with a topic name and message type. Next, the message is generated and filled with some data.

```
1 | std_msgs::String topic_message;
2 | std::stringstream message_data;
3 | message_data << "hello world ";
4 | topic_message.data = message_data.str();
```

Finally, the topic is published.

```
1 | topic_publisher.publish(topic_message);
```

Subscribing to a topic

Subscribing to a topic is handled similarly to publishing. First, a subscriber object is obtained through the node handler.

```
1 | ros::Subscriber subscriber = node_handler.subscribe("topic_name", 1000,
  | topicCallback);
```

The subscriber object is defined with a topic name and a callback function that is called when a new topic message is received. A callback method could look like the following:

```
1 | void topicCallback(const std_msgs::String::ConstPtr& topic_message)
2 | {
3 |     // Perform this action whenever the topic is updated.
4 |     ROS_INFO("I heard: [%s]", topic_message->data.c_str());
5 | }
```

The node is now configured to subscribe to the topic *topic_name*.

Defining a message

The following is an example of a *struct* declaration:

```
1 | struct human{
2 |     int age;
3 |     double height;
4 |     double weight;
5 | };
```

Now, this custom datatype can be defined as a message in the ROS framework using a *.msg* file containing the following lines of code:

```
1 | int age;
2 | double height;
3 | double weight;
```

As evident by the code examples above, the declaration of a struct and a message is very similar. The use of messages in topics allows applications to wrap a high amount of data in a single message. One important aspect of messages, services and topics is that they are globally defined, meaning that any node running in the ROS framework can use these user defined types.

3.3 Testing setup

3.3.1 3D object detection accuracy

In order to test the positional accuracy of the 3D object detection output, a test was performed. In this test, the two different parts that is to be assembled was positioned on the table at known locations. The resulting position from the 3D object detection is compared to the known position of the parts. Figure 3.13 shows the setup used, where a part is positioned in different known positions on the table.

The grid used is measured to have 5cm by 5cm squares, and positioned so that the lower left corner of the paper is located directly above the world origin reference frame. This allows for easy positioning of the parts at 5cm increments in negative x direction, and positive y direction. It is fair to assume that the positional accuracy of the camera is close to constant for the entire field of view, thus only one section of the table was used to carry out this test.

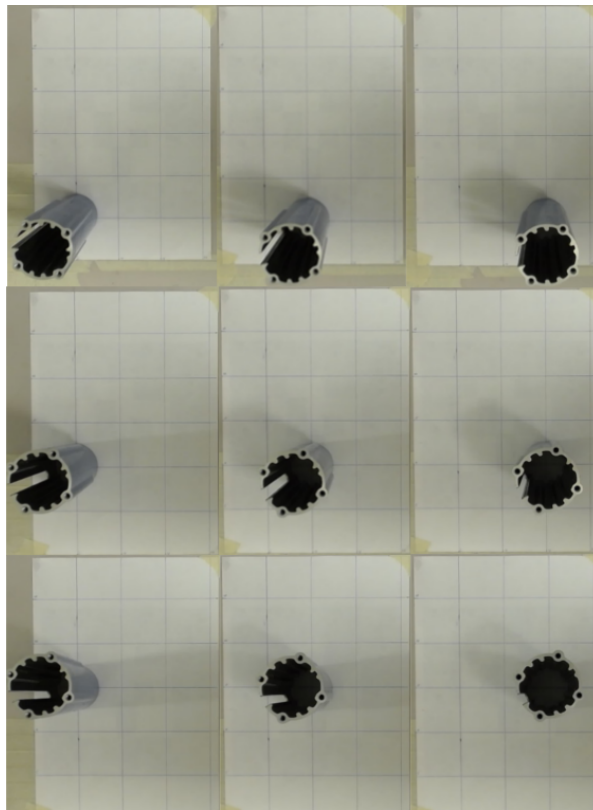


Figure 3.13: Shows the test setup used to measure the positional accuracy of the 3D object detection process.

The expected result from this test is based on a couple of different factors:

1. The accuracy of the Microsoft Kinect™ is known to decrease quadratically with distance as shown by [Khoshelham and Elberink \(2012\)](#). Also shown in this paper is an expected positional accuracy of $< 2\text{mm}$ for the particular working area used.
2. The accuracy of the camera position calibration. Given that the camera position calibration is performed using the 3D camera data, the accuracy of the sensor data will also affect the calibration. The *ar-tag* used was placed approximately 1m away from the sensor. This is within the area where the expected accuracy of the sensor is $< 2\text{mm}$ as shown by [Khoshelham and Elberink \(2012\)](#).

Given these two factors, the theoretical accuracy should be within 0.4cm, however we do not expect to achieve such high accuracy. Adding for some margin of error, we expect the positional accuracy for the 3D object detection process to be within 0.6cm.

3.3.2 Testing global descriptors

In an effort to investigate which global descriptors were most suited to this assembly task, a test was performed. This test was done by taking a 3D depth picture of one of the parts that is to be assembled. This image was processed as described in section 3.2.5. From this point cloud, a VFH descriptor and a CVFH descriptor was estimated.

Two different training sets were created to perform this test. One using the VFH global descriptor, and the other using the CVFH global descriptor. We choose to test these two descriptors based on the previous work by [Alexandre \(2012\)](#). This comparative analysis shows that the more complicated descriptors (such as CVFH and OUR-CVFH) have a higher recognition rate than the more basic VFH descriptor. Because of this, we want to use a complex global descriptor if this is possible. Figure 3.14 shows the 3D scene used for testing the different global descriptors.

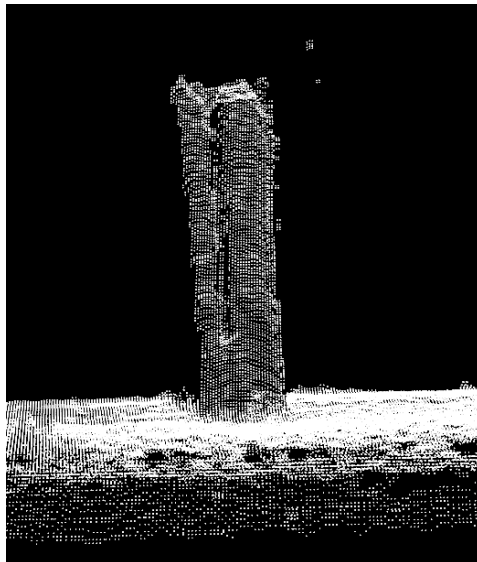


Figure 3.14: The scene used to test match the different global descriptors.

The reason behind only testing the VFH and CVFH descriptor is that the OUR-CVFH descriptor is based on the CVFH descriptor. This means that if the CVFH descriptor is not usable, this will also apply to the OUR-CVFH descriptor.

The two different global descriptors were tested by performing descriptor matching. The expected result is to see a better confidence value (a value that describes how good a match is) for the more complex CVFH descriptor than the basic VFH descriptor.

3.3.3 2D object detection processing time

A test was performed in order to get knowledge of the difference in processing time between SIFT, SURF, BRISK and ORB used in the developed object detection ROS node. The processing time needed to detect keypoints, compute descriptors and match descriptors was of interest individually and as a total. The camera was stationary at the table during all tests pointing towards the same image taped to the wall as illustrated in Figure 3.15. OpenCV has methods for timing implemented as `cv::getTickCount()` and `cv::getTickFrequency()`. The following code is used to compute the detection time of keypoints in seconds:

```
1 | double d = (double)cv::getTickCount();
2 | detector->detect(video, keypoints_scene);
3 | d = ((double)cv::getTickCount() - d)/cv::getTickFrequency();
```

The method is similar for the descriptor extraction time and matching time.

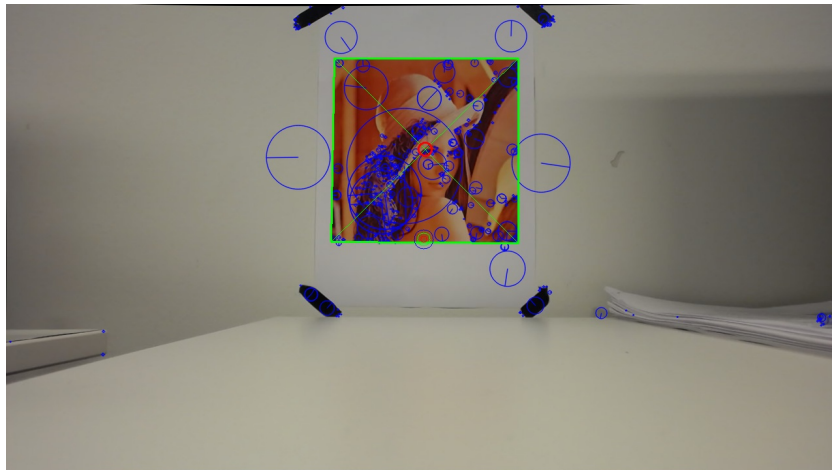


Figure 3.15: The view from the camera during testing of SIFT.

It is expected to see a clear difference in total processing time between the binary descriptor methods, i.e. BRISK and ORB, and the real-valued methods, i.e. SIFT and SURF.

3.3.4 2D object detection matching stability

Matching stability is important to ensure high rate of success when assembling *part A* and *part B*. It is also desirable that this works under the conditions present in the robotic cell used for assembly. In order to test the stability of the object detection procedure described in section 3.2.4 a simple test was performed in the robotic cell. It consisted of the following steps:

1. Place the object to be tested at the table with a given orientation.
2. Move the *eye-in-hand* (*Agilus 2*) manipulator above the object at a chosen z -coordinate in the world frame.
3. Start object detection with a chosen algorithm, like SIFT, against a chosen query image of the object.
4. Move the manipulator closer to the object along the z -axis until matching fails.

The steps were repeated for both parts using all four algorithms. From this test we want to map the range of different distances between object and camera lens where the matching is stable enough to be used for the final assembly task.

3.3.5 2D object detection orientation stability

A very important part of the 2D pose of an detected object is the orientation. In order to determine the stability of the orientation computation as described in section 3.2.4 using the different algorithms of interest, a test was performed.

The steps performed to acquire test data is:

1. Place the object to be tested at the table with a given orientation.
2. Move the *eye-in-hand* (*Agilus 2*) manipulator above the object at a given z -coordinate in the world frame. The specific coordinate has been determined from the matching stability results as presented in 4.3.2.
3. Compute the mean of n acquired orientations until m data points is generated.

This test was repeated for each algorithm that showed to have sufficient matching stability (tested as described in section 3.3.4), for both parts that is to be assembled. From each detection cycle of the algorithm presented in 3.2.4 an orientation is published as an angle in the range $[-180, 180]$ degrees. The test data is more specifically formatted as:

- One data point in the series of test data is obtained by computing the mean of
 - $n = 10$ acquired orientations when using SIFT
 - $n = 20$ for the rest of the algorithms
- Each test consists of $m = 25$ of these data points.

This is chosen based on the results from the tests concerning processing time as presented in section 4.3.1. From this orientation test we want to determine which approach is better for stable orientation computation.

Chapter 4: Result

4.1 Physical setup

4.1.1 Robotic cell networking

The physical components of the robotic cell is networked together in a way that allows the ROS master computer to communicate with both the two robotic controllers and the Intel NUC computer. A simple sketch of the network setup used for the robotic cell is shown in Figure 4.1.

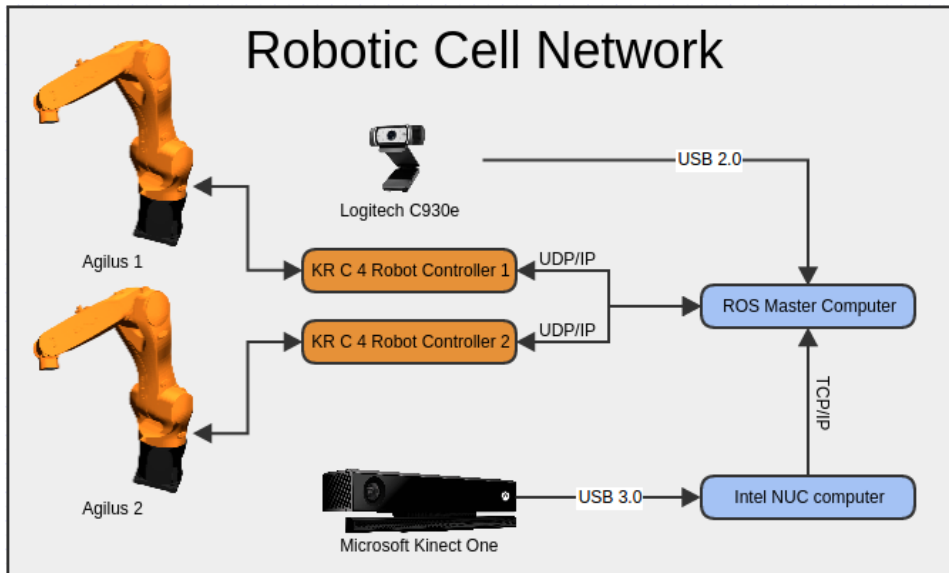


Figure 4.1: Shows the network connections between the critical hardware in the robotic cell.

4.1.2 Calibrating 3D camera position

Since the objective of this task is to combine the data acquired from a 3D depth sensor and a traditional camera. The position of the 3D camera was calibrated using the approach described in section 3.1.2. Figure 4.2 shows the detected *ar-tag* with the corresponding reference frame. This reference frame is used to estimate the position of the 3D camera.

The calibration process resulted in the following homogeneous transformation matrix:

$$T_{tag}^c = \begin{bmatrix} -0.0007 & -0.9998 & -0.0182 & -0.1160 \\ -0.5692 & 0.0154 & -0.8220 & -0.0416 \\ 0.8221 & 0.0097 & -0.5691 & 1.0405 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

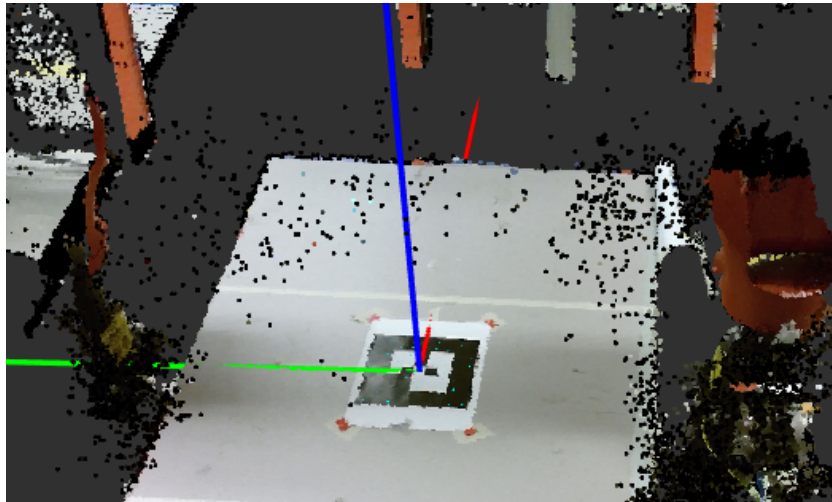


Figure 4.2: Shows the output of the *ar_track_alvar* application used for position calibration of the 3D camera.

This transformation matrix is used as shown in equation 3.2 to produce the following rigid transformation from the world reference frame to the camera origin:

$$T_c^w = T_{tag}^w \times (T_{tag}^c)^{-1}$$

$$T_c^w = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0.87 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} -0.0007 & -0.9998 & -0.0182 & -0.1160 \\ -0.5692 & 0.0154 & -0.8220 & -0.0416 \\ 0.8221 & 0.0097 & -0.5691 & 1.0405 \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1}$$

$$T_c^w = \begin{bmatrix} -0.0008 & -0.5692 & 0.8222 & -0.8793 \\ -0.9998 & 0.0155 & 0.0098 & -0.1256 \\ -0.0183 & -0.8220 & -0.5692 & 1.4258 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Using the calibration approach described in section 3.1.2, the resulting camera position is accurate to the point where the part is fully visible in the 2D camera when the robot manipulator is positioned above the origin of the part. This means that any inaccuracies caused by the calibration of the 3D camera position, and 3D object detection is negligible for the assembly process from the point where the 2D object detection starts.

4.1.3 Calibrating eye-in-hand transform

In order to move the manipulators based on detected objects in the camera frame a calibration step had to be performed. First, the camera intrinsic matrix and distortion coefficients were determined using the method described in section 3.1.4:

$$\mathbf{K} = \begin{pmatrix} 781.585 & 0 & 640 \\ 0 & 781.585 & 360 \\ 0 & 0 & 1 \end{pmatrix} \quad (4.1)$$

$$\begin{aligned} \text{distortion}_{coefficients} &= (k_1, k_2, p_1, p_2, k_3) \\ &= (0.088995, -0.21592, 0.0021548, -0.0039320, 0.095365) \end{aligned} \quad (4.2)$$

Applying correction for the lens distortion of the image stream from the Logitech C930e web camera using the distortion coefficients expressed in equation 4.2 yields a clear improvement as shown in Figure 4.3.

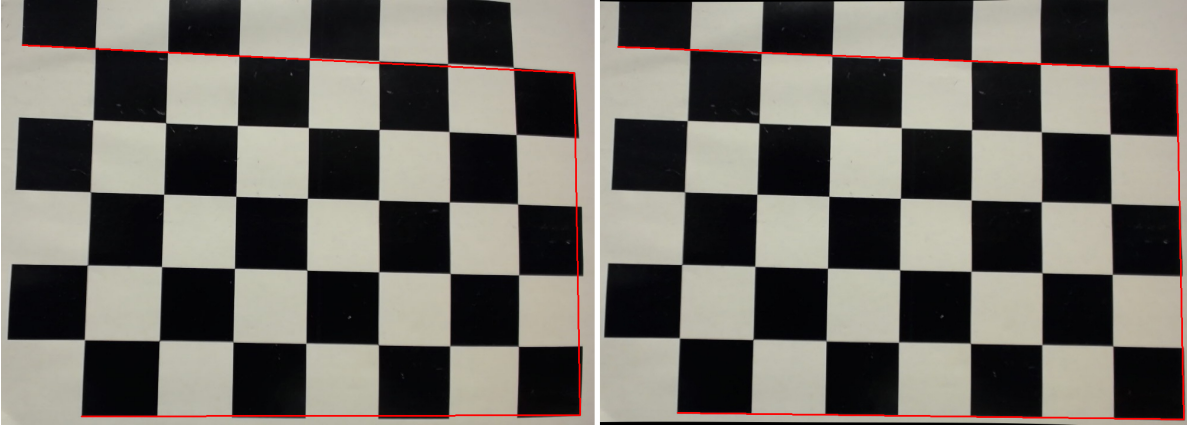


Figure 4.3: Left: Image with lens distortion. Notice the curvature of the image along edges compared to the red lines. Right: The same image, but with correction for lens distortion. The curvature is minimized.

By using the method presented in section 3.1.5, image coordinates of detected objects are computed and used to acquire a relative movement from the current manipulator pose to the detected object in the xy -plane of \mathcal{W} .

As previously stated in section 3.1.5, the camera frame is always fixed and perpendicular to the table surface. Therefore, the rotation between the world frame \mathcal{W} and the camera frame \mathcal{C} is constant. This means that the x -axis of \mathcal{C} always corresponds to the negative y -axis of \mathcal{W} , and the y -axis of \mathcal{C} always corresponds to the negative x -axis of \mathcal{W} . Any image coordinate computed as long as the orientation of \mathcal{C} is fixed can be simplified to the following relative manipulator movement in \mathcal{W} :

$$\begin{aligned} x_{\mathcal{C}} &= -y_{\mathcal{W}} \\ y_{\mathcal{C}} &= -x_{\mathcal{W}} \end{aligned} \quad (4.3)$$

This simplification is implemented in the final code (see digital appendix for the full source code of the `agilus_master_project` application). As illustrated in Figure 4.4, this approach is shown to be a viable solution. In this case the distance between the camera lens and object is approximately $\lambda = 26.7\text{cm}$. In the left image of Figure 4.4, the object is detected at pixel coordinates $\tilde{\mathbf{p}} = (480 \ 48 \ 1)^T$ which in image coordinates scaled with $\lambda = 26.7\text{cm}$ is -5.47cm along the x -axis and -10.66cm along the y -axis in the camera frame \mathcal{C} . In the right image of Figure 4.4, the manipulator has moved according to the relationship between the image coordinates as expressed in equation 3.8 and simplified in equation 4.3, which in this case means 10.66cm along the x -axis and 5.47cm along the y -axis in the world frame \mathcal{W} .

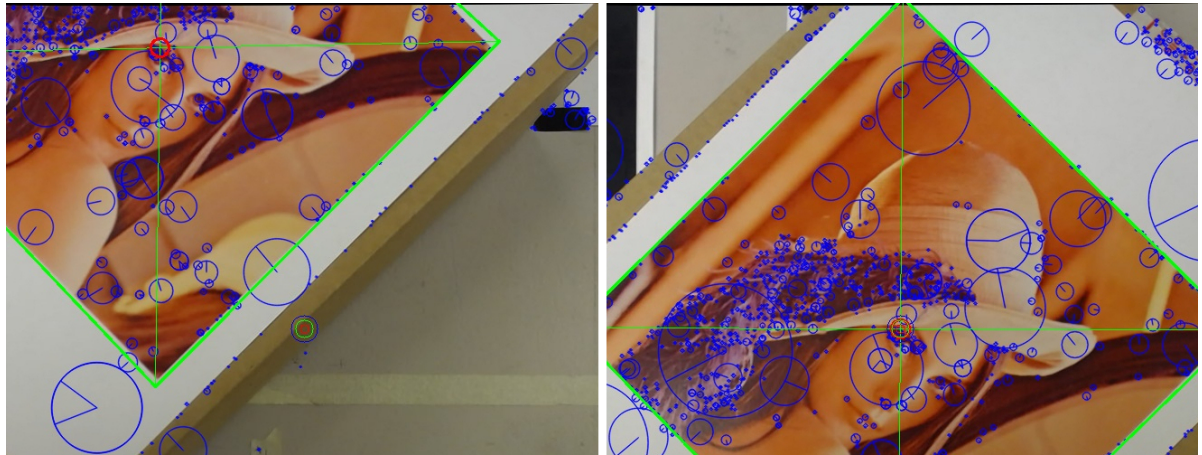


Figure 4.4: Illustrates the *eye-in-hand* robot control based on image coordinates. The optical center of the image is marked with three concentric circles in red, green and blue. The object center is marked with a thick red circle at the intersection of the diagonals.

With the *eye-in-hand* manipulator (*Agilus 2*) positioned above the object as shown to the right in Figure 4.4, an accurate end-effector pose in world coordinates is acquired as described in section 3.2.3.

4.2 3D Computer Vision

4.2.1 Accuracy test of 3D object detection

Using the test method described in section 3.3.1, the following data was obtained:

Testing Object Detection Accuracy Part A (measured in cm)					
Actual X	Actual Y	Measured X	Measured Y	$ \Delta X $	$ \Delta Y $
-5	5	-6.18	5.4	1.18	0.4
-5	10	-6.25	10.55	1.25	0.55
-5	15	-6.28	16.11	1.28	1.11
-5	20	-5.17	21.56	1.28	1.56
-10	5	-11.12	5.08	1.12	0.08
-10	10	-10.83	10.43	0.83	0.43
-10	15	-10.98	15.92	0.98	0.92
-10	20	-11.46	20.85	1.46	0.85
-15	5	-15.89	5.2	0.89	0.2
-15	10	-15.81	10.56	0.81	0.56
-15	15	-15.97	15.77	0.97	0.77
-15	20	-16.18	21.01	1.18	1.01
-20	5	-20.43	5.4	0.43	0.4
-20	10	-20.68	10.72	0.68	0.72
-20	15	-20.72	16.27	0.72	1.27
-20	20	-21.18	21.38	1.18	1.38

The maximum and minimum positional deviations for the testing using *Part A* is shown in table 4.1.

Min/Max Recorded Values	
Max ΔX [cm]	1.46
Max ΔY [cm]	1.56
Min ΔX [cm]	0.43
Min ΔY [cm]	0.08

Table 4.1: Shows the minimum and maximum deviation for the testing using *Part A*.

Testing Object Detection Accuracy Part B (measured in cm)					
Actual X	Actual Y	Measured X	Measured Y	$ \Delta X $	$ \Delta Y $
-5	5	-5.76	5.16	0.76	0.16
-5	10	-6.12	10.8	1.12	0.8
-5	15	-5.98	15.94	0.98	0.94
-5	20	-6.17	20.88	1.17	0.88
-10	5	-10.65	5.47	0.65	0.47
-10	10	-10.62	10.21	0.62	0.21
-10	15	-10.73	15.81	0.73	0.81
-10	20	-10.91	20.79	0.91	0.79
-15	5	-15.22	5.46	0.22	0.46
-15	10	-15.46	10.62	0.46	0.62
-15	15	-15.71	16.2	0.71	1.2
-15	20	-15.85	21.14	0.85	1.14
-20	5	-20.1	5.43	0.1	0.43
-20	10	-20.73	10.06	0.73	0.06
-20	15	-20.26	16.35	0.26	1.35
-20	20	-21.43	21.96	1.43	1.96

The maximum and minimum positional deviations for the testing using *Part B* is shown in table 4.2.

Min/Max Recorded Values	
Max ΔX [cm]	1.43
Max ΔY [cm]	1.96
Min ΔX [cm]	0.1
Min ΔY [cm]	0.06

Table 4.2: Shows the minimum and maximum deviation for the testing using *Part B*.

It is important to note that there are uncertainties regarding this test. The parts that was detected were manually placed on the table as accurately as possible. Even though this was done using a reference grid, it is not guaranteed that the actual position of the part was 100% accurate. One other uncertainty is the accuracy of the 3D camera position calibration. The calibrated position of the camera will affect the position output for the parts when using the 3D object detection.

The test results show that the maximum positional error for both the x and y axis is below $2cm$. This is well within the margin of error that is tolerated in order for the 2D object detection to be performed based on this initial position, however, it does not meet the expected accuracy as stated in section 3.3.1. A brief discussion regarding this result is found in 5.2.

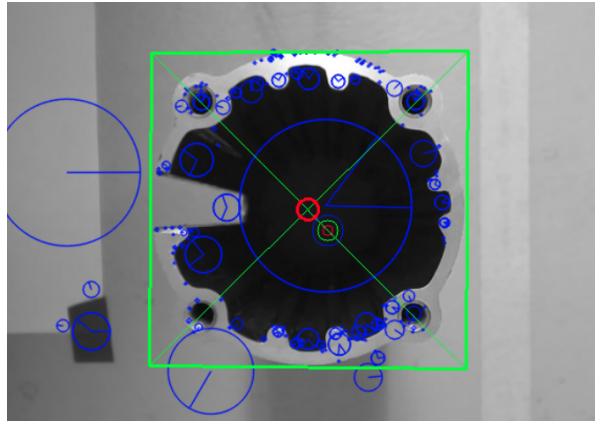


Figure 4.5: Shows the view from the 2D object detection camera when *Agilus 2* is positioned in the initial position found using 3D object detection.

Figure 4.5 shows an image taken from the 2D camera when positioned in the initial position found by 3D object detection. As is evident by the image, the position is not accurate enough to be used on its own, but is more than adequate to be used as the starting point for 2D object detection.

4.2.2 Testing and selecting global descriptor

The testing of global descriptors was carried out as described in section 3.3.2. The result from the matching process is presented in table 4.3 and 4.4. A brief description of the different table entries are described below.

Cluster nr. This number corresponds with the cluster number in the scene that is being matched. The scene contains 4 different clusters, where cluster number two is known to be the part we are searching for.

Segment nr. When using the CVFH descriptor, each cluster is separated into multiple segments. For each segment, a VFH descriptor is estimated. The CVFH descriptor is matched with a set of CVFH descriptors from the training set, and the segment with lowest confidence level is the best matching segment.

Best match This number corresponds with the best matching model from the training set. We manually estimated that model number 6 should be the best match, since it is the model closest to the object as seen in the 3D scene.

Confidence level This number corresponds with the difference between the descriptor of the object in the scene, and the descriptor of the best matching model. The lower this number is, the better the match is. Throughout this work, we found that a confidence level between 0-4000 usually dictates a good match (when using the VFH global descriptor).

Cluster nr.	Best match	Confidence level
1	41	14564.8
2	6	3043.06
3	32	33093
4	32	27204.6

Table 4.3: Shows the test result from a matching process using the VFH descriptor. The table entry with the best matching result is indicated in green.

The result from the test using the VFH descriptor shows a predictable result. The cluster that corresponds with the part we are searching for is cluster number two. This is the cluster with the lowest confidence level. In addition, the best matching model is, as expected, model number 6. However, the confidence level is quite high. This indicates a positive, but somewhat unreliable match.

Cluster nr.	Segment nr.	Best match	Confidence level
1	1	10	24495500
	2	74	18602800
	3	74	18928400
2	1	56	1637620
	2	8	3139210
	3	71	1320850
3	1	61	1089380
	2	4	2965940
4	1	43	174401

Table 4.4: Shows the test result from a matching process using the CVFH descriptor. The table entry with the best matching result is indicated in green.

The result from the test using the CVFH descriptor was unexpected. It was expected to achieve a positive and decisive match between a segment of cluster number two and one of the segments from model number six in the training set. This is not the case. The best matching cluster is number 4, which in this scene is just a cluster of noise left behind from the model segmentation step. In addition, the confidence level for this match is extremely high, and is not a value that is indicative of a positive match at all.

The result from this test shows that using the basic VFH descriptor will provide us with the best matching result, and overall reliability when it comes to the 3D object detection. The main benefits the CVFH descriptor holds over the VFH descriptor is the robustness when it comes to occluded scenes. This is not a part of our problem scenario, since the two parts that is to be assembled will be placed in separate areas on the table. Based on this we selected to use the VFH global descriptor.

4.2.3 Selecting keypoint and local descriptor estimator

The selection of the keypoint estimation method was done based on the work of [Filipe and Alexandre \(2014\)](#). Their comparison of the most common 3D keypoint selectors show that

both the SIFT3D and ISS3D method performs with equal repeatability. Based on this, we chose to use the SIFT3D keypoint estimator. Throughout this thesis, the conclusion made by Filipe and Alexandre (2014) has shown to be consistent and the use of SIFT3D as keypoint estimator performed as expected.

The selection of local descriptors was also made on the basis of previous work. The work by Alexandre (2012) shows that the PFH family of descriptors are the fastest to compute while still maintain high robustness with regards to viewpoint differences. Because of this, we chose to use the FPFH local descriptor. Throughout the work with this project, the FPFH local descriptor was found to perform as reliable as expected, and no problems caused by this choice was encountered.

4.2.4 Creating training sets

The C++ class used to generate training sets from 3D CAD models is called *Modelloader.cpp* (source code available in Appendix A). The initial functionality of this class was created by Adam Leon Kleppe. The class was modified to allow for feature estimation, and for the features to be added to the training set. The initial functionality was limited to viewpoint specific point cloud rendering and object pose information.

The resulting class is a useful tool for creating a training set, and also to load all the data in a training set from disk to the system memory. This allows for faster processing times when the 3D object detection routine is performed (since all features of the different point clouds located in a training set is pre-calculated).

The data output from this class is saved to a directory created under the root directory of the application using it. Each training set (each part) is located in its own sub directory under this directory.

Multiple different training sets are available online at the *GitHub* repository used for the software development (Larsen and Bjørkedal, 2016a). The training sets are located at "*agilus_master_project/trace_clouds*". The naming convention used for the training sets are "name - number of viewpoints - render resolution". Example of a training set name is "cone-42-200".

Figure 4.6 shows some of the different viewpoint point clouds generated from a part.

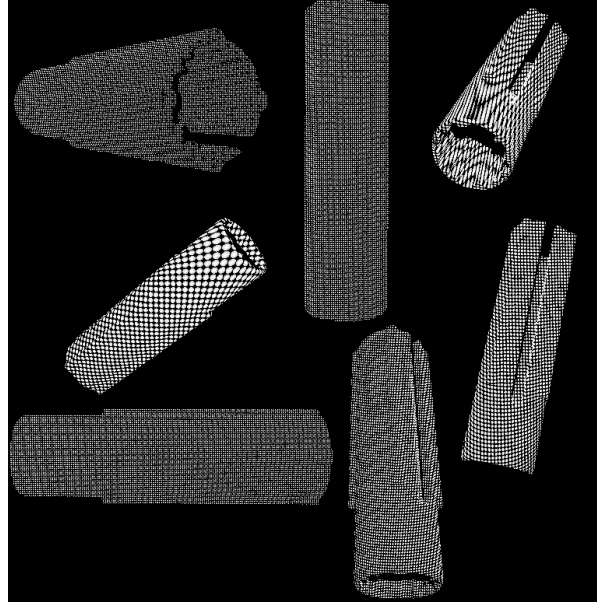


Figure 4.6: A collection of different point clouds that illustrates the different viewpoints generated in the process of creating a training set.

4.2.5 3D object detection

The result from the 3D global descriptor testing described in section 4.2.2 made it clear that the best option for 3D object detection using global descriptors was to use the *Viewpoint Feature Histogram* (VFH) global descriptor. Based on this choice, the resulting pipeline used to perform 3D object detection is shown in Figure 4.7. This pipeline uses a combination of global and local descriptors to perform a complete object detection and pose estimation process. The local descriptors are used to estimate an initial position based on the *Random Sample Consensus* approach, and the global descriptor is used for viewpoint matching (necessary to select the best suited model from the training set). The code implementation of the pipeline can be found in the `object_detection` method located in `pcl_filters.cpp` in Appendix A.

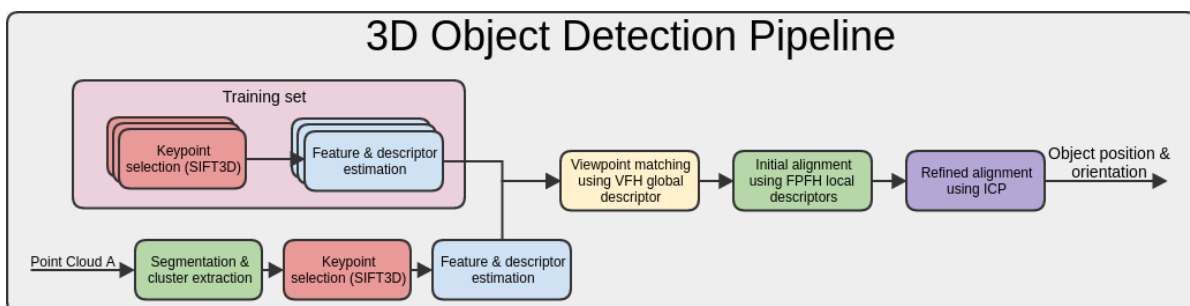


Figure 4.7: Illustrates the pipeline used for 3D object detection.

One problem that quickly became evident was the similarities of the two different parts. The 3D point cloud produced by the depth sensor lacks quite a lot of detail. The result of this is that the point clouds for the two different parts are hard to distinguish from each other. Both

parts have similar height, diameter and are equally featureless (both are symmetrical cylinders with few detectable features). This made the task of distinguishing the different parts from each other using global descriptor matching highly unreliable. In order to circumvent this issue, we decided to divide the working surface (the table in the robotic cell) in two, equally sized, working areas. We then assume that *part A* is always located in the first working area, and similar for *part B* located in the second working area. The two different working areas are illustrated in Figure 4.8.

Using this approach, we have yet to produce a scenario where the 3D object detection is unable to detect and estimate the position of the two parts. An example of such a detection and position estimation is shown in Figure 4.9. The best matching model for the two training sets used for the matching is placed on the original 3D scene captured using the 3D depth sensor.

Given that the parts are both symmetrical cylinders, the detected orientation is ambiguous. However, since the final position and orientation of the parts is detected using 2D object detection, this does not cause any issues. The important information gained from the 3D object detection is an approximate position of the parts in the xy -plane of the table.

A demonstration video produced to show the complete automated assembly of the two parts is available online at [Larsen and Bjørkedal \(2016b\)](#) and through the digital appendix for this thesis. The contents of the digital appendix is described in Appendix D. This video also show the working 3D object detection process.

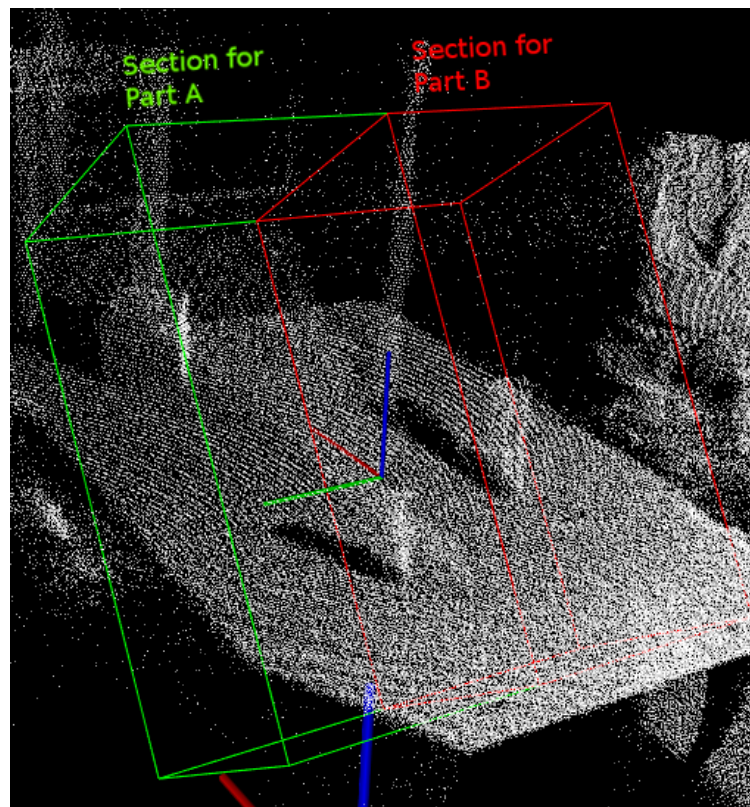


Figure 4.8: Illustrates the virtual separation of the two work areas used when performing the 3D object detection routine.

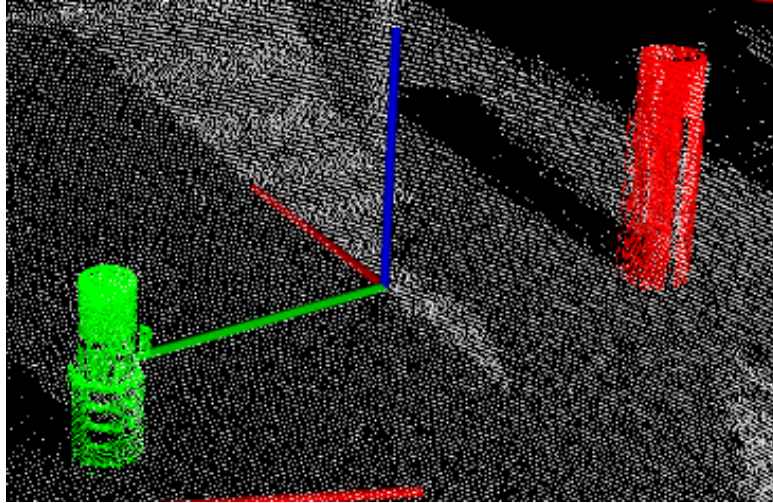


Figure 4.9: Shows the result from a 3D object detection process.

4.3 2D computer vision

4.3.1 Processing time

The approach that is examined in this thesis is aimed at a flexible assembly task for use in the industry. This means that there are, in many cases, defined limits for the cycle time of each task. This may lead to a demand for lower processing time of object detection. For research purposes this is not considered a problem, but it is of interest to determine if the assembly task can be solved using faster methods and how they compare to the slower ones.

The testing of processing time was performed as described in section 3.3.3. The needed time to detect keypoints and extract descriptors in the test scene is presented in Figure 4.10.

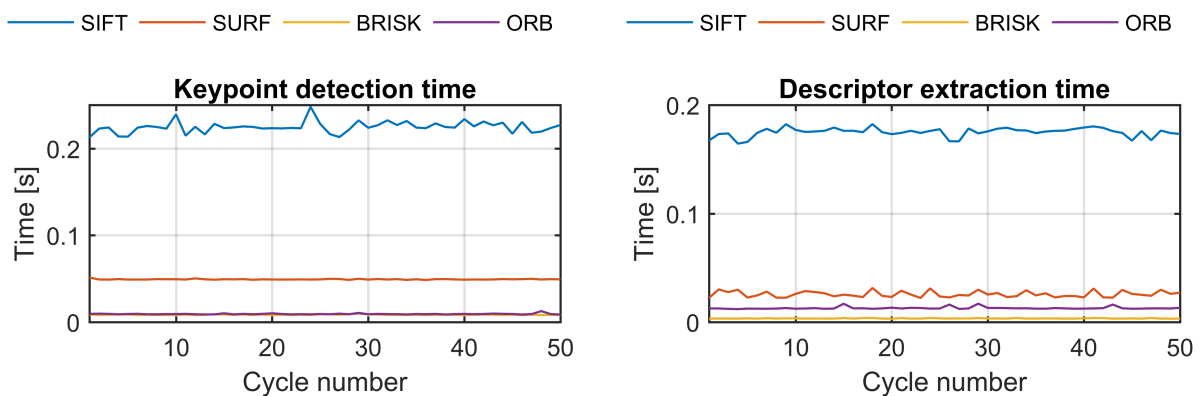


Figure 4.10: Left: Time needed to detect keypoints in a test scene. Right: Time needed to extract descriptors from the detected keypoints.

As evident from Figure 4.10, SIFT is the slowest both at keypoint detection and descriptor extraction. The result is as expected, where the convolution by integral images and determinant-of-Hessian for detection of SURF features is faster than the difference-of-Gaussian approach for

detection of SIFT features. Both BRISK and ORB are really fast at this task, mostly because of the use of FAST as base for keypoint detection. In terms of descriptor extraction, the ones represented by binary bit-strings from intensity tests, i.e. BRISK and ORB, are really fast to compute. The use of Haar wavelet filters and integral images speeds up the SURF descriptor, but it is still slower than the binary descriptors. SIFT and its approach using histogram of oriented gradients for description is outperformed in terms of speed. As evident from Table 4.5 and Table 4.6, BRISK is faster than ORB at both tasks, marked with green cell colour.

Mean keypoint detection time [s]			
SIFT	SURF	BRISK	ORB
0.22467492	0.04927685	0.0084592534	0.009322315

Table 4.5: The mean keypoint detection time of the 50 cycles illustrated in Figure 4.10.

Mean descriptor extraction time [s]			
SIFT	SURF	BRISK	ORB
0.17505564	0.025696292	0.0033582588	0.012909806

Table 4.6: The mean descriptor extraction time of the 50 cycles illustrated in Figure 4.10.

The next part of the test determines the time needed to match descriptors between a query image and the training images of the test scene. Figure 4.11 presents the test results and the total processing time needed from detection to a final match is complete.

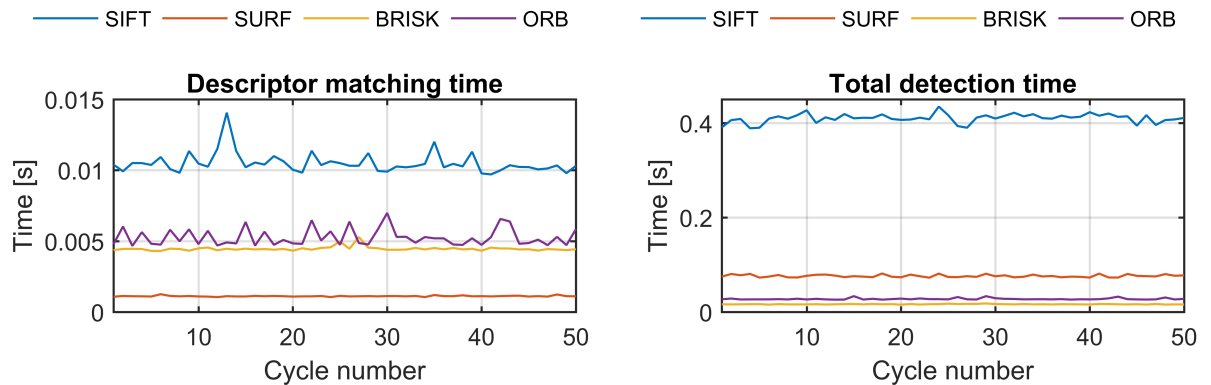


Figure 4.11: Left: Time needed to extract descriptor from the detected keypoints. Right: The total time needed to detect keypoints, extract descriptors from the test scene and match the descriptors with descriptors from a query image.

The matching method used during this test was brute-force matching with $n = 2$ best matches sorted by a distance ratio test at threshold 0.9. The ratio test is described in section 2.6.5. The distance measurement used for SIFT and SURF is the L1 norm - Manhattan distance, and Hamming norm for BRISK and ORB. SIFT is once again the slowest. However, this is not unexpected considering the descriptor type and size. Compared to a binary descriptor like BRISK and ORB, SIFT is more demanding in terms of computational cost when matching because of the different natures of the descriptors. A surprise from this test is the matching time of the SURF descriptors, which is even faster than both BRISK and ORB. The mean

matching time of the 50 test cycles is shown in Table 4.7. In total, BRISK is the fastest algorithm in this particular test as evident from Figure 4.11 and Table 4.8.

Mean descriptor matching time [s]			
SIFT	SURF	BRISK	ORB
0.0105037502	0.0011120464	0.0044597802	0.0052647372

Table 4.7: The mean descriptor matching time of the 50 cycles illustrated in Figure 4.11.

Mean total detection time [s]			
SIFT	SURF	BRISK	ORB
0.4102343	0.07608519	0.016277292	0.027496862

Table 4.8: The mean total detection time of the 50 cycles illustrated in Figure 4.11.

Worth noticing is that the processing time is directly connected to the number of layers used for scale pyramids, parameters for keypoint rejection and other parameters and thresholds for each individual algorithm. The results from this test was produced using standard parameters as specified by the OpenCV documentation, except slight adjustments of thresholds in order to control the number of keypoints estimated. This is further discussed in section 5.1.

4.3.2 Matching stability

In order to make a choice about which algorithm to use for the 2D part of the assembly task, a matching stability test was conducted. Each part was photographed from both ends at a decided reference orientation, resulting in four query images to detect in the training scene. These images are shown in Figure 4.12.



Figure 4.12: The query images used to detect the parts. From the left: Part A - top view, part A - bottom view, part B - bottom view, part B - top view.

The test was carried out as explained in section 3.3.4 and yields the results shown in Table 4.9, 4.10, 4.11 and 4.12. Note that BRISK is not included in any of these tests, as it failed to produce any matches with the parts shown in Figure 4.12. The data in the tables presented below therefore only consists of data where an actual match was achieved.

Part A - top view						
Keypoint	Descriptor	Matcher	RGB	λ_{min} [cm]	λ_{max} [cm]	Note
SIFT	SIFT	Brute-force	Yes	15.3	30.3	Stable
ORB	ORB	Brute-force	Yes	15.3	-	Low stability

Table 4.9: Matching stability of part A - top view. λ denotes the distance between the camera lens and the object.

Part B - bottom view						
Keypoint	Descriptor	Matcher	RGB	λ_{min} [cm]	λ_{max} [cm]	Note
SIFT	SIFT	Brute-force	Yes	9.8	39.8	Stable
SURF	SURF	Brute-force	Yes	9.8	35.3	Stable
ORB	ORB	Brute-force	Yes	9.8	21.8	Low stability

Table 4.10: Matching stability of part B - bottom view. λ denotes the distance between the camera lens and the object.

Part A - bottom view						
Keypoint	Descriptor	Matcher	RGB	λ_{min} [cm]	λ_{max} [cm]	Note
SIFT	SIFT	Brute-force	Yes	10.3	35.3	Stable
ORB	ORB	Brute-force	Yes	12.3	18.3	Small range

Table 4.11: Matching stability of part A - bottom view. λ denotes the distance between the camera lens and the object.

Part B - top view						
Keypoint	Descriptor	Matcher	RGB	λ_{min} [cm]	λ_{max} [cm]	Note
SIFT	SIFT	Brute-force	Yes	12.8	34.8	Stable
SURF	SURF	Brute-force	Yes	12.8	34.8	Stable
ORB	ORB	Brute-force	Yes	9.8	23.8	Low stability

Table 4.12: Matching stability of part B - top view. λ denotes the distance between the camera lens and the object.

As evident from Table 4.9 and Table 4.11, *part A* is a difficult part to detect with satisfying stability. SIFT is the only algorithm that managed to do this for both views of the part. ORB did also achieve some matching results, but with low stability. Because of this, ORB is not suitable for this specific assembly operation. Table 4.10 and Table 4.12 shows that SIFT is the most stable algorithm for detection of *part B*. SURF is also a good candidate with stable matching comparable to SIFT at the same distance range between camera and object. ORB may be used, but it is not the best solution because of less stable results and a smaller range.

The parts can only be assembled in one way, because of the way they are designed. Based on the previously presented results, *part B - bottom view* was chosen as the best view to detect *part B*, thus *part A* needs to be detected using the top view (because of the particular way the parts are assembled). Figure 4.13 shows the desired assembly pose of the parts.

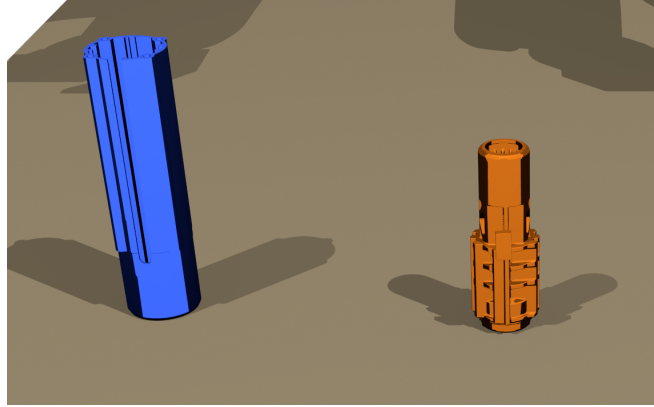


Figure 4.13: Shows the desired pose of the parts before assembly. Part A (right) is to be assembled into part B (left).

Based on these results, SIFT was chosen for detection of *part A* - *top view* and *part B* - *bottom view* at a distance of $\lambda = 15.3\text{cm}$ and $\lambda = 13.8\text{cm}$ (can be as low as 10.3cm), respectively.

4.3.3 Orientation stability

Matching stability is important and directly connected to the computation of the orientation of the detected parts. Without a stable matching, the difference between each computed orientation will be too large to assemble *part A* and *part B* with success. The desired result is to minimize this difference and determine the orientation with certainty. To further test the stability of the 2D object detection, a test was conducted as described in section 3.3.5 for *part A* and *part B* individually.

As evident from the results in section 4.3.2, ORB is not stable for detection of *part A*. A combination of SIFT as keypoint detector and SURF as descriptor extractor, denoted SIFT/SURF, was then added to the following orientation stability test of *part A*. The results are presented in Figure 4.14.

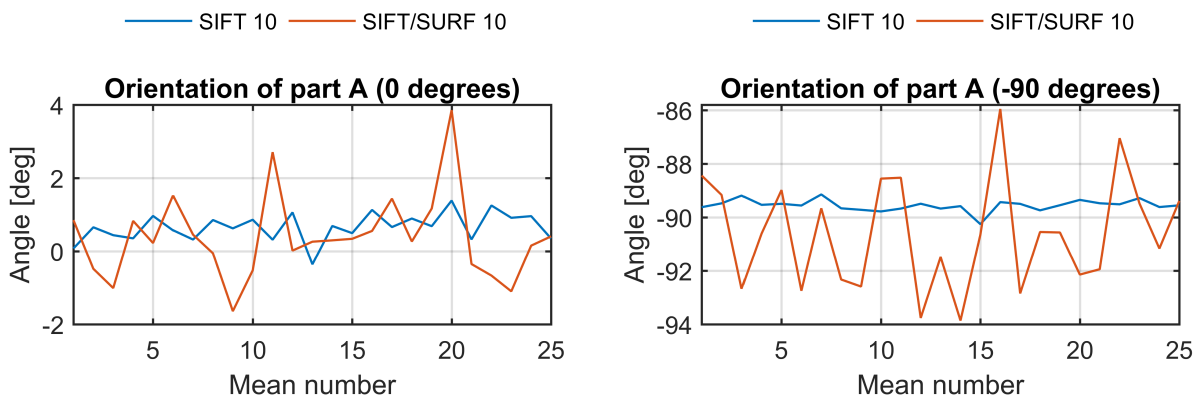


Figure 4.14: Orientation of *part A* detected at approximately 0 degrees (left) and -90 degrees (right). Each data point is the mean of 10 measurements.

As seen from Figure 4.14, SIFT is clearly more stable. SIFT/SURF works, but is not as stable

as SIFT. The difference between the maximum and minimum measured orientations for *part A* are presented in Table 4.13.

0 degrees		-90 degrees	
SIFT	SIFT/SURF	SIFT	SIFT/SURF
1.746911	5.49935	1.1102	7.9095

Table 4.13: The difference between the maximum and minimum measured orientations for *part A*, as presented in Figure 4.14.

As presented in section 4.3.2, SIFT, SURF and ORB are all usable for detection of *part B*. The results of the orientation stability test are presented in Figure 4.15.

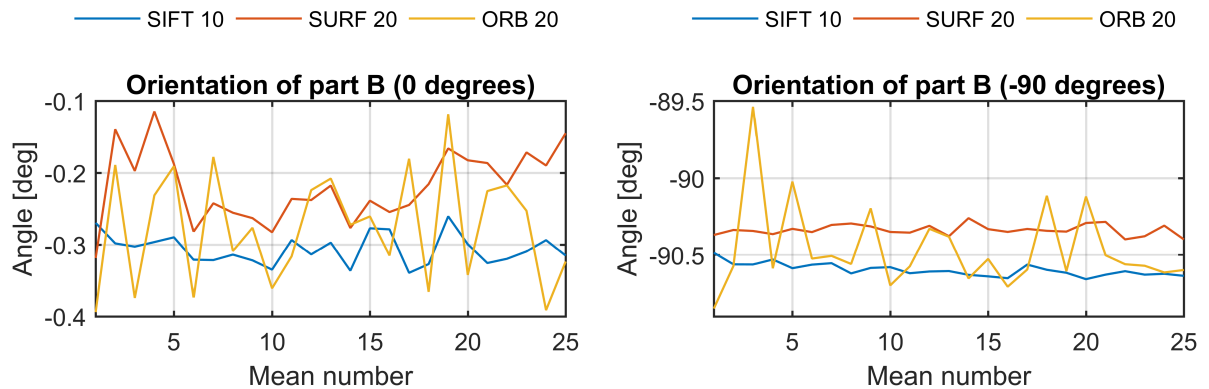


Figure 4.15: Orientation of *part B* detected at approximately 0 degrees (left) and -90 degrees (right). Each data point is the mean of 10 or 20 measurements.

As evident from Figure 4.15, SIFT is once again the approach with highest stability. SURF performs better than ORB, but does not compete with SIFT. ORB is shown to be highly unstable compared to SIFT and SURF, thus resulting in measurement spikes that can not be tolerated. The difference between the maximum and minimum value of each test is shown in Table 4.14.

0 degrees			-90 degrees		
SIFT	SURF	ORB	SIFT	SURF	ORB
0.078886	0.204128	0.275597	0.1721	0.1379	1.3109

Table 4.14: The difference between the maximum and minimum measured orientations for *part B*, as presented in Figure 4.15.

The results presented in this section points towards SIFT for stable detection with consistently low difference between the computed orientations. The final solution computes the mean of 20 detected orientations using SIFT.

4.3.4 2D object detection

Based on the results obtained from the test results presented in section 4.3.1, SIFT is the slowest of the four algorithms examined in this thesis. It is on the other hand the most robust

and stable choice for detection of both *part A* and *part B*. Since a stable object detection is of great importance to successfully assemble the parts, SIFT is used to ensure this.

The final object detection algorithm is implemented in a C++ ROS application as described in section 3.2.4. The class is named *object_2D_matcher.cpp* and is available in Appendix B. It utilized a class named *openCV_matching.cpp* (available in Appendix B). This class is based on OpenCV and implements all the methods needed in order to capture, process and visualize image matching. Figure 4.16 illustrates the object detection procedure.

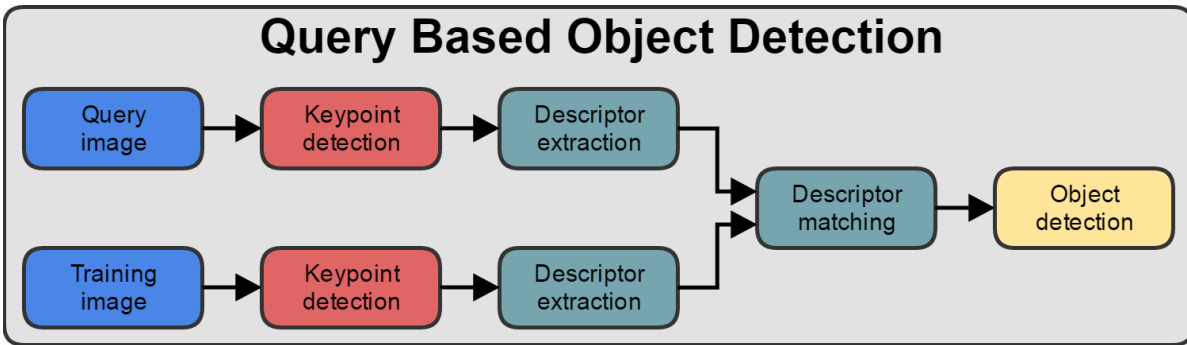


Figure 4.16: Illustrates the implemented object detection.

Object detection is obtained by acquiring a training image of the scene and a query image of the object we are looking for in the scene. A set of keypoints are detected for both images. From these sets of keypoints a set of descriptor feature vectors are extracted. The query descriptors are matched with the training descriptors using brute-force. After the descriptor feature vectors of the two images are compared, a set of good matches are returned. If the number of good matches is higher than a given threshold the object detection is considered to be successful and the position and orientation of the object is computed. As evident from the code in Appendix B and the algorithm in section 3.2.4, the object detection runs in a loop for each captured image of the training scene. The features of the query image is computed once when the ROS node is launched and matched against each captured training scene. Figure 4.17 shows a successful detection of *part A* and *part B* using SIFT.

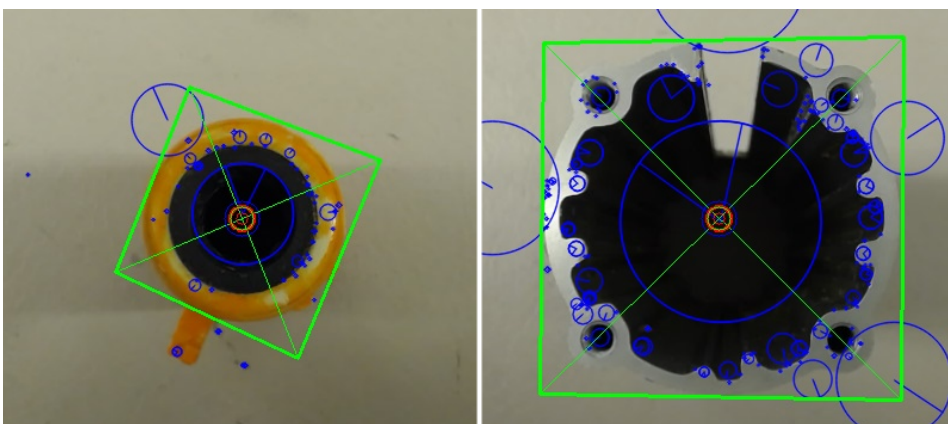


Figure 4.17: Successful detection of *part A* rotated approximately -30 degrees and *part B* rotated approximately 180 degrees.

4.4 Software solution

4.4.1 ROS communication

The software solution used to run the robotic cell is implemented in the Robotic Operating System (ROS) framework. The full system is separated in multiple different applications that run in parallel. The different applications used is shown in Figure 4.18, together with the information that is sent between them.

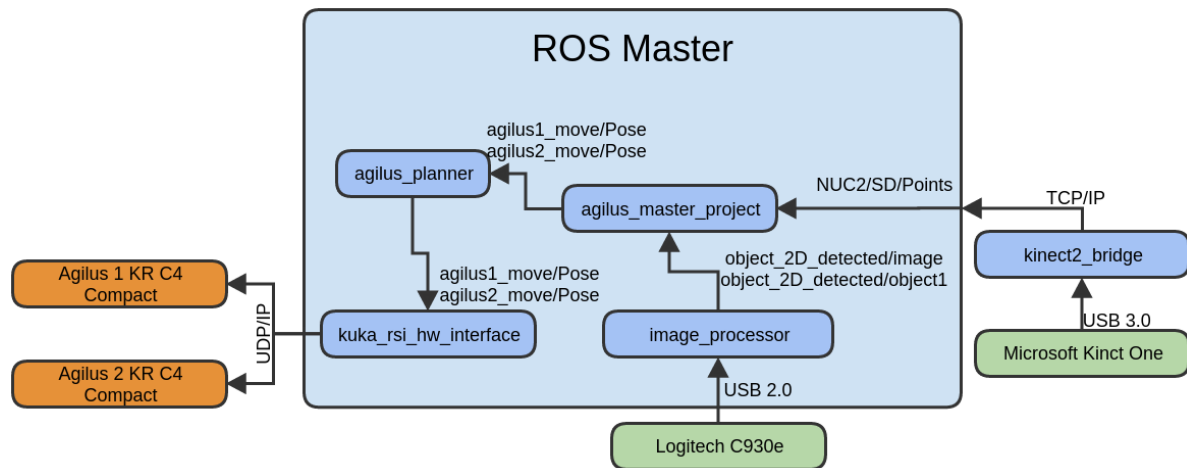


Figure 4.18: Shows the different applications that run in the ROS framework in order to perform the automated assembly task.

The following is an explanation of the purpose that each application serve, and what data is being sent between them.

kinect2_bridge An instance of the Kinect 3D image grabber (Wiedemeyer, 2016). This application publishes a topic called *NUC2/SD/Points* that contain the point cloud acquired from the 3D camera. This topic is accessed by the *agilus_master_project* application and used for 3D object detection.

image_processor This application acquires the video feed from the Logitech C930e web camera and runs the 2D object detection. The output from this application is published in two different topics. The *object_2D_detected/image* topic contains the image acquired from the web camera with some added graphics that illustrates the optical center of the camera and the detected object. The *object_2D_detected/object1* topic contains the positional and angular data about the detected object.

agilus_planner The purpose of this application is to publish services for controlling the robotic manipulators. This is done to provide a simple interface for robotic control that is easily accessible from within the ROS framework. The data provided to this application through a service call is turned into actual motion planning for the two robotic manipulators. This application outputs data to the *kuka_rsi_hw_interface* application.

kuka_rsi_hw_interface The *kuka_rsi_hw_interface* application is responsible for the movement of the robotic manipulators. This is done through the *Kuka Robot Sensor Interface*

(RSI). The input data to this application is the desired position of the robot, which is sent to the robotic controllers over the network using the UDP/IP protocol.

agilus_master_project This is the main application that runs the entire assembly process. This application receives input data from the *kinect2_bridge* and *image_processor* applications and runs further processing. This application runs the 3D object detection. The graphical user interface used to control the assembly process is also produced by this application.

4.4.2 Automated assembly sequence

The complete system produced for this thesis performs a series of actions in a particular sequence in order to perform the automated assembly task. This sequence is carried out by the *agilus_master_project* application. Figure 4.19 illustrates the sequence of events performed in order to automatically assemble the two parts. The illustration uses simple colour codes to identify what is performed in each step. Light blue illustrates 3D point cloud processing and 3D object detection, orange illustrates controlled movement of the robotic manipulators and green illustrates operations using 2D object detection.

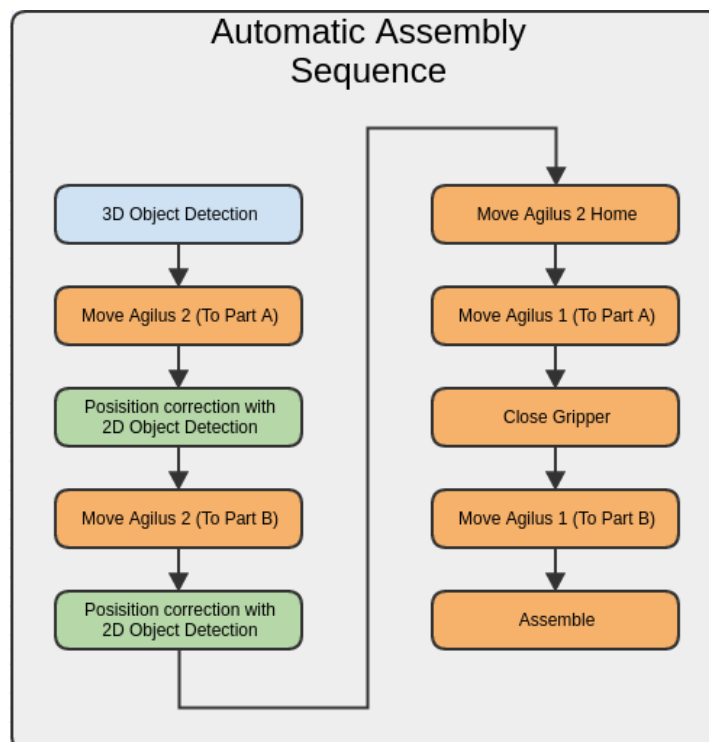


Figure 4.19: Illustrates the sequence used to perform an automatic assembly of two parts.

4.4.3 Applications

The `agilus_master_project` application

The `agilus_master_project` program is the master application that runs the completed automatic assembly task. This program was created to produce a polished product where the main functionality is to perform the automated assembly task. A demonstration video showing the functionality of the application, as well as a complete automated assembly task is available online at [Larsen and Bjørkedal \(2016b\)](#) and through the digital appendix (the content of the digital appendix is described in appendix D). In addition to perform the assembly task, some useful features that are not specific for this task are included. These features are:

- Manual control of the two robotic manipulators. This can be done both as a relative movement with relation to the home position of the robot, or as an absolute movement with relation to the world reference frame.
- Visualizing any 3D point cloud feed published within ROS.
- Visualize any 3D point cloud saved to a `.PCD` file.
- Visualize any 2D camera image feed published within ROS.
- Manually select keypoint detector, descriptor extractor and matching method for 2D object detection.
- Manually select the reference image used for 2D object detection.
- Open and close the pneumatic linear gripper mounted on Agilus 1.
- Create a training set with customizable parameters based on a 3D CAD model. The 3D CAD model used must be of the `.STL` format.
- Manually run a 3D object detection process with a user selectable training set as the reference model (the model we want to detect in the 3D scene).

This application consists of the following classes:

main.cpp - This is the main entry point of the application. The main class creates one instance of the **main_window.cpp** which initializes the graphical user interface. The source code is available through the digital appendix as described in Appendix D.

main_window.cpp - This class handles all the user interaction. The graphical user interface is connected to this class, and all user actions performed in the user interface is defined here. The source code is available through the digital appendix as described in Appendix D.

modelloader.cpp - This class handles both the creation of new training set, and loading pre-existing training sets to the system memory. Source code is available in Appendix A.

pcl_filters.cpp - This class handles all 3D point cloud processing. It implements all the tools necessary to perform a complete 3D object detection process. It also contains functions that allows for easy visualization of 3D point clouds in the graphical user interface. This

class was initially created for the `qt_filter_tester` application as a toolbox for handling 3D point clouds. Source code is available in Appendix A.

qnode.cpp - This class runs in a separate thread, and is responsible for all communication within ROS. All data publication and acquisition in ROS is done through this class. The source code is available through the digital appendix as described in Appendix D.

Figure 4.20 shows the graphical user interface for the `agilus_master_project` application as presented when the application is launched.

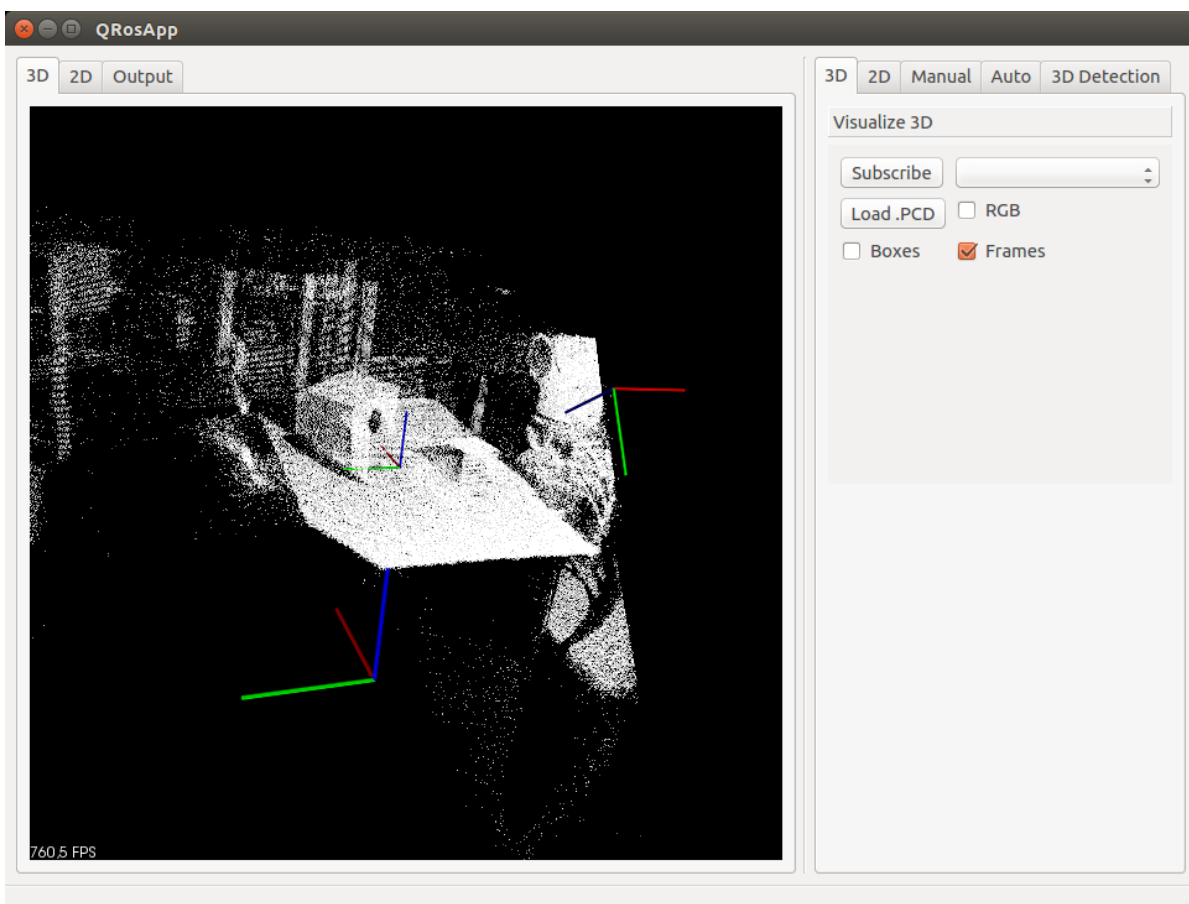
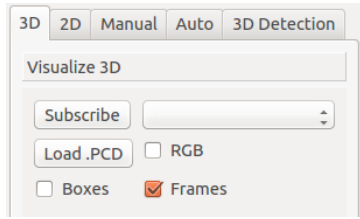
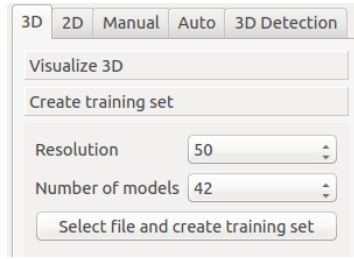


Figure 4.20: The main window of the `agilus_master_project` application as displayed at launch.

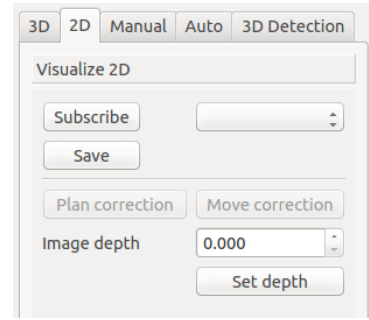
The full source code for this application is available in the digital appendix. The digital appendix is described in Appendix D. The following figures show a more detailed view of the different actions available through this application.



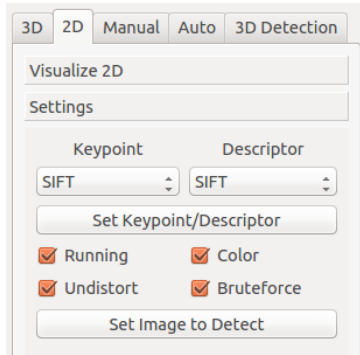
(a) Shows the user input related to the 3D point cloud visualization.



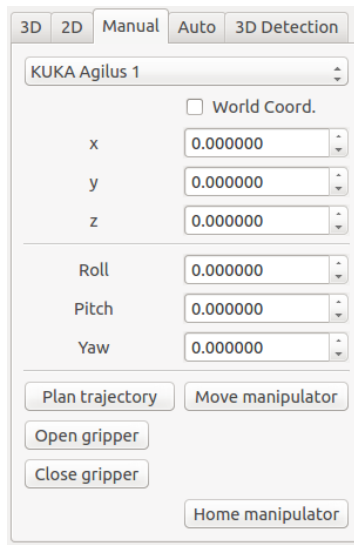
(b) Shows the user input related to creating a training set.



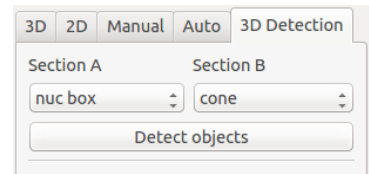
(c) Shows the user input related to the 2D image visualization.



(a) Shows the user input related to the 2D object detection.



(b) Shows the user input related to manual control of the robotic manipulators.



(c) Shows the user input related to manually running 3D object detection.

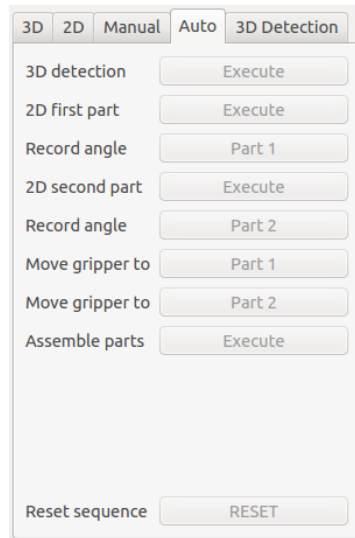


Figure 4.23: Shows the user input section that is used to run the automated assembly sequence.

The `qt_filter_tester` application

The `qt_filter_tester` program was initially created as a tool for working with 3D point clouds. It contains graphical tools for performing the most common filtering and processing tasks. This application was used to test different approaches for 3D object detection as well as different parameters for all the different filters and algorithms used. The available functionality in this application is the following:

- Create training sets from a user specified 3D CAD models (the specified models must be of the *.STL* format).
- Visualizing 3D point cloud images saved as *.PCD* format.
- Perform the following filtering and processing tasks:
 - Passthrough filtering
 - Voxel grid filtering
 - Median filtering
 - Shadow point removal filtering
 - Normal estimation
 - Statistical outlier removal filtering
 - Plane model segmentation
 - Euclidean cluster extraction
 - Bilateral filtering
- Visualize the result from the above mentioned filtering and processing actions.

- Save the filtering results as a *.PCD* file.

The application consists of the following classes:

main.cpp - This is the main entry point of the application. The main class creates one instance of the **main_window.cpp** which initializes the graphical user interface. The source code is available through the digital appendix as described in Appendix D.

main_window.cpp - This class handles all the user interaction. The graphical user interface is connected to this class, and all user actions performed in the user interface is defined here. The source code is available through the digital appendix as described in Appendix D.

modelloader.cpp - This class handles both the creation of new training set, and loading pre-existing training sets to the system memory. Source code is available in Appendix A.

pcl_filters.cpp - This class handles all 3D point cloud processing. This class implements all the tools necessary to perform a complete 3D object detection process. It also contains functions that allows for easy visualization of 3D point clouds in the graphical user interface. This class was initially created for the **qt_filter_tester** application as a toolbox for handling 3D point clouds. Source code is available in Appendix A.

qnode.cpp - This class runs in a separate thread, and is responsible for all communication within ROS. This application does not require any ROS communication, but it was created from a ROS template application. This class was not removed in order to keep the possibility of ROS communication open. Because of this, the content of this class is limited to the basic initialization of a ROS node in the ROS framework. The source code is available through the digital appendix as described in Appendix D.

This application is not used when running the robotic cell, but served its purpose as a test platform when working with 3D point clouds. The full source code for this application is available in the digital appendix. The digital appendix is described in Appendix D.

The **image_processor** ROS node

The **image_processor** ROS node was initially created for testing and evaluation of object detection algorithms like SIFT, SURF, BRISK and ORB using OpenCV. Due to successful detection of the parts to be assembled in this thesis the functionality of the node was extended for further use in the final solution. The functionality of the node is:

- Capture the video stream from a USB web camera.
- Load any stored image of format *.png* or *.jpg* as reference matching image.
- Implements 7 keypoint detectors:
 - SIFT, SURF, BRISK, ORB, STAR, FAST and AKAZE.
- Implements 7 descriptor extractors:
 - SIFT, SURF, BRISK, ORB, FREAK, BRIEF and AKAZE.
- Descriptor matching by brute-force or FLANN.

- Publish image data of the detected object via ROS.
- Publish the position and orientation of the detected object ROS.
- Controlled by ROS services.

The node consists of the following classes:

object_2D_matcher.cpp - This is the main entry point of the node. It initializes the properties of the object detection algorithm such as keypoint detector, descriptor extractor, matching type and video resolution. In addition it advertises ROS services for control. Callback methods for each service is implemented. Methods from an instance of **openCV_matching.cpp** is utilized in an object detection loop. The source code is available in Appendix B.

openCV_matching.cpp - This class handles all the crucial image processing using OpenCV. It implements methods for object detection and computation of object image coordinates and orientation needed to perform the object detection loop as implemented in **object_2D_matcher.cpp**. Source code is available in Appendix B.

calibration.cpp - This is a stand-alone node for calibration of camera parameters. It is not directly connected to the above classes. However, it is needed in order to provide a K-matrix and distortion coefficients for use in the main object detection ROS node. The code is originally a sample code from the OpenCV repository at Github ([OpenCV, 2015c](#)). It is used with slight modifications in order to calibrate images of resolution 1280×720 pixels. The source code is available in the digital appendix. The digital appendix is described in Appendix D.

The **agilus_planner** ROS node

The **agilus_planner** program is created as a tool for simpler interfacing with the manipulators via ROS. It advertise ROS services for trajectory planning and execution. This is used as the main interface from **agilus_master_project** in order to move the manipulators based on 3D and 2D object detection. The node consists of the following classes:

robot_movement.cpp - This is the main entry point. It advertises two services, *go_to_pose* and *plan_pose*. Callback methods for each service is defined. These methods utilize the methods implemented in **robot_planning_execution.cpp** for trajectory planning. This node may run for each *move_group* available in MoveIt!, where an instance of **robot_planning_execution.cpp** is created for each *move_group*. The source code is available in Appendix C.

robot_planning_execution.cpp - This class handles the computation of a linearly interpolated trajectory from a current pose to a target pose. The target pose may be specified as relative to the current pose or in world coordinates. The trajectory is sent to MoveIt! via the *move_group* interface. This code is originally written by Adam Leon Kleppe. Source code is available in the digital appendix. The digital appendix is described in Appendix D.

4.5 Automated assembly solution

By combining the 3D and 2D computer vision systems in a software solution as presented in section 4.4, the parts are automatically assembled using robotic manipulators. A full assembly is executed by detecting the approximated positions of the objects at the table using 3D object detection. The *eye-in-hand* system is then positioned above each part in order to refine their position using the computed image coordinates and orientation. This is illustrated in Figure 4.24. When both parts have been detected, the gripper is rotated about its end-effector z -axis to the computed orientation of *part A* and the part is picked up. This is done to ensure that *part A* is gripped approximately in the same way every time. Figure 4.25 illustrates the orientation of the gripper before gripping the part.

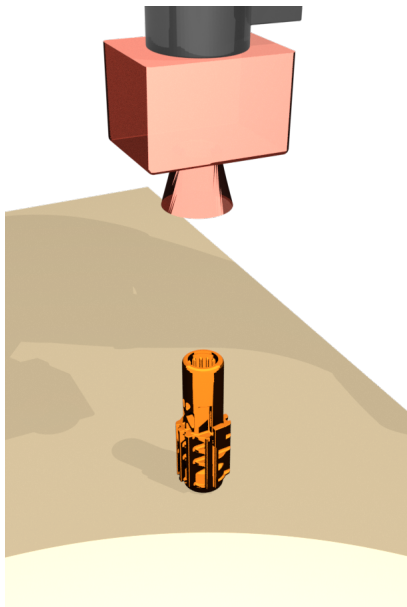


Figure 4.24: Camera positioned above *part A* using the position acquired from the 3D system.

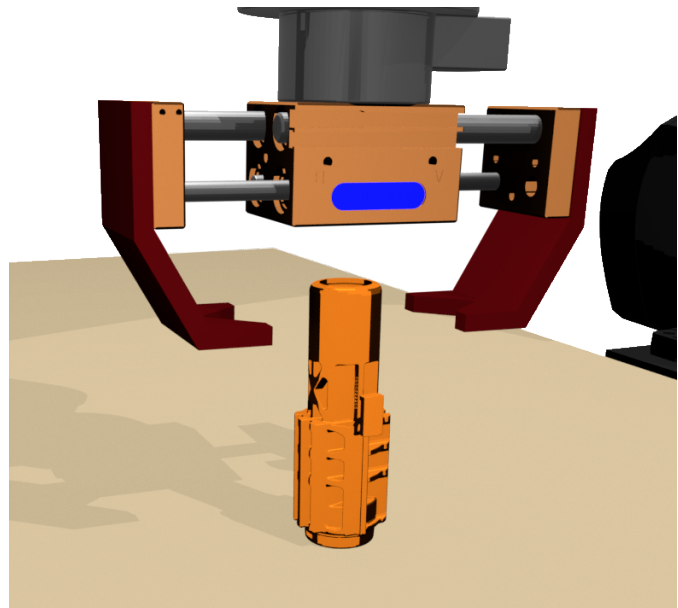


Figure 4.25: Gripper picking up *part A* at the refined position and orientation acquired from the 2D system.

The gripper holding *part A* is then positioned above *part B* and rotated about its z -axis to the detected orientation of *part B*. The parts are then assembled as illustrated in Figure 4.26. In order for this to work, it is important that the empirical calibration as explained in section 3.1.5 has been accurately conducted. It eliminates the offset between the camera optical axis and the gripper end-effector z -axis. In addition, by performing this calibration for both *part A* and *part B* individually, a center offset between the query images used for matching will be minimized. This offset *error* affects the refined position acquired from the 2D system, and must be eliminated. This is further discussed in section 5.1.

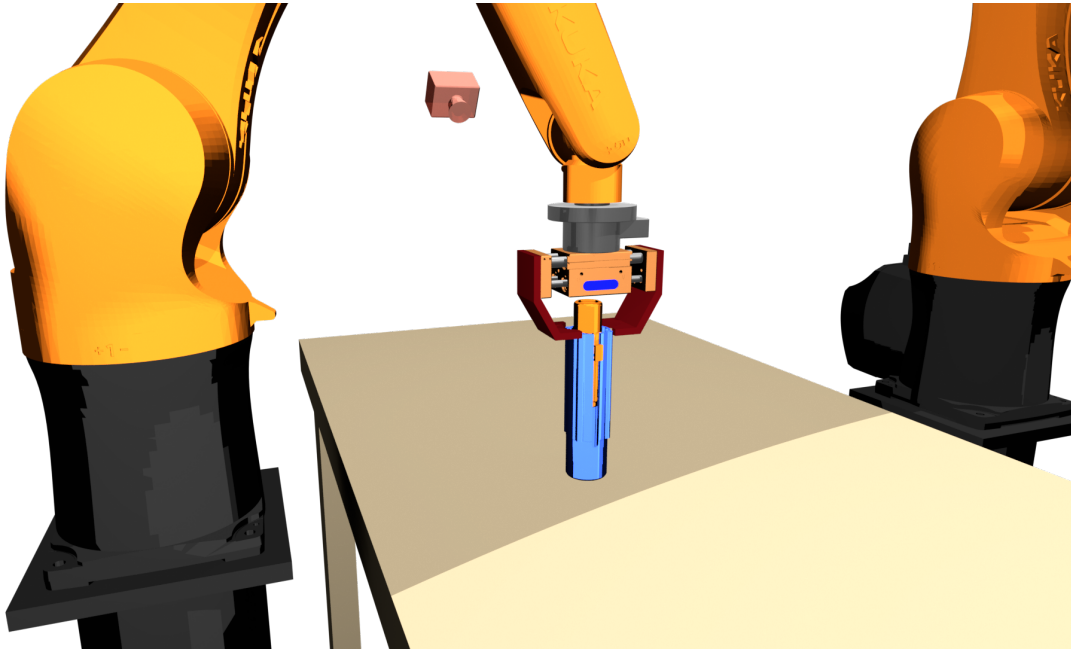


Figure 4.26: Illustration of a successful assembly of *part A* into *part B*.

Another offset revealed itself during test assemblies. The query images used for 2D detection is also rotated in-plane relative to each other. This is a constant offset and it was eliminated by adding 3.5° of orientation about the gripper z -axis before deploying *part A* into *part B*. Figure 4.27 shows the result from this. The positional accuracy is good in both cases, and the orientation is corrected as seen to the right. This is discussed in section 5.1.

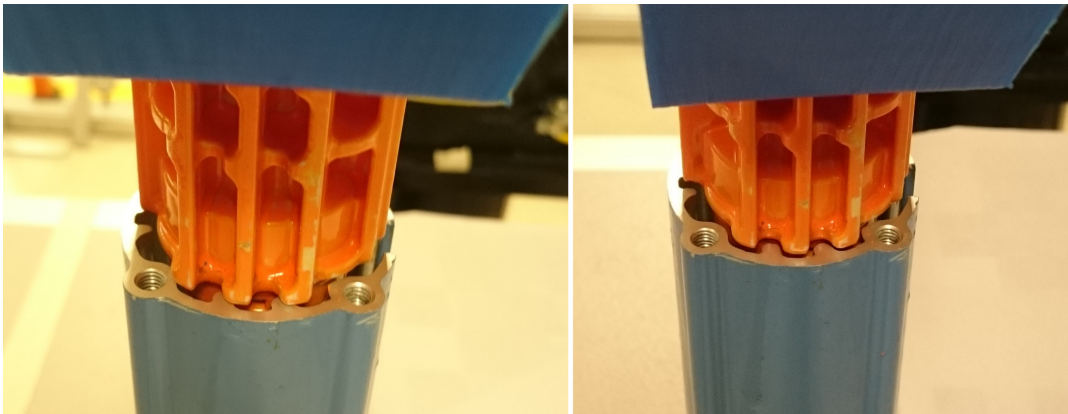


Figure 4.27: Left: Assembly operation conducted without correction for in-plane rotation offset between query images. Right: Another assembly operation with correction for in-plane rotation offset.

Testing performed in the robotic lab at the robotics lab at the *Department of Production and Quality Engineering* showed that this assembly task was successful at 7/10 unique attempts with different positions and orientations of the parts. A video showing 4 successful assembly attempts and the graphical user interface used to control the assembly is available in the digital

appendix. The digital appendix is described in Appendix D. The video is also available online at [Larsen and Bjørkedal \(2016b\)](#).

Chapter 5: Discussion

5.1 2D computer vision

The results presented in section 4.3 shows that object detection using SIFT keypoints and descriptors is sufficient for the assembly task presented in this thesis. Based on results presented in available literature regarding object detection, the expectations of SIFT and SURF were high. BRISK and ORB were considered as alternatives worth testing, although they did not fulfill the demands for stability as evident from the tests presented in 4.3. We believe that this has to do with the lack of strong features at both *part A* and *part B*. *Part A* is particularly difficult to detect, and the fact that SIFT managed to do this, proves the robustness of the over 16 years old algorithm as originally presented by David G. Lowe in 1999.

Although our results show that SURF, BRISK and ORB is not capable of detecting *part A* and *part B*, there is a possibility that they may do so if the parameters of the algorithms are tweaked. This means that the number of layers in the scale pyramids may be increased with a finer scale step between each layer, or thresholds for detection and rejection of keypoints may be changed to allow more keypoint candidates to be acquired. This may yield a negative impact on processing time. However, SURF, BRISK and ORB is already at a whole other level than SIFT in total computational time needed, which justifies the sacrifice in computational efficiency. This approach is not systematically tested, thus it may not yield the expected results. On the other hand, a possibility is that the SURF descriptor and the binary descriptors based on comparing pixel intensities used in BRISK and ORB is just not discriminative enough for such objects as the two parts used in this thesis.

Considering the stability of computed object orientation as presented in section 4.3.3, SIFT provides adequate stability. The orientation is computed using the corners of the object as drawn in the training scene using the homography from the points matched in the query image to the points matched in the training scene. If this homography is unstable due to unstable matching, the marked object plane in the training scene will also be unstable. Since the angle of orientation is computed using the corners of this plane, the difference between each computed angle will increase with poor stability. The solution used in order to acquire a more stable value is the mean value of 20 orientations. This is adequate in most assembly operations as evident from section 4.5. However, it is not the optimal solution considering that one poor data point of significant magnitude is enough to bring the mean value out of the accepted range. More advanced filtering may eliminate such spikes of bad data, thus increasing the assembly success-rate.

The computed orientation and positional accuracy when assembling the parts is also dependent on the query image of the object we are looking for. *Part A* and *part B* were photographed at a reference view representing zero degrees of orientation. This is the orientation that corresponds with the configuration of the parts in order to perform a successful assembly. This will not be fully accurate, and there will most likely be an offset in orientation between the query images,

thus resulting in unsuccessful assembly. However, this offset is always constant and may be handled. Because of the empirical calibration method used to detect the offset between the optical axis and end-effector axis, the positional accuracy is adequate for robotic assembly as evident from section 4.5. However, there is a clear error in orientation in order to assemble *part A* and *part B*. This error was detected and corrected as described in section 4.5.

Considering that the error is constant, the main focus should be to photograph the parts at approximately correct views, but under same conditions as the actual assembly task. This means that light sources, camera used and distance to object should correspond to the conditions in the actual process. This is to ensure that SIFT returns the most stable matching possible, thus keeping a low difference between each measured orientation.

5.2 3D computer vision

The pipeline used to process the point cloud acquired from the 3D camera produces a consistent result. The segmentation and cluster extraction process does not produce any unwanted artifacts or noise, and we are consistently able to match the correct part of the point cloud with a model from the training set. The one drawback with the currently implemented 3D detection process is the fact that the detected orientation is somewhat ambiguous. We suspect that this is caused by the fact that both parts are symmetrically shaped and has few detectable features, but we can not conclusively say that this is the cause. The orientation ambiguity does not cause any issue since the final position and orientation of the part is found using 2D object detection.

Using more complex global descriptors might have improved the performance of the 3D matching, but as shown in section 4.2.2 this was not feasible for the specific parts used in this thesis. Exactly why the more complex global descriptors did not perform as expected is hard to conclusively say, but we believe it is caused by the fact that the object cluster extracted from the point cloud lacks a lot of detail compared to the model used in the training set. The more complex global descriptors works by segmenting the object cluster into multiple sections, and estimating a basic global descriptor for each segment. The test results show that the global descriptor for the object cluster and the training set model produce a different number of segments, and thus are incomparable. It is possible that using a different approach for creating the training sets would allow the use of more complex global descriptors. This was not tested since one of the prerequisites for this thesis is the use of 3D models for matching.

The level of accuracy achieved from the 3D point cloud matching is quite a bit higher than what was expected. The maximum measured deviation is just below 2cm, which is not at all close to the expected 0.4cm theoretical accuracy. We believe this is caused by two main reasons. First, the accuracy testing was performed by manually placing the parts on a measured grid. It is highly likely that the placement of the grid is not perfectly centered in the robotic cell. This will lead to measurement errors. In addition, the manual placement of the parts on the measured grid will also lead to some errors (the parts were placed as accurately as possible, but it is still hard to guarantee a perfect placement). Second, the *kinect2_bridge* ROS node used to acquire the 3D point cloud implements a bilateral filter (see section 2.3.3). This filter will attempt to remove holes and noise from the point cloud at the cost of point accuracy. It is not beyond reason to assume that using a 3D camera grabber that does not implement such a filter might eliminate some of the inaccuracies.

We intended the 3D object detection to be used as a rough estimate from the start of this project based on the problem description. Because of this, the low positional accuracy achieved using the 3D object detection process did not cause any issues in the final implementation of the system. The position obtained through the 3D object detection process is used as a rough estimate and a starting point for the 2D object detection. We found that the accuracy achieved by the 3D object detection process was adequate to perform this task.

5.3 Combining 2D and 3D computer vision

Given the premise of the task, and the expected result from the 3D object detection process, the Microsoft Kinect™ depth camera worked adequately. This thesis shows that given the right expectations, the use of such a low cost sensor can yield reliable and usable results. It is important to note that our results for the 3D object detection process did not show a level of accuracy that would have been usable for automated assembly on its own. That said, 3D object detection as used in our application works well as a mechanism to produce a rough position estimate for the detected objects quickly.

The combination of the 2D and the 3D camera is what made it possible to complete an automated assembly task with the parts used in this thesis. The 2D object detection provides the high level of positional and orientation accuracy necessary to ensure a successful assembly.

5.4 Hardware

The fact that the Microsoft Kinect™ depth sensor is not a sensor well suited for industrial applications is evident throughout this thesis. Low repeatability and accuracy limits the use cases for this sensor and dictates that it is to be used in conjunction with some other form of instrumentation (unless the task at hand does not demand high accuracy and repeatability). The high data acquisition rate makes this sensor ideal as a tool to quickly estimate a rough object pose. When utilized in this form, the sensor works well, as is shown throughout this thesis.

The Logitech C930e web camera used for 2D object detection performed adequately, as evident from the results. However, it is a consumer grade camera aimed at good quality video for e.g. Skype-conversations and not for industrial applications. The auto-focus can not be disabled, which in many cases blurred the image too much to be able to detect the objects. This was a problem during many assembly operations. In addition, there is no need for a camera with higher resolution than the C930e capable of 1920×1080 , since our object detection runs at 1280×720 to keep the frame rate at an acceptable level when using SIFT. If higher resolution and high frame rate is needed, the processing unit must be upgraded, or the code must be rewritten to support execution on a *graphical processing unit* (GPU). An alternative to the C930e would be a proper camera aimed at industrial applications, with high quality optics delivering sharp images with fixed focus. With that said, this thesis shows that a consumer grade camera like the C930e can yield 2D object detection results usable for robotic assembly.

The linear pneumatic gripper used to manipulate the parts for assembly performed as intended. The main issues caused by the gripper, was related to the finger extensions use (the parts mounted on the linear cylinder that actually makes contact with the parts). The finger extensions are made from 3D printed plastic, and is not designed with this specific assembly

task in mind. This led to some minor inconveniences during the assembly process. We noticed a slight shift in orientation (rotation about the z axis) of the part that was being gripped at the time of contact between the finger extensions and the part. Even though this change in orientation was consistent, and we were able to correct for the movement, this is not an ideal situation. A set of finger extensions designed for this specific task could potentially eliminate the orientation shift, and increase the success rate of the automated assembly.

The two robotic manipulators were controlled using the ROS framework. We did not encounter any issues with this solution, and the control tools were easily implemented in our software solution. This allowed us to control the robotic manipulators using code in a simple manner. The only drawback encountered using this system, is the lack of a *point to point* movement command. The only way to move the manipulators in the current version of the system is to use linear interpolated trajectories. This did not cause any problems directly, but a *point to point* option would be useful to achieve more efficient robotic movements. Such a *point to point* command option would be quite simple to implement using the *move_group_interface* API available through MoveIt!.

Chapter 6: Conclusion

This thesis presents the methods developed to solve a robotic assembly problem. The results presented in Chapter 4 show how these methods perform when applied to an actual assembly operation. As evident from the presented results, the assembly problem may be solved using the presented approach for 3D and 2D object detection. In fact, it can successfully detect two rather difficult objects with no clear geometric features, and assemble them with sufficient accuracy in terms of position and orientation.

The kinematic relationship between the physical components in the robotic cell is described using conventional kinematics, and is proven to be accurate to within a reasonable margin of error. The kinematic description is a key aspect for successful robotic assembly and is used in conjunction with the 2D and 3D computer vision systems to perform the assembly task.

Different aspects of the system developed to perform 2D object detection is investigated, and all decisions related to choice of implementation is anchored in results achieved through testing. The 2D detection implementation is shown to produce reliable results both in practical experiments and synthetic tests.

This thesis presents different approaches for 3D object object detection, and shows an implemented system working in a relevant use case. The different approaches are compared, and tests were performed in order to choose the best combination of tools used to perform the task. The implemented 3D object detection is shown to provide an excellent way of acquiring a rough position estimate.

By combining the 2D and 3D detection systems, the issue caused by the relatively low accuracy of the 3D object detection procedure is minimized. The results obtained from testing the full solution shows that such a detection system is viable in an industrial use case.

To conclude, the methods developed to perform automated assembly has shown, through practical experiments, to yield promising results for this type of automated assembly. The complete solution is shown to fulfill the problem description as provided in Chapter 1. However, there are still room for improvements. The main focus areas we feel could be improved is presented in the following section.

6.1 Future work

In terms of future work, the following section describes a few key points that we feel could be investigated to improve the performance of the robotic assembly.

Finger extensions for the gripper The current finger extensions for the pneumatic gripper is not designed or suited to perform assembly of the parts used in this thesis. This did lead to some inconsistencies and we believe that with finger extensions designed to grip the parts used in a repeatable manner would improve the reliability of the assembly setup.

Training set The training set used for 3D matching in this work was generated virtually using a 3D model of the part of interest. Throughout this thesis work this approach has proven to be a simple and reliable way to produce a training set. We believe that the issue of not being able to use the more complex CVFH and OUR-CVFH global descriptor over the basic VFH descriptor is related to the difference in detail produced by the virtual training set and the 3D camera. Because of this, investigating the viability of using a physically created training set could produce quite useful results.

Upgrading the 2D image sensor The 2D camera currently used in the robotic cell is a consumer grade web camera. Because of this, it lacks some features that simplify the object detection process quite a bit. Such as a fixed focus, and little to no radial distortion. The radial distortion is dealt with using a calibration and correction process, but using a camera where this step is unnecessary could simplify the 2D object detection pipeline. In addition, the auto focus feature of the current camera has a tendency to not select the correct object to focus on, which leads to increased processing time and longer detection time. This drawback can be fully negated by using an industrial grade camera with fixed focus. Despite this, we do not believe a higher quality camera would drastically increase the detection performance, but rather simplify and streamline the process.

Upgrading the 3D depth sensor The current 3D depth sensor does not provide the necessary spatial accuracy or resolution needed to fully detect an accurate object pose using the 3D camera alone. By replacing the current depth sensor with an industrial grade sensor, it is possible that the automated assembly could be done using only a 3D camera, which would drastically decrease the assembly time and system complexity needed. This is, in our opinion, the biggest factor limiting the current setup and the area of the system where the most improvement could be done.

Point to point motion The current robotic control system only allows for linearly interpolated trajectories. This does not cause any major problems for the assembly task. However, a *point to point* command is considered as a "nice to have" functionality. We believe that this could be implemented in the already functioning service-based system quite easily using the *move_group_interface* available through MoveIt!.

Bibliography

- Aldoma, A., Marton, Z. C., Tombari, F., Wohlkinger, W., Potthast, C., Zeisl, B., Rusu, R. B., Gedikli, S., and Vincze, M. (2012a). Tutorial: Point cloud library: Three-dimensional object recognition and 6 dof pose estimation. *IEEE Robotics*, 19(3):80–91.
- Aldoma, A., Tombari, F., Rusu, R. B., and Vincze, M. (2012b). *OUR-CVFH – Oriented, Unique and Repeatable Clustered Viewpoint Feature Histogram for Object Recognition and 6DOF Pose Estimation*, pages 113–122. Springer Berlin Heidelberg, Berlin, Heidelberg.
- Aldoma, A., Vincze, M., Blodow, N., Gossow, D., Gedikli, S., Rusu, R. B., and Bradski, G. (2011). Cad-model recognition and 6dof pose estimation using 3d cues. In *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on*, pages 585–592.
- Alexandre, L. (2012). 3d descriptors for object and category recognition: a comparative evaluation. In *IEEE International Conf. on Intelligent Robotic Systems - IROS*, volume Workshop on Color-Depth Camera Fusion in Robotics, pages 1–6.
- Arun, K. S., Huang, T. S., and Blostein, S. D. (1987). Least-squares fitting of two 3-d point sets. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-9(5):698–700.
- Bay, H., Ess, A., Tuytelaars, T., and Van Gool, L. (2008). Speeded-up robust features (surf). *Computer Vision and Image Understanding*, 110(3):346–359.
- Bay, H., Schmid, C., and Gool, L. v. (2006a). From wide-baseline point and line correspondences to 3d.
- Bay, H., Tuytelaars, T., and Van Gool, L. (2006b). *SURF: Speeded Up Robust Features*, pages 404–417. Springer Berlin Heidelberg, Berlin, Heidelberg.
- Calonder, M., Lepetit, V., Strecha, C., and Fua, P. (2010). *Computer Vision – ECCV 2010: 11th European Conference on Computer Vision, Heraklion, Crete, Greece, September 5-11, 2010, Proceedings, Part IV*, chapter BRIEF: Binary Robust Independent Elementary Features, pages 778–792. Springer Berlin Heidelberg, Berlin, Heidelberg.
- Cipolla, R., Battiato, S., and Farinella, G. M. (2010). *Computer vision : detection, recognition and reconstruction*, volume 285 of *Studies in computational intelligence*. Springer, Berlin.
- Corke, P. (2013). *Robotics - Vision and Control*. Springer-Verlag.
- dictionary.com (2016). Kinematics definition. <http://www.dictionary.com/browse/kinematics>. [Online; accessed 28-May-2016].
- Fan, B., Wang, Z., and Wu, F. (2015). *Local Image Descriptor: Modern Approaches*. Springer-Briefs in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, Berlin, Heidelberg.

- Filipe, S. and Alexandre, L. A. (2014). A comparative evaluation of 3d keypoint detectors in a RGB-D object dataset. In *VISAPP 2014 - Proceedings of the 9th International Conference on Computer Vision Theory and Applications, Volume 1, Lisbon, Portugal, 5-8 January, 2014*, pages 476–483.
- Forsyth, D. and Ponce, J. (2003). *Computer vision : a modern approach*. An Alan R. Apt book. Prentice Hall, Upper Saddle River, N.J.
- Frome, A., Huber, D., Kolluri, R., Bülow, T., and Malik, J. (2004). *Computer Vision - ECCV 2004: 8th European Conference on Computer Vision, Prague, Czech Republic, May 11-14, 2004. Proceedings, Part III*, chapter Recognizing Objects in Range Data Using Regional Point Descriptors, pages 224–237. Springer Berlin Heidelberg, Berlin, Heidelberg.
- Gaskell, G. (2014). 3 ½ minutes could save you thousands on 3d scanning technology. <http://wenzel-metrology-equipment.blogspot.no/2014/12/3-minutes-could-save-you-thousands-on.html>. Accessed: 2016-05-09.
- GmbH, K. R. (2016). Kr 6 r900 sixx (kr agilus). http://www.kuka-robotics.com/en/products/industrial_robots/small_robots/kr6_r900_sixx/. [Online; accessed 28-May-2016].
- Harris, C. and Stephens, M. (1988). A combined corner and edge detector. In *In Proc. of Fourth Alvey Vision Conference*, pages 147–151.
- Johnson, A. (1997). *Spin-Images: A Representation for 3-D Surface Matching*. PhD thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA.
- Kadambi, A., Bhandari, A., and Raskar, R. (2014). *Computer Vision and Machine Learning with RGB-D Sensors*, chapter 3D Depth Cameras in Vision: Benefits and Limitations of the Hardware, pages 3–26. Springer International Publishing, Cham.
- Khoshelham, K. and Elberink, S. O. (2012). Accuracy and resolution of kinect depth data for indoor mapping applications. *Sensors*, 12(2):1437.
- Larsen, K. and Bjørkedal, A. (2016a). agilus_master_project. https://github.com/kristofferlarsen/agilus_master_project. [Online; accessed 28-May-2016].
- Larsen, K. and Bjørkedal, A. (2016b). Robotic assembly using 3d and 2d computer vision - demonstration video. <https://youtu.be/yR37s1aHcG0>. [Online; accessed 08-June-2016].
- Leutenegger, S., Chli, M., and Siegwart, R. Y. (2011). Brisk: Binary robust invariant scalable keypoints. In *2011 International Conference on Computer Vision*, pages 2548–2555.
- Liebhhardt, M. (2016). ar_track_alvar documentation. http://wiki.ros.org/ar_track_alvar. [Online; accessed 28-May-2016].
- Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110.
- Mair, E., Burschka, D., Hager, G. D., Suppa, M., and Hirzinger, G. (2010). Adaptive and generic corner detection based on the accelerated segment test.

- Marton, Z. C., Pangercic, D., Blodow, N., and Beetz, M. (2011). Combined 2d–3d categorization and classification for multimodal perception systems. *The International Journal of Robotics Research*.
- Mihelich, P. and Bowman, J. (2010). cv_bridge. http://wiki.ros.org/cv_bridge. [Online; accessed 08-February-2016].
- Nixon, M. and Aguado, A. S. (2012). *Feature Extraction & Image Processing for Computer Vision*. Feature Extraction & Image Processing. Elsevier Science, Burlington, 3rd ed. edition.
- OpenCV (2015a). Feature matching. http://docs.opencv.org/3.1.0/dc/dc3/tutorial_py_matcher.html#gsc.tab=0. Accessed: 2016-05-20.
- OpenCV (2015b). Opencv 3.1.0 documentation. <http://docs.opencv.org/3.1.0/#gsc.tab=0>. [Online; accessed 31-May-2016].
- OpenCV (2015c). Opencv github. <https://github.com/Itseez/opencv/blob/master/samples/cpp/calibration.cpp>. [Online; accessed 31-May-2016].
- PCL. Cluster recognition and 6dof pose estimation using vfh descriptors. http://pointclouds.org/documentation/tutorials/vfh_recognition.php. Accessed: 2016-05-10.
- PCL. Estimating surface normals in a pointcloud. http://pointclouds.org/documentation/tutorials/normal_estimation.php. Accessed: 2016-05-09.
- PCL. Euclidean cluster extraction. http://www.pointclouds.org/documentation/tutorials/cluster_extraction.php. Accessed: 2016-05-09.
- PCL. Pcl :: Registration. <http://www.pointclouds.org/assets/iros2011/registration.pdf>. Accessed: 2016-05-10.
- PCL. Removing outliers using a statisticaloutlierremoval filter. http://www.pointclouds.org/documentation/tutorials/statistical_outlier.php#statistical-outlier-removal. Accessed: 2016-05-09.
- PCL (2016). Point cloud library. <http://pointclouds.org/>. [Online; accessed 28-May-2016].
- ROBOTICA, G. D. (2015). Raytracing. [http://robotica.unileon.es/mediawiki/index.php/PCL/OpenNI_tutorial_5:_3D_object_recognition_\(pipeline\)#Raytracing](http://robotica.unileon.es/mediawiki/index.php/PCL/OpenNI_tutorial_5:_3D_object_recognition_(pipeline)#Raytracing). Accessed: 2016-05-10.
- Rosin, P. L. (1999). Measuring corner properties. *Computer Vision and Image Understanding*, 73(2):291–307.
- ROS.org (2016a). Ros documentation. <http://wiki.ros.org/>. [Online; accessed 28-May-2016].
- ROS.org (2016b). sensor_msgs/pointcloud2 message. http://docs.ros.org/api/sensor_msgs/html/msg/PointCloud2.html. [Online; accessed 28-May-2016].
- Rosten, E. and Drummond, T. (2006). Machine learning for high-speed corner detection. *Computer Vision - Eccv 2006 , Pt 1, Proceedings*, 3951:430–443.

- Rublee, E., Rabaud, V., Konolige, K., and Bradski, G. (2011). Orb: An efficient alternative to sift or surf. In *2011 International Conference on Computer Vision*, pages 2564–2571.
- Rusu, R. B. (2009). *Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments*. PhD thesis, Technische Universität München.
- Rusu, R. B., Blodow, N., and Beetz, M. (2009). Fast point feature histograms (fpfh) for 3d registration. In *Robotics and Automation, 2009. ICRA '09. IEEE International Conference on*, pages 3212–3217.
- Rusu, R. B., Bradski, G., Thibaux, R., and Hsu, J. (2010). Fast 3d recognition and pose using the viewpoint feature histogram. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 2155–2162.
- Rusu, R. B. and Cousins, S. (2011). 3d is here: Point cloud library (pcl). In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 1–4.
- Shapiro, L. G. and Stockman, G. C. (2001). *Computer vision*. Prentice Hall, Upper Saddle River, N.J.
- Siciliano, B., Sciavicco, L., Villani, L., and Oriolo, G. (2010). *Robotics - Modelling, Planning and Control*. Springer-Verlag.
- Smith, S. M. and Brady, J. M. (1997). Susan—a new approach to low level image processing. *International Journal of Computer Vision*, 23(1):45–78.
- Sucan, I. A. and Chitta, S. (2013). Move group interface/c++ api. http://docs.ros.org/indigo/api/pr2_moveit_tutorials/html/planning/src/doc/move_group_interface_tutorial.html. [Online; accessed 21-March-2016].
- Sucan, I. A. and Chitta, S. (2016). Moveit! <http://moveit.ros.org/>. [Online; accessed 01-February-2016].
- The MathWorks, I. (2016). What is camera calibration? <http://se.mathworks.com/help/vision/ug/camera-calibration.html>. [Online; accessed 16-February-2016].
- Tombari, F., Salti, S., and Di Stefano, L. (2010a). *Computer Vision – ECCV 2010: 11th European Conference on Computer Vision, Heraklion, Crete, Greece, September 5-11, 2010, Proceedings, Part III*, chapter Unique Signatures of Histograms for Local Surface Description, pages 356–369. Springer Berlin Heidelberg, Berlin, Heidelberg.
- Tombari, F., Salti, S., and Di Stefano, L. (2010b). Unique shape context for 3d data description. In *Proceedings of the ACM Workshop on 3D Object Retrieval, 3DOR '10*, pages 57–62, New York, NY, USA. ACM.
- Wiedemeyer, T. (2016). Kinect2 bridge. https://github.com/code-iai/iai_kinect2/tree/master/kinect2_bridge. [Online; accessed 28-May-2016].
- Wikipedia (2015). Programmable logic controller — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Programmable_logic_controller. [Online; accessed 24-November-2015].

- Wohlkinger, W. and Vincze, M. (2011). Ensemble of shape functions for 3d object classification. In *Robotics and Biomimetics (ROBIO), 2011 IEEE International Conference on*, pages 2987–2992.
- Xiang, L., Echtler, F., Kerl, C., Wiedemeyer, T., Lars, hanyazou, Gordon, R., Facioni, F., laborer2008, Wareham, R., Goldhoorn, M., alberth, gaborpapp, Fuchs, S., jmtatsch, Blake, J., Federico, Jungkurth, H., Mingze, Y., vinouz, Coleman, D., Burns, B., Rawat, R., Mokhov, S., Reynolds, P., Viau, P., Fraissinet-Tachet, M., Ludique, Billingham, J., and Alistair (2016). libfreenect2: Release 0.2.
- Zhong, Y. (2009). Intrinsic shape signatures: A shape descriptor for 3d object recognition. In *Computer Vision Workshops (ICCV Workshops), 2009 IEEE 12th International Conference on*, pages 689–696.

Appendix

Appendix A: Software Tools Created

This appendix contain source code for the two toolbox classes created for this thesis. These toolboxes are used in both the *agilus_master_project* and *qt_filter_tester* applications (full source code available in this documents digital appendix). The following is a short description of the two classes:

pcl_filters.cpp - This class handles all 3D point cloud processing. It includes functionality to perform the most common and useful filtering tasks. In addition, functionality to perform 3D object detection by matching a 3D point cloud acquired through a 3D depth sensor to a training set generated from a 3D CAD model is also available.

pcl_filters.hpp - This is the *Header* file for the *pcl_filters.cpp* class. This file defines the content of the .cpp file.

modelloader.cpp - This class is used to generate, and load, a training set based on a 3D CAD model. As mentioned in section 4.2.4, the original code for this class was written by Adam Leon Kleppe. This class has been modified to allow feature estimation to be a part of the training set.

modelloader.hpp - This is the *Header* file for the *modelloader.cpp* class. This file defines the content of the .cpp file.

pcl_filters.cpp

Listing A.1: Source file - *agilus_master_project/pcl_filters.cpp*

```
1 //
2 // Original author Kristoffer Larsen. Latest change date 01.05.2016
3 // pcl_filters.cpp is a toolbox that implements the most commonly used features
4 // from PCL.
5 // In this application, pcl_filters.cpp is used for 3D point cloud processing.
6 // Created as part of the software solution for a Master's Thesis in Production
7 // Technology at NTNU Trondheim.
8 //
9 #include "../include/agilus_master_project/pcl_filters.hpp"
10
11 namespace agilus_master_project{
12
13 PclFilters::PclFilters(QObject *parent):
14     QObject(parent){}
15
16 PclFilters::~PclFilters() {}
17
18 int PclFilters::search_for_model(std::vector<RayTraceCloud> clusters, std::
19     vector<RayTraceCloud> model){
```



```

18     pcl::KdTreeFLANN<pcl::VFHSignature308>::Ptr search_tree =
        generate_search_tree(model);
19     float min_distance = 10000;
20     int correct_cluster;
21     std::vector<float> search_result;
22     for (int i = 0; i < clusters.size(); i++){
23         search_result = match_cloud(clusters.at(i), search_tree);
24         if(search_result[1] < min_distance){
25             min_distance = search_result[1];
26             correct_cluster = i;
27         }
28     }
29     return correct_cluster;
30 }
31
32 void PclFilters::ransac_recognition(std::vector<RayTraceCloud> models,
    RayTraceCloud object){
33     pcl::recognition::ObjRecRANSAC recognition(40.0,5.0);
34     std::list<pcl::recognition::ObjRecRANSAC::Output> matchingList;
35     for(int i = 0; i< models.size(); i++){
36         QString name = "model_";
37         name.append(QString::number(i));
38         recognition.addModel(*models.at(i).cloud,*models.at(i).normals,name.
            toStdString());
39     }
40     recognition.recognize(*models.at(0).cloud,*models.at(0).normals,
        matchingList,0.99);
41 }
42
43 Eigen::Matrix4f PclFilters::calculateInitialAlignment(RayTraceCloud source,
    RayTraceCloud target, float min_sample_distance, float
    max_correspondence_distance, int nr_iterations){
44     pcl::SampleConsensusInitialAlignment<pcl::PointXYZ,pcl::PointXYZ,pcl::
        FPFHSignature33> sac_ia;
45     sac_ia.setMinSampleDistance(min_sample_distance);
46     sac_ia.setMaxCorrespondenceDistance(max_correspondence_distance);
47     sac_ia.setMaximumIterations(nr_iterations);
48     sac_ia.setInputSource(source.keypoints);
49     sac_ia.setSourceFeatures(source.local_descriptors);
50     sac_ia.setInputTarget(target.keypoints);
51     sac_ia.setTargetFeatures(target.local_descriptors);
52     pcl::PointCloud<pcl::PointXYZ> registration_output;
53     sac_ia.align(registration_output);
54     return (sac_ia.getFinalTransformation());
55 }
56
57 Eigen::Matrix4f PclFilters::calculateRefinedAlignment(RayTraceCloud source,
    RayTraceCloud target, Eigen::Matrix4f initial_alignment, float
    max_correspondence_distance, float outlier_rejection_threshold, float
    transformation_epsilon, float euclidian_fitness_epsilon, int max_iterations
    ){
58     pcl::IterativeClosestPoint<pcl::PointXYZ,pcl::PointXYZ> icp;
59     icp.setMaxCorrespondenceDistance(max_correspondence_distance);
60     icp.setRANSACOutlierRejectionThreshold(outlier_rejection_threshold);
61     icp.setTransformationEpsilon(transformation_epsilon);
62     icp.setEuclideanFitnessEpsilon(euclidian_fitness_epsilon);
63     icp.setMaximumIterations(max_iterations);

```

```

64     pcl::PointCloud<pcl::PointXYZ>::Ptr source_transformed (new pcl::PointCloud
        <pcl::PointXYZ>);
65     pcl::transformPointCloud(*source.cloud,*source_transformed,
        initial_alignment);
66     icp.setInputSource(source_transformed);
67     icp.setInputTarget(target.cloud);
68     pcl::PointCloud<pcl::PointXYZ> registration_output;
69     icp.align(registration_output);
70     return (icp.getFinalTransformation() * initial_alignment);
71 }
72
73 RayTraceCloud PclFilters::calculate_features(RayTraceCloud inputcloud){
74     inputcloud.normals = get_normals(inputcloud.cloud,0.05);
75     inputcloud.keypoints = calculate_keypoints(inputcloud.cloud,0.001,3,3,0.0);
76     inputcloud.local_descriptors = calculate_local_descriptor(inputcloud.cloud,
        inputcloud.normals,inputcloud.keypoints,0.15);
77     inputcloud.global_descriptors = calculate_vfh_descriptors(inputcloud.cloud,
        inputcloud.normals);
78     return (inputcloud);
79 }
80
81 pcl::KdTreeFLANN<pcl::VFHSignature308>::Ptr PclFilters::generate_search_tree(
        std::vector<RayTraceCloud> models){
82     pcl::PointCloud<pcl::VFHSignature308>::Ptr global_descriptor(new pcl::
        PointCloud<pcl::VFHSignature308>);
83     pcl::KdTreeFLANN<pcl::VFHSignature308>::Ptr search_tree(new pcl::
        KdTreeFLANN<pcl::VFHSignature308>);
84     for(int i = 0; i< models.size(); i++){
85
86         RayTraceCloud model = models.at(i);
87         *global_descriptor += *(model.global_descriptors);
88     }
89     search_tree->setInputCloud(global_descriptor);
90     return (search_tree);
91 }
92
93 std::vector<float> PclFilters::match_cloud(RayTraceCloud object_model, pcl::
        KdTreeFLANN<pcl::VFHSignature308>::Ptr search_tree){
94     std::vector<float> returnvalues;
95     std::vector<int> best_match(1);
96     std::vector<float> square_distance(1);
97     search_tree->nearestKSearch (object_model.global_descriptors->points[0],1,
        best_match,square_distance);
98     returnvalues.push_back(best_match[0]);
99     returnvalues.push_back(square_distance[0]);
100    return (returnvalues);
101 }
102
103 std::vector<float> PclFilters::temp_matching_cvfh(RayTraceCloud object_model,
        pcl::KdTreeFLANN<pcl::VFHSignature308>::Ptr search_tree){
104     std::vector<float> returnvalues;
105     std::vector<int> best_match(1);
106     std::vector<float> square_distance(1);
107     int nr_of_descriptors = object_model.global_descriptors->points.size();
108     for(int i = 0; i< nr_of_descriptors; i++){
109         search_tree->nearestKSearch (object_model.global_descriptors->points[i
        ],1,best_match,square_distance);

```

```

110         std::cout << "loop nr: " << i << ", best match: " << best_match.at(0)
111             << ", confidence level: " << square_distance.at(0) << std::endl;
112     }
113     returnvalues.push_back(best_match[0]);
114     returnvalues.push_back(square_distance[0]);
115     return (returnvalues);
116 }
117 pcl::PointCloud<pcl::PointXYZ>::Ptr PclFilters::calculate_keypoints(pcl::
118     PointCloud<pcl::PointXYZ>::Ptr cloud, float min_scale, int nr_octaves, int
119     nr_scales_per_octave, float min_contrast){
120     pcl::PointCloud<pcl::PointXYZRGB>::Ptr rgbcloud(new pcl::PointCloud<pcl::
121         PointXYZRGB>);
122     pcl::copyPointCloud(*cloud,*rgbcloud);
123     for(int i = 0; i< rgbcloud->size(); i++){
124         rgbcloud->points[i].r = 255;
125         rgbcloud->points[i].g = 255;
126         rgbcloud->points[i].b = 255;
127     }
128     pcl::SIFTKeypoint<pcl::PointXYZRGB, pcl::PointWithScale> sift_detect;
129     sift_detect.setSearchMethod (pcl::search::Search<pcl::PointXYZRGB>::Ptr (
130         new pcl::search::KdTree<pcl::PointXYZRGB>));
131     sift_detect.setScales (min_scale, nr_octaves, nr_scales_per_octave);
132     sift_detect.setMinimumContrast (min_contrast);
133     sift_detect.setInputCloud (rgbcloud);
134     pcl::PointCloud<pcl::PointWithScale> keypoints_temp;
135     sift_detect.compute (keypoints_temp);
136     pcl::PointCloud<pcl::PointXYZ>::Ptr keypoints (new pcl::PointCloud<pcl::
137         PointXYZ>);
138     pcl::copyPointCloud (keypoints_temp, *keypoints);
139     return (keypoints);
140 }
141
142 pcl::PointCloud<pcl::FPFHSignature33>::Ptr PclFilters::
143     calculate_local_descriptor(pcl::PointCloud<pcl::PointXYZ>::Ptr cloud, pcl::
144     PointCloud<pcl::Normal>::Ptr normal, pcl::PointCloud<pcl::PointXYZ>::Ptr
145     keypoints, float feature_radius){
146     pcl::FPFHEstimationOMP<pcl::PointXYZ, pcl::Normal, pcl::FPFHSignature33>
147         fpfh_estimation;
148     fpfh_estimation.setNumberOfThreads(8);
149     fpfh_estimation.setSearchMethod (pcl::search::Search<pcl::PointXYZ>::Ptr (
150         new pcl::search::KdTree<pcl::PointXYZ>));
151     fpfh_estimation.setRadiusSearch (feature_radius);
152     fpfh_estimation setSearchSurface (cloud);
153     fpfh_estimation.setInputNormals (normal);
154     fpfh_estimation.setInputCloud (keypoints);
155     pcl::PointCloud<pcl::FPFHSignature33>::Ptr local_descriptors (new pcl::
156         PointCloud<pcl::FPFHSignature33>);
157     fpfh_estimation.compute (*local_descriptors);
158     return (local_descriptors);
159 }
160
161 boost::shared_ptr<pcl::visualization::PCLVisualizer> PclFilters::visualize(
162     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud){
163     viewer.reset(new pcl::visualization::PCLVisualizer ("3D Viewer",false));
164     viewer->addPointCloud<pcl::PointXYZ> (cloud, "sample cloud");
165     viewer->initCameraParameters ();

```

```

154     return (viewer);
155 }
156
157 boost::shared_ptr<pcl::visualization::PCLVisualizer> PclFilters::visualize_rgb(
    pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud){
158     viewer.reset(new pcl::visualization::PCLVisualizer ("3D Viewer",false));
159     pcl::visualization::PointCloudColorHandlerRGBField<pcl::PointXYZRGB> rgb(
        cloud);
160     viewer->addPointCloud<pcl::PointXYZRGB> (cloud, rgb, "sample cloud");
161     viewer->initCameraParameters ();
162     return (viewer);
163 }
164
165 boost::shared_ptr<pcl::visualization::PCLVisualizer> PclFilters::
    visualize_normals(pcl::PointCloud<pcl::PointXYZ>::Ptr cloud, double radius,
        int numOfNormals){
166     viewer.reset(new pcl::visualization::PCLVisualizer ("3D Viewer",false));
167     viewer->addPointCloud<pcl::PointXYZ> (cloud, "sample cloud");
168     viewer->addPointCloudNormals<pcl::PointXYZ, pcl::Normal> (cloud,
        get_normals(cloud,radius), numOfNormals, 0.05, "normals");
169     viewer->setPointCloudRenderingProperties (pcl::visualization::
        PCL_VISUALIZER_COLOR, 1.0, 0.0, 0.0, "normals");
170     viewer->initCameraParameters ();
171     filteredCloud = cloud;
172     return (viewer);
173 }
174
175 std::vector<pcl::PointCloud<pcl::PointXYZ>::Ptr> PclFilters::cluster_extraction
    (pcl::PointCloud<pcl::PointXYZ>::Ptr cloud, double distance){
176     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_f (new pcl::PointCloud<pcl::
        PointXYZ>), incloud (new pcl::PointCloud<pcl::PointXYZ>);
177     pcl::copyPointCloud(*cloud,*incloud);
178     pcl::SACSegmentation<pcl::PointXYZ> seg;
179     pcl::PointIndices::Ptr inliers (new pcl::PointIndices);
180     pcl::ModelCoefficients::Ptr coefficients (new pcl::ModelCoefficients);
181     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_plane (new pcl::PointCloud<pcl::
        PointXYZ> ());
182     seg.setOptimizeCoefficients (true);
183     seg.setModelType (pcl::SACMODEL_PLANE);
184     seg.setMethodType (pcl::SAC_RANSAC);
185     seg.setMaxIterations (100);
186     seg.setDistanceThreshold (distance);
187     seg.setInputCloud (incloud);
188     seg.segment (*inliers, *coefficients);
189
190     pcl::ExtractIndices<pcl::PointXYZ> extract;
191     extract.setInputCloud (incloud);
192     extract.setIndices (inliers);
193     extract.setNegative (false);
194     extract.filter (*cloud_plane);
195     extract.setNegative (true);
196     extract.filter (*cloud_f);
197     *incloud = *cloud_f;
198
199     pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new pcl::search::KdTree<pcl::
        PointXYZ>);
200     tree->setInputCloud (incloud);

```

```

201     std::vector<pcl::PointIndices> cluster_indices;
202     pcl::EuclideanClusterExtraction<pcl::PointXYZ> ec;
203     ec.setClusterTolerance (0.01); // 1cm
204     ec.setMinClusterSize (300);
205     ec.setMaxClusterSize (25000);
206     ec.setSearchMethod (tree);
207     ec.setInputCloud (incloud);
208     ec.extract (cluster_indices);
209     std::vector<pcl::PointCloud<pcl::PointXYZ>::Ptr> clusters;
210     for(int i = 0; i < cluster_indices.size(); i++){
211         pcl::PointCloud<pcl::PointXYZ>::Ptr tmpcloud (new pcl::PointCloud<pcl::
                PointXYZ>);
212         pcl::copyPointCloud(*incloud, cluster_indices[i], *tmpcloud);
213         clusters.push_back(tmpcloud);
214     }
215     filteredCloud = combine_clouds(clusters);
216     return (clusters);
217 }
218
219 pcl::PointCloud<pcl::PointXYZ>::Ptr PclFilters::plane_segmentation(pcl::
    PointCloud<pcl::PointXYZ>::Ptr cloud, double distance){
220     pcl::PointCloud<pcl::PointXYZ>::Ptr incloud (new pcl::PointCloud<pcl::
        PointXYZ>);
221     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_plane (new pcl::PointCloud<pcl::
        PointXYZ>);
222     pcl::copyPointCloud(*cloud, *incloud);
223     pcl::ModelCoefficients::Ptr coefficients (new pcl::ModelCoefficients);
224     pcl::PointIndices::Ptr inliers (new pcl::PointIndices);
225     pcl::SACSegmentation<pcl::PointXYZ> seg;
226
227     seg.setOptimizeCoefficients (true);
228     seg.setModelType (pcl::SACMODEL_PLANE);
229     seg.setMethodType (pcl::SAC_RANSAC);
230     seg.setDistanceThreshold (distance);
231     seg.setInputCloud (cloud);
232     seg.segment (*inliers, *coefficients);
233
234     pcl::ExtractIndices<pcl::PointXYZ> extract;
235     extract.setInputCloud (incloud);
236     extract.setIndices (inliers);
237     extract.setNegative (false);
238     extract.filter (*cloud_plane);
239     filteredCloud = cloud;
240     return (cloud_plane);
241 }
242
243 pcl::PointCloud<pcl::PointXYZ>::Ptr PclFilters::get_filtered_cloud(){
244     return filteredCloud;
245 }
246
247 pcl::PointCloud<pcl::PointXYZRGB>::Ptr PclFilters::color_cloud(pcl::PointCloud<
    pcl::PointXYZ>::Ptr cloud, int r, int g, int b){
248     pcl::PointCloud<pcl::PointXYZRGB>::Ptr rgb_cloud(new pcl::PointCloud<pcl::
        PointXYZRGB>);
249     pcl::copyPointCloud(*cloud, *rgb_cloud);
250     for (int i = 0; i < rgb_cloud->points.size(); i++)
251     {

```

```

252     rgb_cloud->points[i].r = r;
253     rgb_cloud->points[i].g = g;
254     rgb_cloud->points[i].b = b;
255 }
256 return (rgb_cloud);
257 }
258
259 pcl::PointCloud<pcl::Normal>::Ptr PclFilters::get_normals(pcl::PointCloud<pcl::
PointXYZ>::Ptr cloud, double radius){
260     pcl::PointCloud<pcl::Normal>::Ptr normals_out (new pcl::PointCloud<pcl::
Normal>);
261     pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new pcl::search::KdTree<pcl::
PointXYZ> ());
262     pcl::NormalEstimationOMP<pcl::PointXYZ, pcl::Normal> norm_est;
263     norm_est.setNumberOfThreads(8);
264     norm_est.setSearchMethod(tree);
265     norm_est.setRadiusSearch (radius);
266     norm_est.setInputCloud (cloud);
267     norm_est.compute (*normals_out);
268     return (normals_out);
269 }
270
271 pcl::PointCloud<pcl::PointXYZ>::Ptr PclFilters::passthrough_filter(pcl::
PointCloud<pcl::PointXYZ>::Ptr cloud, double min, double max, std::string
axis){
272     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_filtered (new pcl::PointCloud<
pcl::PointXYZ>);
273     passfilter.setKeepOrganized(true);
274     passfilter.setInputCloud(cloud);
275     passfilter.setFilterFieldName(axis);
276     passfilter.setFilterLimits(min,max);
277     passfilter.filter(*cloud_filtered);
278     filteredCloud = cloud_filtered;
279     return (cloud_filtered);
280 }
281
282 pcl::PointCloud<pcl::PointXYZ>::Ptr PclFilters::voxel_grid_filter(pcl::
PointCloud<pcl::PointXYZ>::Ptr cloud, double lx, double ly, double lz){
283     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_filtered (new pcl::PointCloud<
pcl::PointXYZ>);
284     voxelfilter.setInputCloud(cloud);
285     voxelfilter.setLeafSize(lx,ly,lz);
286     voxelfilter.filter(*cloud_filtered);
287     filteredCloud = cloud_filtered;
288     return (cloud_filtered);
289 }
290
291 pcl::PointCloud<pcl::PointXYZ>::Ptr PclFilters::median_filter(pcl::PointCloud<
pcl::PointXYZ>::Ptr cloud, int window_size, double max_allowed_movement){
292     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_filtered (new pcl::PointCloud<
pcl::PointXYZ>);
293     medianfilter.setInputCloud(cloud);
294     medianfilter.setWindowSize(window_size);
295     medianfilter.setMaxAllowedMovement(max_allowed_movement);
296     medianfilter.filter(*cloud_filtered);
297     filteredCloud = cloud_filtered;
298     return (cloud_filtered);

```

```

299 }
300
301 pcl::PointCloud<pcl::PointXYZ>::Ptr PclFilters::shadowpoint_removal_filter(
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud, double threshold, double radius)
    {
302     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_filtered (new pcl::PointCloud<
        pcl::PointXYZ>);
303     shadowpoint_filter.setInputCloud(cloud);
304     shadowpoint_filter.setKeepOrganized(true);
305     shadowpoint_filter.setThreshold(threshold);
306     shadowpoint_filter.setNormals(get_normals(cloud, radius));
307     shadowpoint_filter.filter(*cloud_filtered);
308     filteredCloud = cloud_filtered;
309     return (cloud_filtered);
310 }
311
312 pcl::PointCloud<pcl::PointXYZ>::Ptr PclFilters::statistical_outlier_filter(
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud, int meanK, double
    std_deviation_threshold){
313     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_filtered (new pcl::PointCloud<
        pcl::PointXYZ>);
314     statistical_outlier.setKeepOrganized(true);
315     statistical_outlier.setInputCloud(cloud);
316     statistical_outlier.setMeanK(meanK);
317     statistical_outlier.setStddevMulThresh(std_deviation_threshold);
318     statistical_outlier.filter(*cloud_filtered);
319     filteredCloud = cloud_filtered;
320     return (cloud_filtered);
321 }
322
323 pcl::PointCloud<pcl::PointXYZ>::Ptr PclFilters::combine_clouds(std::vector<
    pcl::PointCloud<pcl::PointXYZ>::Ptr> input){
324     pcl::PointCloud<pcl::PointXYZ>::Ptr cluster_cloud (new pcl::PointCloud<
        pcl::PointXYZ>);
325     *cluster_cloud = *input.at(0);
326     for (unsigned i=0; i<input.size(); i++){
327         *cluster_cloud += *(pcl::PointCloud<pcl::PointXYZ>::Ptr) input.at(i);
328     }
329     return (cluster_cloud);
330 }
331
332 pcl::PointCloud<pcl::VFHSignature308>::Ptr PclFilters::
    calculate_cvfh_descriptors(pcl::PointCloud<pcl::PointXYZ>::Ptr cloud){
333     pcl::PointCloud<pcl::VFHSignature308>::Ptr descriptors(new pcl::PointCloud<
        pcl::VFHSignature308>);
334     pcl::search::KdTree<pcl::PointXYZ>::Ptr kdtree(new pcl::search::KdTree<
        pcl::PointXYZ>);
335     pcl::PointCloud<pcl::Normal>::Ptr normals = get_normals(cloud,0.01);
336
337     pcl::CVFHEstimation<pcl::PointXYZ, pcl::Normal, pcl::VFHSignature308> cvfh;
338     cvfh.setInputCloud(cloud);
339     cvfh.setInputNormals(normals);
340     cvfh.setSearchMethod(kdtree);
341     cvfh.setEPSAngleThreshold(5.0 / 180.0 * M_PI);
342     cvfh.setCurvatureThreshold(1.0);
343     cvfh.setNormalizeBins(false);
344     cvfh.compute(*descriptors);

```

```

345     return (descriptors);
346 }
347
348 pcl::PointCloud<pcl::PointXYZ>::Ptr PclFilters::bilateral_filter(pcl::
    PointCloud<pcl::PointXYZ>::Ptr cloud, double sigmaS, double sigmaR){
349     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_filtered (new pcl::PointCloud<
        pcl::PointXYZ>);
350     pcl::FastBilateralFilterOMP<pcl::PointXYZ> bifilter;
351     bifilter.setInputCloud(cloud);
352     bifilter.setSigmaR(sigmaR);
353     bifilter.setSigmaS(sigmaS);
354     bifilter.filter(*cloud_filtered);
355     filteredCloud = cloud_filtered;
356     return (cloud_filtered);
357 }
358
359 pcl::PointCloud<pcl::ESFSignature640>::Ptr PclFilters::
    calculate_esf_descriptors(pcl::PointCloud<pcl::PointXYZ>::Ptr cloud){
360     pcl::PointCloud<pcl::ESFSignature640>::Ptr descriptor(new pcl::PointCloud<
        pcl::ESFSignature640>);
361     pcl::ESFEstimation<pcl::PointXYZ, pcl::ESFSignature640> esf;
362     esf.setInputCloud(cloud);
363     esf.compute(*descriptor);
364     return (descriptor);
365 }
366
367 pcl::PointCloud<pcl::VFHSignature308>::Ptr PclFilters::
    calculate_ourcvfh_descriptors(pcl::PointCloud<pcl::PointXYZ>::Ptr cloud,
    pcl::PointCloud<pcl::Normal>::Ptr normal){
368     pcl::PointCloud<pcl::VFHSignature308>::Ptr descriptors(new pcl::PointCloud<
        pcl::VFHSignature308>);
369     pcl::search::KdTree<pcl::PointXYZ>::Ptr kdtree(new pcl::search::KdTree<
        pcl::PointXYZ>);
370     pcl::OURCVFHEstimation<pcl::PointXYZ, pcl::Normal, pcl::VFHSignature308>
        ourcvfh;
371     ourcvfh.setInputCloud(cloud);
372     ourcvfh.setInputNormals(normal);
373     ourcvfh.setSearchMethod(kdtree);
374     ourcvfh.setEPSAngleThreshold(5.0 / 180.0 * M_PI);
375     ourcvfh.setCurvatureThreshold(0.1);
376     ourcvfh.setNormalizeBins(false);
377     ourcvfh.setAxisRatio(0.8);
378     ourcvfh.compute(*descriptors);
379     return (descriptors);
380 }
381
382 pcl::PointCloud<pcl::GFPFHSignature16>::Ptr PclFilters::
    calculate_gfpfh_descriptors(pcl::PointCloud<pcl::PointXYZ>::Ptr cloud){
383     pcl::PointCloud<pcl::PointXYZL>::Ptr object(new pcl::PointCloud<pcl::
        PointXYZL>);
384     pcl::PointCloud<pcl::GFPFHSignature16>::Ptr descriptor(new pcl::PointCloud<
        pcl::GFPFHSignature16>);
385     pcl::copyPointCloud(*cloud,*object);
386     for (size_t i = 0; i < object->points.size(); ++i)
387     {
388         object->points[i].label = 1 + i % 4;
389     }

```



```

390     pcl::GFPFHEstimation<pcl::PointXYZL, pcl::PointXYZL, pcl::GFPFHSignature16>
        gfpfh;
391     gfpfh.setInputCloud(object);
392     gfpfh.setInputLabels(object);
393     gfpfh.setOctreeLeafSize(0.01);
394     gfpfh.setNumberOfClasses(4);
395     gfpfh.compute(*descriptor);
396     return (descriptor);
397 }
398
399 pcl::PointCloud<pcl::VFHSignature308>::Ptr PclFilters::
    calculate_vfh_descriptors(pcl::PointCloud<pcl::PointXYZ>::Ptr points, pcl::
    PointCloud<pcl::Normal>::Ptr normals){
400     pcl::VFHEstimation<pcl::PointXYZ, pcl::Normal, pcl::VFHSignature308>
        vfh_estimation;
401     vfh_estimation.setSearchMethod (pcl::search::Search<pcl::PointXYZ>::Ptr (
        new pcl::search::KdTree<pcl::PointXYZ>));
402     vfh_estimation.setInputCloud (points);
403     vfh_estimation.setInputNormals (normals);
404     pcl::PointCloud<pcl::VFHSignature308>::Ptr global_descriptor (new pcl::
        PointCloud<pcl::VFHSignature308>);
405     vfh_estimation.compute (*global_descriptor);
406     return (global_descriptor);
407 }
408
409
410 icpResult PclFilters::object_detection(pcl::PointCloud<pcl::PointXYZ>::Ptr
    cloud, std::vector<RayTraceCloud> model_a, std::vector<RayTraceCloud>
    model_b){
411     pcl::PointCloud<pcl::PointXYZ>::Ptr section_a(new pcl::PointCloud<pcl::
        PointXYZ>);
412     pcl::PointCloud<pcl::PointXYZ>::Ptr section_b(new pcl::PointCloud<pcl::
        PointXYZ>);
413     pcl::copyPointCloud(*cloud,*section_a);
414     //Passthrough
415     section_a = passthrough_filter(section_a,0.760,1.190,"z");
416     pcl::copyPointCloud(*section_a,*section_b);
417     section_a = passthrough_filter(section_a,-0.510,-0.130,"x");
418     section_b = passthrough_filter(section_b,-0.130,0.270,"x");
419     //Voxelgrid
420     section_a = voxel_grid_filter(section_a,0.001,0.001,0.001);
421     section_b = voxel_grid_filter(section_b,0.001,0.001,0.001);
422     //Cluster extraction
423     std::vector<pcl::PointCloud<pcl::PointXYZ>::Ptr> clusters_section_a =
        cluster_extraction(section_a,0.005);
424     std::vector<pcl::PointCloud<pcl::PointXYZ>::Ptr> clusters_section_b =
        cluster_extraction(section_b,0.005);
425
426     //find the cluster that contains the part we are looking for.
427     RayTraceCloud part_a, part_b;
428     if(clusters_section_a.size() != 1){
429         //more than one cluster, find the correct one
430         std::vector<RayTraceCloud> cluster_a_models;
431         for(int i = 0; i<clusters_section_a.size(); i++){
432             RayTraceCloud tmp_model;
433             tmp_model.cloud = clusters_section_a.at(i);
434             cluster_a_models.push_back(calculate_features(tmp_model));

```

```

435     }
436     int tmp = search_for_model(cluster_a_models,model_a);
437     part_a = cluster_a_models.at(tmp);
438 }
439 else{
440     part_a.cloud = clusters_section_a.at(0);
441     part_a = calculate_features(part_a);
442 }
443 if(clusters_section_b.size() != 1){
444     //more than one cluster, find the correct one
445     std::vector<RayTraceCloud> cluster_b_models;
446     for(int i = 0; i<clusters_section_b.size(); i++){
447         RayTraceCloud tmp_model;
448         tmp_model.cloud = clusters_section_b.at(i);
449         cluster_b_models.push_back(calculate_features(tmp_model));
450     }
451     int tmp = search_for_model(cluster_b_models,model_b);
452     part_b = cluster_b_models.at(tmp);
453 }
454 else{
455     part_b.cloud = clusters_section_b.at(0);
456     part_b = calculate_features(part_b);
457 }
458
459 //from here, we assume that left and right part contains the correct
cluster for each of the parts.
460 std::vector<float> result_a,result_b;
461 result_a = match_cloud(part_a,generate_search_tree(model_a));
462 result_b = match_cloud(part_b,generate_search_tree(model_b));
463
464 //alignment part a
465 Eigen::Matrix4f initial_a = calculateInitialAlignment(model_a.at(result_a.
    at(0)),part_a,0.01,1,50);
466 Eigen::Matrix4f final_a = calculateRefinedAlignment(model_a.at(result_a.at
    (0)),part_a,initial_a,0.1,0.1,1e-10,0.00001,50);
467
468 //alignment part b
469 Eigen::Matrix4f initial_b = calculateInitialAlignment(model_b.at(result_b.
    at(0)),part_b,0.01,1,50);
470 Eigen::Matrix4f final_b = calculateRefinedAlignment(model_b.at(result_b.at
    (0)),part_b,initial_b,0.1,0.1,1e-10,0.00001,50);
471
472 //the following is just to produce a pleasing image showing the result.
473 //Transform the models
474 pcl::PointCloud<pcl::PointXYZ>::Ptr a_positioned(new pcl::PointCloud<pcl::
    PointXYZ>);
475 pcl::copyPointCloud(*model_a.at(result_a.at(0)).cloud,*a_positioned);
476 pcl::transformPointCloud(*a_positioned,*a_positioned,final_a);
477
478 pcl::PointCloud<pcl::PointXYZ>::Ptr b_positioned(new pcl::PointCloud<pcl::
    PointXYZ>);
479 pcl::copyPointCloud(*model_b.at(result_b.at(0)).cloud,*b_positioned);
480 pcl::transformPointCloud(*b_positioned,*b_positioned,final_b);
481
482 //Display the cloud and models
483 pcl::PointCloud<pcl::PointXYZRGB>::Ptr scene_copy,b_transformed,
    a_transformed;

```

```
484 |     scene_copy = color_cloud(cloud,255,255,255);
485 |     b_transformed = color_cloud(b_positioned,255,0,0);
486 |     a_transformed = color_cloud(a_positioned,0,255,0);
487 |     *scene_copy += *b_transformed;
488 |     *scene_copy += *a_transformed;
489 |
490 |     icpResult result;
491 |     result.cloud = scene_copy;
492 |     result.partAFinal = final_a * model_a.at(result_a.at(0)).pose;
493 |     result.partBFinal = final_b * model_b.at(result_b.at(0)).pose;
494 |     return result;
495 | }
496 | }
```

pcl_filters.hpp

Listing A.2: Source file - agilus_master_project/pcl_filters.hpp

```
1 //
2 // Original author Kristoffer Larsen. Latest change date 01.05.2016
3 // This .hpp file defines the content of pcl_filters.cpp which is a toolbox
4 // that implements the most commonly used features from PCL.
5 // In this application, pcl_filters.cpp is used for 3D point cloud processing.
6 //
7 // Created as part of the software solution for a Master's Thesis in Production
8 // Technology at NTNU Trondheim.
9 //
10 #ifndef qt_filter_tester_PCLFILTERS_H
11 #define qt_filter_tester_PCLFILTERS_H
12
13 #include <QObject>
14 #include <iostream>
15 #include <boost/thread/thread.hpp>
16
17 #include <pcl/common/common_headers.h>
18 #include <pcl/io/pcd_io.h>
19 #include <pcl/visualization/pcl_visualizer.h>
20 #include <pcl/console/parse.h>
21
22 //PCL Filters
23 #include <pcl/filters/voxel_grid.h>
24 #include <pcl/filters/shadowpoints.h>
25 #include <pcl/filters/extract_indices.h>
26 #include <pcl/filters/passthrough.h>
27 #include <pcl/filters/median_filter.h>
28 #include <pcl/filters/statistical_outlier_removal.h>
29 #include <pcl/filters/fast_bilateral_omp.h>
30
31 //PCL Feature estimation
32 #include <pcl/kdtree/kdtree_flann.h>
33 #include <pcl/features/integral_image_normal.h>
34 #include <pcl/features/normal_3d.h>
35 #include <pcl/features/normal_3d_omp.h>
36 #include <pcl/features/cvfh.h>
37 #include <pcl/features/gfpfh.h>
38 #include <pcl/features/our_cvfh.h>
39 #include <pcl/features/esf.h>
40 #include <pcl/features/fpfh.h>
41 #include <pcl/features/fpfh_omp.h>
42 #include "pcl/keypoints/sift_keypoint.h"
43
44 //PCL Registration and object detection
45 #include <pcl/registration/ia_ransac.h>
46 #include <pcl/sample_consensus/method_types.h>
47 #include <pcl/sample_consensus/model_types.h>
48 #include <pcl/ModelCoefficients.h>
49 #include <pcl/segmentation/sac_segmentation.h>
50 #include <pcl/segmentation/extract_clusters.h>
```

```

51 #include <pcl/registration/icp.h>
52 #include <pcl/recognition/ransac_based/obj_rec_ransac.h>
53
54 /*!
55 * \brief The RayTraceCloud struct contain all the data related to a training
56 set model.
57 */
58 struct RayTraceCloud {
59     /*! The point cloud of the ray trace */
60     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud;
61
62     /*! The pose transformation from the camera to the mesh when the ray trace
63     was generated */
64     Eigen::Matrix4f pose;
65
66     /*! The amount of the whole mesh seen in the camera */
67     float entrophy;
68
69     /*! The clouds keypoints */
70     pcl::PointCloud<pcl::PointXYZ>::Ptr keypoints;
71
72     /*! The clouds surface normal */
73     pcl::PointCloud<pcl::Normal>::Ptr normals;
74
75     /*! The clouds local descriptors */
76     pcl::PointCloud<pcl::FPFHSignature33>::Ptr local_descriptors;
77
78     /*! The clouds global descriptor */
79     pcl::PointCloud<pcl::VFHSignature308>::Ptr global_descriptors;
80 };
81
82 /*!
83 * \brief The icpResult struct contain all the important data derived from the
84 3D object detection procedure.
85 */
86 struct icpResult {
87     /*! A 3D point cloud that illustrates the 3D object detection result */
88     pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud;
89
90     /*! The estimated pose of part A */
91     Eigen::Matrix4f partAFinal;
92
93     /*! The estimated pose of part B */
94     Eigen::Matrix4f partBFinal;
95 };
96
97 namespace agilus_master_project {
98
99     class PclFilters : public QObject{
100     Q_OBJECT
101
102     public:
103
104     /*!
105     * \brief Constructor for the PclFilters class.
106     * \param parent Default 0.

```

```

105     */
106     PclFilters(QObject *parent = 0);
107
108     ~PclFilters();
109
110     /*!
111     * \brief Return the cluster that is most similar to the provided training
112     * set.
113     * \param clusters Clusters that is to be searched.
114     * \param model The training set of the model that is searched for in the
115     * scene.
116     * \return The index that corresponds to the best matching cluster in the
117     * input cluster vector.
118     */
119     int search_for_model(std::vector<RayTraceCloud> clusters, std::vector<
120     RayTraceCloud> model);
121
122     /*!
123     * \brief Method for testing an experimental 3D object detection
124     * implementation. This implementation is based on RANSAC.
125     * \param models Training set that is to be searched for.
126     * \param object Object cluster.
127     */
128     void ransac_recognition(std::vector<RayTraceCloud> models, RayTraceCloud
129     object);
130
131     /*!
132     * \brief Estimates an initial alignment bewteen the source and target
133     * model based on the Sigular Value Decomposition approach.
134     * \param source The source point cloud.
135     * \param target The target point cloud.
136     * \param min_sample_distance Mathcing parameter used to limit
137     * correspondences.
138     * \param max_correspondence_distance Matching parameter used to limit
139     * correspondences.
140     * \param nr_iterations Maximum number of iterations run before returning a
141     * pose.
142     * \return The estimated initial alignment.
143     */
144     Eigen::Matrix4f calculateInitialAlignment(RayTraceCloud source,
145     RayTraceCloud target, float min_sample_distance, float
146     max_correspondence_distance, int nr_iterations);
147
148     /*!
149     * \brief Estimates a final alignment between the source and target model
150     * based on the Iterative Closest Point approach.
151     * \param source The source point cloud.
152     * \param target The target point cloud.
153     * \param initial_alignment The initial alignment between the models used
154     * as a starting point.
155     * \param max_correspondence_distance Matching parameter used to limit
156     * correspondences.
157     * \param outlier_rejection_threshold Parameter used to set the outlier
158     * rejection threshold.
159     * \param transformation_epsilon Parameter that defines an acceptable
160     * transformation epsilon.
161     * \param eucledian_fitness_epsilon Parameter that defines an acceptable

```

```

145     euclidian fitness epsilon.
146 * \param max_iterations Maximum number of iterations run before returning
147   a pose.
148 * \return The estimated final alignment.
149 */
150 Eigen::Matrix4f calculateRefinedAlignment (RayTraceCloud source,
151   RayTraceCloud target, Eigen::Matrix4f initial_alignment, float
152   max_correspondence_distance, float outlier_rejection_threshold, float
153   transformation_epsilon, float euclidian_fitness_epsilon, int
154   max_iterations);
155
156 /*!
157 * \brief Generates a Kd-tree used for nearest neighbour search from a set
158   of global descriptors.
159 * \param Training set containing global descriptors.
160 * \return The generated Kd-tree.
161 */
162 pcl::KdTreeFLANN<pcl::VFHSignature308>::Ptr generate_search_tree(std::
163   vector<RayTraceCloud> models);
164
165 /*!
166 * \brief Descriptor matching between the input model and a generated Kd-
167   tree based on the VFH global descriptor.
168 * \param object_model The input model.
169 * \param search_tree The input Kd-tree
170 * \return A vector containing <index of best matching model, confidence
171   level of the match>.
172 */
173 std::vector<float> match_cloud(RayTraceCloud object_model, pcl::KdTreeFLANN
174   <pcl::VFHSignature308>::Ptr search_tree);
175
176 /*!
177 * \brief Descriptor matching between the input model and a generated Kd-
178   tree based on the CVFH global descriptor.
179 * \param object_model The input model.
180 * \param search_tree The input Kd-tree.
181 * \return A vector containing <index of best matching model, confidence
182   level of the match>.
183 */
184 std::vector<float> temp_matching_cvfh(RayTraceCloud object_model, pcl::
185   KdTreeFLANN<pcl::VFHSignature308>::Ptr search_tree);
186
187 /*!
188 * \brief Calculated the keypoints of an input point cloud based on the
189   SIFT3D keypoint selector method.
190 * \param cloud The input point cloud.
191 * \param min_scale SIFT3D minimum scale parameter.
192 * \param nr_octaves SIFT3D number of octaves calculated parameter.
193 * \param nr_scales_per_octave SIFT3D number of scales per octave
194   calculated parameter.
195 * \param min_contrast SIFT3D minimum contrast parameter.
196 * \return A point cloud containing the resulting keypoints.
197 */
198 pcl::PointCloud<pcl::PointXYZ>::Ptr calculate_keypoints(pcl::PointCloud<
199   pcl::PointXYZ>::Ptr cloud, float min_scale, int nr_octaves, int
200   nr_scales_per_octave, float min_contrast);

```

```

184  /*!
185  * \brief Estimates the FPFH local descriptor for a 3D point cloud.
186  * \param cloud The input point cloud.
187  * \param normal The surface normals of the input point cloud.
188  * \param keypoints The keypoints of the input point cloud.
189  * \param feature_radius The feature radius FPFH parameter.
190  * \return A point cloud containing a local descriptor for each keypoint of
191         the original input point cloud.
192  */
193  pcl::PointCloud<pcl::FPFHSignature33>::Ptr calculate_local_descriptor(pcl::
194  PointCloud<pcl::PointXYZ>::Ptr cloud, pcl::PointCloud<pcl::Normal>::Ptr
195  normal, pcl::PointCloud<pcl::PointXYZ>::Ptr keypoints, float
196  feature_radius);
197
198  /*!
199  * \brief Estimates the VFH global descriptor for a 3D point cloud.
200  * \param points The input point cloud.
201  * \param normals The surface normals of the input point cloud.
202  * \return The VFH global descriptor for the input point cloud.
203  */
204  pcl::PointCloud<pcl::VFHSignature308>::Ptr calculate_vfh_descriptors(pcl::
205  PointCloud<pcl::PointXYZ>::Ptr points, pcl::PointCloud<pcl::Normal>::
206  Ptr normals);
207
208  /*!
209  * \brief Estimates surface normals, keypoints, local descriptors and
210         global descriptor for the input point cloud.
211  * \param inputcloud The input point cloud.
212  * \return A struct containing all the calculated features.
213  */
214  RayTraceCloud calculate_features(RayTraceCloud inputcloud);
215
216  /*!
217  * \brief Creates a PCL Visualizer containing the input cloud.
218  * \param cloud Input point cloud.
219  * \return A pcl visualizer containing the input cloud.
220  */
221  boost::shared_ptr<pcl::visualization::PCLVisualizer> visualize (pcl::
222  PointCloud<pcl::PointXYZ>::Ptr cloud);
223
224  /*!
225  * \brief Creates a PCL Visualizer to visualize a PointXYZRGB point cloud.
226  * \param cloud Input point cloud.
227  * \return A pcl visualizer containing the input cloud.
228  */
229  boost::shared_ptr<pcl::visualization::PCLVisualizer> visualize_rgb(pcl::
230  PointCloud<pcl::PointXYZRGB>::Ptr cloud);
231
232  /*!
233  * \brief Creates a PCL Visualizer used to visualize a point clouds normals
234         .
235  * \param cloud Input point cloud.
236  * \param radius Double value for the search radius for the normal
237         estimation of a p.oint cloud.
238  * \param numOfNormals Integer value for the number of normals to display
239         in the visualizer.

```



```

229     * \return A pcl visualizer containing the input cloud and normals as
230         defined by the input parameters.
231     */
232 boost::shared_ptr<pcl::visualization::PCLVisualizer> visualize_normals (
233     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud, double radius, int
234     numOfNormals);
235
236     /*!
237     * \brief Segments out the biggest plane in the input point cloud.
238     * \param cloud Input point cloud.
239     * \param distance Double value for the maximum distance between points in
240         a plane.
241     * \return A pcl point cloud containing the points of the segmented plane.
242     */
243 pcl::PointCloud<pcl::PointXYZ>::Ptr plane_segmentation(pcl::PointCloud<
244     pcl::PointXYZ>::Ptr cloud, double distance);
245
246     /*!
247     * \brief Creates a PCL Visualizer containing a point cloud of the clusters
248         extracted from the input cloud using Euclidian cluster extraction.
249     * \param cloud Input point cloud.
250     * \param distance Double value for the maximum distance between points in
251         a plane.
252     * \return A pcl visualizer containing the extracted clusters from the
253         input cloud.
254     */
255 std::vector<pcl::PointCloud<pcl::PointXYZ>::Ptr> cluster_extraction (pcl::
256     PointCloud<pcl::PointXYZ>::Ptr cloud, double distance);
257
258     /*!
259     * \brief Returns the most recent point cloud handled by the class.
260     * \return The most recent point cloud handled by the class.
261     */
262 pcl::PointCloud<pcl::PointXYZ>::Ptr get_filtered_cloud();
263
264     /*!
265     * \brief Colors all the points in a point cloud.
266     * \param cloud Input point cloud.
267     * \param r Integer value for the red component of the color.
268     * \param g Integer value for the green component of the color.
269     * \param b Integer value for the blue component of the color.
270     * \return The input point cloud colored in the specified color.
271     */
272 pcl::PointCloud<pcl::PointXYZRGB>::Ptr color_cloud (pcl::PointCloud<pcl::
273     PointXYZ>::Ptr cloud, int r, int g, int b);
274
275     /*!
276     * \brief Returns the surface normals of a Point cloud.
277     * \param cloud Input point cloud.
278     * \param radius Double value for the search radius of the normal
279         estimation.
280     * \return The surface normals of the input point cloud.
281     */
282 pcl::PointCloud<pcl::Normal>::Ptr get_normals (pcl::PointCloud<pcl::
283     PointXYZ>::Ptr cloud, double radius);
284
285     /*!

```

```

274 * \brief Returns the a point cloud filtered using passthrough filtering.
275 * \param cloud Input point cloud.
276 * \param min Double value for the minimum value of the filter.
277 * \param max Double value for the maximum value of the filter.
278 * \param axis std::string value for the axis of the filter (lower case).
279 * \return A point cloud filtered using passthrough filtering as specified.
280 */
281 pcl::PointCloud<pcl::PointXYZ>::Ptr passthrough_filter (pcl::PointCloud<
    pcl::PointXYZ>::Ptr cloud, double min, double max, std::string axis);
282
283 /*!
284 * \brief Returns the a point cloud filtered using voxel grid filtering.
285 * \param cloud Input point cloud.
286 * \param lx Double value for the voxel size in the "x" axis of the filter.
287 * \param ly Double value for the voxel size in the "y" axis of the filter.
288 * \param lz Double value for the voxel size in the "z" axis of the filter.
289 * \return A point cloud filtered using voxel grid filtering as specified.
290 */
291 pcl::PointCloud<pcl::PointXYZ>::Ptr voxel_grid_filter (pcl::PointCloud<
    pcl::PointXYZ>::Ptr cloud, double lx, double ly, double lz);
292
293 /*!
294 * \brief Returns the a point cloud filtered using median filtering.
295 * \param cloud Input point cloud.
296 * \param window_size Integer value for the window size of the filter.
297 * \param max_allowed_movement Double value for the maximum allowed movenet
    of the filter.
298 * \return A point cloud filtered using median filtering as specified.
299 */
300 pcl::PointCloud<pcl::PointXYZ>::Ptr median_filter (pcl::PointCloud<pcl::
    PointXYZ>::Ptr cloud, int window_size, double max_allowed_movement);
301
302 /*!
303 * \brief Returns the a point cloud filtered using shadow point removal
    filtering.
304 * \param cloud Input point cloud.
305 * \param threshold Double value for the filter threshold.
306 * \param radius Double value for the filter search radius.
307 * \return A point cloud filtered using shadow point removal filtering as
    specified.
308 */
309 pcl::PointCloud<pcl::PointXYZ>::Ptr shadowpoint_removal_filter(pcl::
    PointCloud<pcl::PointXYZ>::Ptr cloud, double threshold, double radius);
310
311 /*!
312 * \brief Returns the a point cloud filtered using statistical outlier
    removal filtering.
313 * \param cloud Input point cloud.
314 * \param meanK Integer value for the number of nearest neighbors to use
    for mean distance estimation.
315 * \param std_deviation_threshold Double value for the standard deviation
    multiplier for the distance threshold calculation.
316 * \return A point cloud filtered using statistical outlier removal
    filtering as specified.
317 */
318 pcl::PointCloud<pcl::PointXYZ>::Ptr statistical_outlier_filter(pcl::
    PointCloud<pcl::PointXYZ>::Ptr cloud, int meanK, double

```

```

std_deviation_threshold);
319
320 /*!
321  * \brief Combines all clouds in an vector to one cloud.
322  * \param input std::vector containing all clouds to be combined.
323  * \return A point cloud containing all clouds in the input vector.
324  */
325 pcl::PointCloud<pcl::PointXYZ>::Ptr combine_clouds(std::vector<pcl::
    PointCloud<pcl::PointXYZ>::Ptr> input);
326
327 /*!
328  * \brief Generates the CVFH descriptors for an object (cluster).
329  * \param object Input point cloud, cluster of the object.
330  * \param normals Input normal cloud of the object.
331  * \return The corresponding CVFH global descriptor.
332  */
333 pcl::PointCloud<pcl::VFHSignature308>::Ptr calculate_cvfh_descriptors(pcl::
    PointCloud<pcl::PointXYZ>::Ptr object);
334
335 /*!
336  * \brief Returns a pcl point cloud filtered using a bilateral filter.
337  * \param cloud Input point cloud.
338  * \param sigmaS Double value for the half size of the gaussian bilateral
 filter window.
339  * \param sigmaR Double value for the standard deviation parameter.
340  * \return A point cloud filtered using a bilateral filter.
341  */
342 pcl::PointCloud<pcl::PointXYZ>::Ptr bilateral_filter(pcl::PointCloud<pcl::
    PointXYZ>::Ptr cloud, double sigmaS, double sigmaR);
343
344 /*!
345  * \brief Returns the esf descriptor for a pcl point cloud.
346  * \param cloud Input point cloud.
347  * \return ESFSignature640 descriptor for the input point cloud.
348  */
349 pcl::PointCloud<pcl::ESFSignature640>::Ptr calculate_esf_descriptors(pcl::
    PointCloud<pcl::PointXYZ>::Ptr cloud);
350
351 /*!
352  * \brief Returns the ourcvfh descriptor for a pcl point cloud.
353  * \param cloud Input point cloud.
354  * \return OurCVFH descriptor for the input point cloud.
355  */
356 pcl::PointCloud<pcl::VFHSignature308>::Ptr calculate_ourcvfh_descriptors(
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud, pcl::PointCloud<pcl::Normal
    >::Ptr normal);
357
358 /*!
359  * \brief Returns the gfpfh descriptor for a pcl point cloud.
360  * \param cloud Input point cloud.
361  * \return GFPFH descriptor for the input point cloud.
362  */
363 pcl::PointCloud<pcl::GFPFHSignature16>::Ptr calculate_gfpfh_descriptors(
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud);
364
365 /*!
 * \brief Detects two objects in an area of the point cloud and returns a
 pointcloud showing the detected parts.

```

```

366     * \param cloud Input point cloud.
367     * \param model_a A list of RayTraceCloud objects generated by the
      ModelLoader class.
368     * \param model_b A list of RayTraceCloud objects generated by the
      ModelLoader class.
369     * \return Pointcloud showing the result.
370     */
371     icpResult object_detection(pcl::PointCloud<pcl::PointXYZ>::Ptr cloud, std::
      vector<RayTraceCloud> model_a, std::vector<RayTraceCloud> model_b);
372
373 private:
374     boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer; //!< A pcl
      viewer used to visualize pcl point clouds.
375     pcl::PointCloud<pcl::PointXYZ>::Ptr filteredCloud; //!< The product of a
      filter operation.
376     pcl::PassThrough<pcl::PointXYZ> passfilter; //!< A pcl passthrough filter
      object.
377     pcl::VoxelGrid<pcl::PointXYZ> voxelfilter; //!< A pcl voxelgrid filter
      object.
378     pcl::MedianFilter<pcl::PointXYZ> medianfilter; //!< A pcl median filter
      object.
379     pcl::StatisticalOutlierRemoval<pcl::PointXYZ> statistical_outlier; //!< A
      pcl statistical outlier removal filter object.
380     pcl::ShadowPoints<pcl::PointXYZ, pcl::Normal> shadowpoint_filter; //!< A
      pcl shadowpoints removal filter object.
381     pcl::KdTreeFLANN<pcl::VFHSignature308>::Ptr kdtree_;
382
383 public Q_SLOTS:
384     //Slots used to recieve events from one another class. All slots and
      signals are connected in main_window.cpp
385
386 Q_SIGNALS:
387     //Signals used to emit event from one class to another. All signals are
      connected in main_window.cpp
388 };
389 }
390 #endif

```

modelloder.cpp

Listing A.3: Source file - agilus_master_project/modelloder.cpp

```

1 //
2 // Original code by Adam Leon Kleppe on 01.02.16, modified by Kristoffer
   Larsen.
3 // Latest change date 01.05.2016
4 // modelloder.cpp is a tool used to create training set used for 3D object
   detection.
5 //
6 // Modified and used as part of the software solution for a Master's Thesis in
   Production Technology at NTNU Trondheim.
7 //
8
9 #include "../include/agilus_master_project/modelloder.hpp"
10
11 namespace agilus_master_project{
12
13 ModelLoader::ModelLoader(pcl::PolygonMesh mesh, std::string mesh_name) :
14     ModelLoader(mesh_name)
15 {
16     this->mesh = mesh;
17 }
18
19 ModelLoader::ModelLoader(std::string mesh_name) :
20     QObject(0)
21 {
22     this->mesh_name = mesh_name;
23     this->setCloudResolution(960);
24     this->setPath(ros::package::getPath("agilus_master_project") + "/"
        trace_clouds/");
25     this->setTesselation_level(1);
26 }
27
28 ModelLoader::~ModelLoader() {}
29
30 void ModelLoader::populateLoader() {
31     if(!this->loadPointClouds()) {
32         if(this->mesh.cloud.data.size() == 0) {
33             ROS_ERROR("There is no defined mesh to generate clouds from");
34             return;
35         }
36         this->generatePointClouds();
37         this->savePointClouds();
38     }
39 }
40
41 std::vector<RayTraceCloud> ModelLoader::getModels(bool load){
42     // Populate the loader if empty
43     if(load && this->ray_trace_clouds.empty()) {
44         this->populateLoader();
45     }
46     return this->ray_trace_clouds;
47 }
48

```

```

49 void ModelLoader::generatePointClouds() {
50     // Create mesh object
51     vtkSmartPointer<vtkPolyData> meshVTK;
52     pcl::VTKUtils::convertToVTK(this->mesh, meshVTK);
53
54     // Set up trace generation
55     ROS_INFO("Generating traces...");
56     ROS_INFO("\033[32m Current settings:");
57     ROS_INFO("\033[32m -mesh_name: %s", this->mesh_name.c_str());
58     ROS_INFO("\033[32m -cloud_resolution: %d", this->cloud_resolution);
59     ROS_INFO("\033[32m -tessellation_level: %d", this->tessellation_level);
60
61     pcl::visualization::PCLVisualizer generator("Generating traces...",false);
62     generator.addModelFromPolyData (meshVTK, "mesh", 0);
63     std::vector<pcl::PointCloud<pcl::PointXYZ>, Eigen::aligned_allocator<pcl::
        PointCloud<pcl::PointXYZ> > > clouds;
64     std::vector<Eigen::Matrix4f, Eigen::aligned_allocator<Eigen::Matrix4f> >
        poses;
65     std::vector<float> entropies;
66
67     // Generate traces
68     generator.renderViewTesselatedSphere(this->cloud_resolution, this->
        cloud_resolution, clouds, poses, entropies, this->tessellation_level);
69
70     // Generate clouds
71     this->ray_trace_clouds.clear();
72     for(int i =0; i < clouds.size(); i++)
73     {
74         RayTraceCloud cloud;
75         cloud.cloud = pcl::PointCloud<pcl::PointXYZ>::Ptr(new pcl::PointCloud<
            pcl::PointXYZ>);
76         *cloud.cloud = clouds.at(i);
77         cloud.pose = poses.at(i);
78         cloud.entropy = entropies.at(i);
79         std::cout << "Calculating features for Cloud nr. " << i << std::endl;
80         cloud = filters->calculate_features(cloud);
81         this->ray_trace_clouds.push_back(cloud);
82     }
83 }
84
85 bool ModelLoader::savePointClouds() {
86     if(this->ray_trace_clouds.empty()){
87         return false;
88     }
89
90     // Set saving path
91     std::string save_path = this->path + this->mesh_name + "/";
92     ROS_INFO("Saving ray traces");
93     ROS_INFO("\tUsing %s", save_path.c_str());
94
95     // Generate YAML node
96     YAML::Node clouds;
97     for(int i = 0; i < this->ray_trace_clouds.size(); i++) {
98         RayTraceCloud ray_trace = this->ray_trace_clouds.at(i);
99
100         std::stringstream filename_cloud;
101         std::stringstream filename_normals;

```

```

102     std::stringstream filename_keypoints;
103     std::stringstream filename_local_descriptor;
104     std::stringstream filename_global_descriptor;
105
106     filename_cloud << this->mesh_name << "_cloud_";
107     filename_cloud << setfill('0') << setw(4) << (i+1);
108     filename_cloud << ".pcd";
109     filename_normals << this->mesh_name << "_normals_";
110     filename_normals << setfill('0') << setw(4) << (i+1);
111     filename_normals << ".pcd";
112     filename_keypoints << this->mesh_name << "_keypoints_";
113     filename_keypoints << setfill('0') << setw(4) << (i+1);
114     filename_keypoints << ".pcd";
115     filename_local_descriptor << this->mesh_name << "_ldescriptor_";
116     filename_local_descriptor << setfill('0') << setw(4) << (i+1);
117     filename_local_descriptor << ".pcd";
118     filename_global_descriptor << this->mesh_name << "_gdescriptor_";
119     filename_global_descriptor << setfill('0') << setw(4) << (i+1);
120     filename_global_descriptor << ".pcd";
121
122     boost::filesystem::create_directories(save_path);
123     pcl::io::savePCDFile(save_path + filename_cloud.str(), *ray_trace.cloud
124         );
125     pcl::io::savePCDFile(save_path + filename_normals.str(), *ray_trace.
126         normals);
127     pcl::io::savePCDFile(save_path + filename_keypoints.str(), *ray_trace.
128         keypoints);
129     pcl::io::savePCDFile(save_path + filename_local_descriptor.str(), *
130         ray_trace.local_descriptors);
131     pcl::io::savePCDFile(save_path + filename_global_descriptor.str(), *
132         ray_trace.global_descriptors);
133
134     YAML::Node node;
135     node["cloud"] = filename_cloud.str();
136     node["normals"] = filename_normals.str();
137     node["keypoints"] = filename_keypoints.str();
138     node["ldescriptor"] = filename_local_descriptor.str();
139     node["gdescriptor"] = filename_global_descriptor.str();
140     for(int j = 0; j < 16; j++) {
141         node["pose"].push_back(ray_trace.pose(j / 4, j % 4));
142     }
143     node["entropy"] = ray_trace.entropy;
144
145     std::stringstream cloud_node;
146     cloud_node << setfill('0') << setw(4) << (i+1);
147     clouds[cloud_node.str()] = node;
148 }
149
150 // Saving the YAML node
151 YAML::Emitter out;
152 out << clouds;
153 boost::filesystem::ofstream f(save_path + this->mesh_name + ".yaml");
154 f << out.c_str();
155 ROS_INFO("\033[33mSuccessfully saved %d ray traces", (int)this->
156     ray_trace_clouds.size());

```

```

153 bool ModelLoader::loadPointClouds() {
154     // Set the save path
155     std::string save_path = this->path + this->mesh_name + "/";
156     YAML::Node clouds;
157
158     // Try to load the files
159     try {
160         ROS_INFO("Loading ray trace clouds...");
161         ROS_INFO("\tUsing %s", save_path.c_str());
162         clouds = YAML::LoadFile(save_path + this->mesh_name + ".yaml");
163     }
164     catch (const std::exception& e) {
165         ROS_INFO("\033[31mFile not found.\033[0m");
166         return false;
167     }
168
169     int i = 1;
170     std::stringstream cloud_node;
171     cloud_node << setfill('0') << setw(4) << i;
172     this->ray_trace_clouds.clear();
173
174     // Populate ray_trace_cloud from the YAML node
175     while(clouds[cloud_node.str()]){
176         YAML::Node cloud = clouds[cloud_node.str()];
177
178         RayTraceCloud ray_trace;
179         ray_trace.cloud = pcl::PointCloud<pcl::PointXYZ>::Ptr(new pcl::
            PointCloud<pcl::PointXYZ>);
180         ray_trace.normals = pcl::PointCloud<pcl::Normal>::Ptr(new pcl::
            PointCloud<pcl::Normal>);
181         ray_trace.keypoints = pcl::PointCloud<pcl::PointXYZ>::Ptr(new pcl::
            PointCloud<pcl::PointXYZ>);
182         ray_trace.local_descriptors = pcl::PointCloud<pcl::FPFHSignature33>::
            Ptr(new pcl::PointCloud<pcl::FPFHSignature33>);
183         ray_trace.global_descriptors = pcl::PointCloud<pcl::VFHSignature308>::
            Ptr(new pcl::PointCloud<pcl::VFHSignature308>);
184
185         pcl::io::loadPCDFFile(save_path + cloud["cloud"].as<std::string>(), *
            ray_trace.cloud);
186         pcl::io::loadPCDFFile(save_path + cloud["normals"].as<std::string>(), *
            ray_trace.normals);
187         pcl::io::loadPCDFFile(save_path + cloud["keypoints"].as<std::string>(),
            *ray_trace.keypoints);
188         pcl::io::loadPCDFFile(save_path + cloud["ldescriptor"].as<std::string>()
            , *ray_trace.local_descriptors);
189         pcl::io::loadPCDFFile(save_path + cloud["gdescriptor"].as<std::string>()
            , *ray_trace.global_descriptors);
190
191         for(int x = 0; x < 4; x++) {
192             for(int y = 0; y < 4; y++) {
193                 ray_trace.pose(x, y) = cloud["pose"][(int)(x*4 + y)].as<float
                    >();
194             }
195         }
196         ray_trace.entropy = cloud["entropy"].as<float>();
197         this->ray_trace_clouds.push_back(ray_trace);
198         cloud_node.clear();

```



```
199     cloud_node.str(std::string());
200     cloud_node << setfill('0') << setw(4) << ++i;
201 }
202 ROS_INFO("\033[33mSuccessfully loaded %d ray traces\033[0m", i-1);
203 return true;
204 }
205 }
```

modelloader.hpp

Listing A.4: Source file - agilus_master_project/modelloader.hpp

```

1 //
2 // Original code by Adam Leon Kleppe on 01.02.16, modified by Kristoffer
   Larsen.
3 // Latest change date 01.05.2016
4 // modelloader.cpp is a tool used to create training set used for 3D object
   detection.
5 //
6 // Modified and used as part of the software solution for a Master's Thesis in
   Production Technology at NTNU Trondheim.
7 //
8
9 #ifndef qt_filter_tester_MODELLOADER_H
10 #define qt_filter_tester_MODELLOADER_H
11
12 #include <QObject>
13
14 #include <ros/ros.h>
15 #include <ros/package.h>
16
17 #include <pcl/common/transforms.h>
18 #include <pcl_conversions/pcl_conversions.h>
19 #include <pcl/point_cloud.h>
20 #include <pcl/point_types.h>
21 #include <pcl/surface/vtk_smoothing/vtk_utils.h>
22 #include <pcl/visualization/pcl_visualizer.h>
23
24 #include <boost/filesystem.hpp>
25 #include <boost/filesystem/fstream.hpp>
26 #include <Eigen/Core>
27 #include <yaml-cpp/yaml.h>
28
29 #include <vector>
30 #include <string.h>
31 #include <sstream>
32 #include <iomanip>
33
34 #include "pcl_filters.hpp"
35
36 namespace agilus_master_project {
37
38 class ModelLoader : public QObject
39 {
40     Q_OBJECT
41
42 public:
43     /*!
44     * \brief Constructor for the ModelLoader class
45     * \param mesh Polygon Mesh that will be used to create a training set.
46     * \param mesh_name The name of the training set.
47     */
48     ModelLoader(pcl::PolygonMesh mesh, std::string mesh_name);
49

```

```

50  /*!
51   * \brief Constructor for the ModelLoader class
52   * \param mesh_name The name of the training set to load.
53   */
54  ModelLoader(std::string mesh_name);
55
56  ~ModelLoader();
57
58  /*!
59   * \brief Returns the models of the selected training set.
60   * \param load Set true if the models are not loaded.
61   * \return The models of the selected training set.
62   */
63  std::vector<RayTraceCloud> getModels(bool load = false);
64
65  /*!
66   * \brief Creates a complete training set for the input polygon mesh.
67   */
68  void populateLoader();
69
70  /*!
71   * \brief Sets the polygon mesh used for training set creation.
72   * \param mesh The polygon mesh that is to be used.
73   */
74  void setMesh(pcl::PolygonMesh mesh){
75      ModelLoader::mesh = mesh;
76  }
77
78  /*!
79   * \brief Sets the name of the training set.
80   * \param mesh_name The name of the training set.
81   */
82  void setMeshName(std::string mesh_name){
83      ModelLoader::mesh_name = mesh_name;
84  }
85
86  /*!
87   * \brief Sets the wanted tessellation level for the viewpoint rendering.
88   * \param tessellation_level Tessellation level, 1=42, 2=162 ...
89   */
90  void setTessellation_level(int tessellation_level){
91      ModelLoader::tessellation_level = tessellation_level;
92  }
93
94  /*!
95   * \brief Sets the viewpoint rendering resolution.
96   * \param cloud_resolution The wanted resolution.
97   */
98  void setCloudResolution(int cloud_resolution){
99      ModelLoader::cloud_resolution = cloud_resolution;
100 }
101
102 /*!
103  * \brief Sets the output path of the training set creation process.
104  * \param path The wanted output path.
105  */
106 void setPath(const std::string &path){

```

```
107     ModelLoader::path = path;
108 }
109
110 Q_SIGNALS:
111     //Signals used to emit events from one class to another. All signals are
        connected in main_window.cpp
112
113 public Q_SLOTS:
114     //Slots used to receive events from one another class. All slots and
        signals are connected in main_window.cpp
115
116 private:
117     /*!
118      * \brief This function will generate the traces from a mesh and populate
        the ray_trace_clouds variable.
119      */
120     void generatePointClouds();
121
122     /*!
123      * \brief This function will load and populate the ray_trace_clouds
        variable from the given path.
124      * \return False if the loading failed.
125      */
126     bool loadPointClouds();
127
128     /*!
129      * \brief This function will save all the information from the
        ray_trace_clouds variable to the given path.
130      * \return False if the saving action failed.
131      */
132     bool savePointClouds();
133
134     std::string path; //!< The path for saving and loading files.
135     std::string mesh_name; //!< The name of the mesh. Used for saving and
        loading file names.
136     std::vector<RayTraceCloud> ray_trace_clouds; //!< List of ray trace clouds.
137     pcl::PolygonMesh mesh; //!< The mesh which is used for generation.
138     int cloud_resolution; //!< The resolution camera when generating clouds.
139     int tessellation_level; //!< The tessellation level of the sphere for the
        camera.
140     PclFilters *filters; //!< Object for calculating features of the raytraced
        models.
141 };
142
143 }
144
145 #endif // MODELLOADER_HPP
```


Appendix B: The `image_processor` Application

This appendix contains the source code for the `image_processor` ROS node. This ROS node is used to publish 2D object detection data required by the `agilus_master_project`.

`object_2D_matcher.cpp` - This class runs the main object detection. It initializes the object detection and utilizes the methods implemented in `openCV_matching.cpp` in order to process each image frame obtained from the connected camera. Furthermore, the object detection data is published via ROS and the detection algorithm is controllable via ROS services.

`object_2D_matcher.hpp` - This is the *Header* file for the `object_2D_matcher.cpp` class. This file defines the content of the `.cpp` file.

`openCV_matching.cpp` - This class handles the image processing using OpenCV. It implements the needed methods in order to perform keypoint detection, descriptor extraction and matching using different algorithms, e.g. SIFT and SURF. It also holds the methods used to visualize the results and compute image coordinates and orientations of the detected objects.

`openCV_matching.hpp` - This is the *Header* file for the `openCV_matching.cpp` class. This file defines the content of the `.cpp` file.

`object_2D_matcher.cpp`

Listing B.1: Source file - `code/image_processor/object_2D_matcher.cpp`

```
1 //
2 // Original author: Asgeir Bjoerkedal. Created: 10.03.16. Last edit: 30.05.16.
3 //
4 // Main application for 2D object detection. Communicates via ROS and utilizes
5 // the methods defined in the header file
6 //
7 // Created as part of the software solution for a Master's thesis in Production
8 // Technology at NTNU Trondheim.
9 //
10 #include "../include/image_processor/openCV_matching.hpp"
11 #include "../include/image_processor/object_2D_matcher.hpp"
12 // Local variables
13 robotcam::OpenCVMatching openCVMatching;
14 robotcam::CurrentMatch match1;
15
16 // Video and reference images
```

```

17 cv::VideoCapture capture;
18 cv::Mat object1;
19
20 // Keypoints and descriptors
21 cv::Ptr<cv::Feature2D> detector, extractor;
22 std::vector<cv::KeyPoint> keypoints_object1, keypoints_scene;
23 cv::Mat descriptor_object1, descriptor_scene;
24
25 // Controls initialized
26 bool running = true;
27 bool binary = false;
28 bool bruteforce = true;
29 bool color = true;
30 bool undistort = true;
31 double lambda = 0.138;
32
33 int main(int argc, char **argv) {
34     ros::init(argc, argv, "object_2D_detection");
35     ros::NodeHandle n;
36     // cv_bridge for image transport.
37     image_transport::ImageTransport it(n);
38     // ROS Topics for image and object data streams.
39     image_transport::Publisher processed_pub = it.advertise("/
40         object_2D_detected/image", 1);
41     ros::Publisher pub1 = n.advertise<geometry_msgs::Pose2D>("/
42         object_2D_detected/object1", 1);
43     // ROS Services for detection controls.
44     ros::ServiceServer service1 = n.advertiseService("/object_2D_detection/
45         setProcessRunning", setProcessRunningCallback);
46     ros::ServiceServer service2 = n.advertiseService("/object_2D_detection/
47         getProcessRunning", getProcessRunningCallback);
48     ros::ServiceServer service3 = n.advertiseService("/object_2D_detection/
49         setBinaryMatching", setBinaryMatchingCallback);
50     ros::ServiceServer service4 = n.advertiseService("/object_2D_detection/
51         getBinaryMatching", getBinaryMatchingCallback);
52     ros::ServiceServer service5 = n.advertiseService("/object_2D_detection/
53         setKeypointDetectorType", setKeypointDetectorTypeCallback);
54     ros::ServiceServer service6 = n.advertiseService("/object_2D_detection/
55         getKeypointDetectorType", getKeypointDetectorTypeCallback);
56     ros::ServiceServer service7 = n.advertiseService("/object_2D_detection/
57         setDescriptorType", setDescriptorTypeCallback);
58     ros::ServiceServer service8 = n.advertiseService("/object_2D_detection/
59         getDescriptorType", getDescriptorTypeCallback);
60     ros::ServiceServer service9 = n.advertiseService("/object_2D_detection/
61         setVideoColor", setVideoColorCallback);
62     ros::ServiceServer service10 = n.advertiseService("/object_2D_detection/
63         getVideoColor", getVideoColorCallback);
64     ros::ServiceServer service11 = n.advertiseService("/object_2D_detection/
65         setBruteforceMatching", setBruteforceMatchingCallback);
66     ros::ServiceServer service12 = n.advertiseService("/object_2D_detection/
67         getBruteforceMatching", getBruteforceMatchingCallback);
68     ros::ServiceServer service13 = n.advertiseService("/object_2D_detection/
69         setVideoUndistortion", setVideoUndistortionCallback);
70     ros::ServiceServer service14 = n.advertiseService("/object_2D_detection/
71         getVideoUndistortion", getVideoUndistortionCallback);
72     ros::ServiceServer service15 = n.advertiseService("/object_2D_detection/set
73         MatchingImage1", setMatchingImage1Callback);

```

```

57     ros::ServiceServer service16 = n.advertiseService("/object_2D_detection/
        setImageDepth", setImageDepthCallback);
58     ros::Rate loop_rate(FREQ);
59     // Check camera
60     if (!capture.open(0)) {
61         ROS_ERROR(" --(!) Could not reach camera");
62         return 0;
63     }
64     initializeMatcher(VIDEO_WIDTH, VIDEO_HEIGHT);
65     // Check reference images
66     if (!object1.data) {
67         ROS_ERROR(" --(!) Error reading image");
68         return 0;
69     }
70     ROS_INFO("Loaded reference image:\n\t%s", temp_path1.c_str());
71     // Prepare the query image
72     detectAndComputeReference(object1, keypoints_object1, descriptor_object1);
73     writeReferenceImage(object1, keypoints_object1, ref_path1);
74     // Load camera matrix and distortion coefficients.
75     cv::Mat cameraMatrix = openCVMatching.getCameraMatrix(CAMERA_PARAMS);
76     cv::Mat distCoeffs = openCVMatching.getDistortionCoeff(CAMERA_PARAMS);
77     // ROS message to be published.
78     sensor_msgs::ImagePtr image_msg;
79     // Loop object detection
80     while (ros::ok()) {
81         cv::Mat video = openCVMatching.captureFrame(color, undistort, capture,
            cameraMatrix, distCoeffs);
82         if (video.empty()) break;
83         if (running) {
84             // Detect keypoints and compute time used
85             double d = (double)cv::getTickCount();
86             detector->detect(video, keypoints_scene);
87             d = ((double)cv::getTickCount() - d)/cv::getTickFrequency();
88             // Extract descriptors and compute time used
89             double e = (double)cv::getTickCount();
90             extractor->compute(video, keypoints_scene, descriptor_scene);
91             e = ((double)cv::getTickCount() - e)/cv::getTickFrequency();
92             // Match descriptors of query and training scene and compute time
                used
93             std::vector<cv::DMatch> good_matches;
94             double m = 0.0;
95             if (!binary) {
96                 if (bruteforce) {
97                     m = (double)cv::getTickCount();
98                     good_matches = openCVMatching.bruteForce(descriptor_object1
                        , descriptor_scene, cv::NORM_L1);
99                     m = ((double)cv::getTickCount() - m)/cv::getTickFrequency()
                ;
100                } else {
101                    m = (double)cv::getTickCount();
102                    good_matches = openCVMatching.knnMatchDescriptors(
                        descriptor_object1, descriptor_scene, 0.9f);
103                    m = ((double)cv::getTickCount() - m)/cv::getTickFrequency()
                ;
104                }
105            } else {
106                if (bruteforce) {

```



```

107         m = (double)cv::getTickCount();
108         good_matches = openCVMatching.bruteForce(descriptor_object1
109             , descriptor_scene, cv::NORM_HAMMING);
110         m = ((double)cv::getTickCount() - m)/cv::getTickFrequency()
111             ;
112     } else {
113         m = (double)cv::getTickCount();
114         good_matches = openCVMatching.knnMatchDescriptorsLSH(
115             descriptor_object1, descriptor_scene, 0.9f);
116         m = ((double)cv::getTickCount() - m)/cv::getTickFrequency()
117             ;
118     }
119 }
120 //std::cout << d << " " << e << " " << m << " " << d+e+m << std::
121 //endl; // Print measured processing time.
122 // Publish image data at ROS topic.
123 if ((!keypoints_object1.size() == 0 && !keypoints_scene.size() ==
124     0) && good_matches.size() >= 0) {
125     match1 = openCVMatching.visualizedMatch(video, object1,
126         keypoints_object1, keypoints_scene, good_matches, true,
127         homographyMethod);
128     image_msg = cv_bridge::CvImage(std_msgs::Header(), sensor_msgs
129         ::image_encodings::BGR8, match1.outFrame).toImageMsg();
130     processed_pub.publish(image_msg);
131 } else {
132     image_msg = cv_bridge::CvImage(std_msgs::Header(), sensor_msgs
133         ::image_encodings::BGR8, video).toImageMsg();
134     processed_pub.publish(image_msg);
135 }
136 } else {
137     image_msg = cv_bridge::CvImage(std_msgs::Header(), sensor_msgs::
138         image_encodings::BGR8, video).toImageMsg();
139     processed_pub.publish(image_msg);
140 }
141 // Publish object pose at ROS topic if the match is good.
142 if (match1.sceneCorners.size() == 4 && openCVMatching.
143     checkObjectInnerAngles(match1.sceneCorners, 80, 100)) {
144     double x = openCVMatching.getXpos(match1.sceneCorners);
145     double y = openCVMatching.getYpos(match1.sceneCorners);
146
147     object_pose_msg.theta = openCVMatching.getObjectAngle(video, match1
148         .sceneCorners);
149
150     Eigen::Vector3d temp = openCVMatching.getNormImageCoords(x,y,lambda
151         , cameraMatrix);
152
153     object_pose_msg.x = temp(0);
154     object_pose_msg.y = temp(1);
155     pub1.publish(object_pose_msg);
156 }
157 ros::spinOnce();
158 loop_rate.sleep();
159 }
160 ROS_INFO("Object detection shutting down");
161 return 0;
162 }
163 }

```

```
150 void initializeMatcher(const int video_width, const int video_height) {
151     object1 = readImage(temp_path1);
152     capture.set(CV_CAP_PROP_FRAME_WIDTH, video_width);
153     capture.set(CV_CAP_PROP_FRAME_HEIGHT, video_height);
154     ROS_INFO("Camera resolution: width=%f, height=%f", capture.get(
        CV_CAP_PROP_FRAME_WIDTH), capture.get(CV_CAP_PROP_FRAME_HEIGHT));
155     detector = openCVMatChing.setKeyPointsDetector(DETECTOR_TYPE);
156     extractor = openCVMatChing.setDescriptorsExtractor(EXTRACTOR_TYPE, binary);
157     ROS_INFO("Bruteforce matching: %d", bruteforce);
158 }
159
160 void detectAndComputeReference(cv::Mat &object, std::vector<cv::KeyPoint> &
    keypoints_object, cv::Mat &descriptor_object) {
161     detector->detect(object, keypoints_object);
162     extractor->compute(object, keypoints_object, descriptor_object);
163 }
164
165 void writeReferenceImage(cv::Mat object, std::vector<cv::KeyPoint>
    keypoints_object, std::string ref_path) {
166     cv::Mat ref_keypoints;
167     cv::drawKeypoints(object, keypoints_object, ref_keypoints, CV_RGB(0, 255,
        255), cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);
168     cv::imwrite(ref_path, ref_keypoints);
169     ROS_INFO("Reference keypoints written to: %s", ref_path.c_str());
170 }
171
172 cv::Mat readImage(std::string path) {
173     cv::Mat object;
174     if (color) {
175         object = cv::imread(path, CV_LOAD_IMAGE_COLOR);
176     } else {
177         object = cv::imread(path, CV_LOAD_IMAGE_GRAYSCALE);
178     }
179     return object;
180 }
181
182 bool setProcessRunningCallback(image_processor::setProcessRunning::Request &req
    , image_processor::setProcessRunning::Response &res) {
183     running = req.running;
184     return true;
185 }
186
187 bool getProcessRunningCallback(image_processor::getProcessRunning::Request &req
    , image_processor::getProcessRunning::Response &res) {
188     res.running = running;
189     return true;
190 }
191
192 bool setBinaryMatChingCallback(image_processor::setBinaryMatChing::Request &req
    , image_processor::setBinaryMatChing::Response &res) {
193     binary = req.binary;
194     return true;
195 }
196
197 bool getBinaryMatChingCallback(image_processor::getBinaryMatChing::Request &req
    , image_processor::getBinaryMatChing::Response &res) {
198     res.binary = binary;
```

```

199     return true;
200 }
201
202 bool setBruteForceMatchingCallback(image_processor::setBruteForceMatching::
    Request &req, image_processor::setBruteForceMatching::Response &res) {
203     bruteForce = req.bruteForce;
204     return true;
205 }
206
207 bool getBruteForceMatchingCallback(image_processor::getBruteForceMatching::
    Request &req, image_processor::getBruteForceMatching::Response &res) {
208     res.bruteForce = bruteForce;
209     return true;
210 }
211
212 bool setKeypointDetectorTypeCallback(image_processor::setKeypointDetectorType::
    Request &req, image_processor::setKeypointDetectorType::Response &res) {
213     DETECTOR_TYPE = req.type;
214     detector = openCVMatching.setKeypointsDetector(DETECTOR_TYPE);
215     detector->detect(object1, keypoints_object1);
216     writeReferenceImage(object1, keypoints_object1, ref_path1);
217     return true;
218 }
219
220 bool getKeypointDetectorTypeCallback(image_processor::getKeypointDetectorType::
    Request &req, image_processor::getKeypointDetectorType::Response &res) {
221     res.type = DETECTOR_TYPE;
222     return true;
223 }
224
225 bool setDescriptorTypeCallback(image_processor::setDescriptorType::Request &req
    , image_processor::setDescriptorType::Response &res) {
226     EXTRACTOR_TYPE = req.type;
227     extractor = openCVMatching.setDescriptorsExtractor(EXTRACTOR_TYPE, binary);
228     extractor->compute(object1, keypoints_object1, descriptor_object1);
229     return true;
230 }
231
232 bool getDescriptorTypeCallback(image_processor::getDescriptorType::Request &req
    , image_processor::getDescriptorType::Response &res) {
233     res.type = EXTRACTOR_TYPE;
234     return true;
235 }
236
237 bool setVideoColorCallback(image_processor::setVideoColor::Request &req,
    image_processor::setVideoColor::Response &res) {
238     color = req.color;
239     object1 = readImage(temp_path1);
240     keypoints_object1.clear();
241     descriptor_object1.release();
242     detectAndComputeReference(object1, keypoints_object1, descriptor_object1);
243     writeReferenceImage(object1, keypoints_object1, ref_path1);
244     return true;
245 }
246
247 bool getVideoColorCallback(image_processor::getVideoColor::Request &req,
    image_processor::getVideoColor::Response &res) {

```

```
248     res.color = color;
249     return true;
250 }
251
252 bool setVideoUndistortionCallBack(image_processor::setVideoUndistortion::
    Request &req, image_processor::setVideoUndistortion::Response &res) {
253     undistort = req.undistort;
254     return true;
255 }
256
257 bool getVideoUndistortionCallBack(image_processor::getVideoUndistortion::
    Request &req, image_processor::getVideoUndistortion::Response &res) {
258     res.undistort = undistort;
259     return true;
260 }
261
262 bool setMatchingImage1CallBack(image_processor::setMatchingImage1::Request &req
    , image_processor::setMatchingImage1::Response &res) {
263     temp_path1 = req.imagePath;
264     object1 = readImage(temp_path1);
265     keypoints_object1.clear();
266     descriptor_object1.release();
267     detectAndComputeReference(object1, keypoints_object1, descriptor_object1);
268     writeReferenceImage(object1, keypoints_object1, ref_path1);
269     return true;
270 }
271
272 bool setImageDepthCallBack(image_processor::setImageDepth::Request &req,
    image_processor::setImageDepth::Response &res) {
273     lambda = req.lambda;
274     return true;
275 }
```

object_2D_matcher.hpp

Listing B.2: Source file - code/image_processor/object_2D_matcher.hpp

```

1 //
2 // Original author: Asgeir Bjoerkedal. Created: 10.03.16. Last edit: 30.05.16.
3 //
4 // Main application for 2D object detection. Communicates via ROS and utilizes
5 // the methods defined in the header file
6 //
7 // Created as part of the software solution for a Master's thesis in Production
8 // Technology at NTNU Trondheim.
9 //
10 #ifndef IMAGE_PROCESSOR_OBJECT_2D_MATCHER_HPP
11 #define IMAGE_PROCESSOR_OBJECT_2D_MATCHER_HPP
12
13 #include <ros/package.h>
14 #include <geometry_msgs/Pose2D.h>
15 #include "image_processor/setProcessRunning.h"
16 #include "image_processor/getProcessRunning.h"
17 #include "image_processor/setBinaryMatching.h"
18 #include "image_processor/getBinaryMatching.h"
19 #include "image_processor/setKeypointDetectorType.h"
20 #include "image_processor/getKeypointDetectorType.h"
21 #include "image_processor/setDescriptorType.h"
22 #include "image_processor/getDescriptorType.h"
23 #include "image_processor/setVideoColor.h"
24 #include "image_processor/getVideoColor.h"
25 #include "image_processor/setBruteforceMatching.h"
26 #include "image_processor/getBruteforceMatching.h"
27 #include "image_processor/setVideoUndistortion.h"
28 #include "image_processor/getVideoUndistortion.h"
29 #include "image_processor/setMatchingImage1.h"
30 #include "image_processor/setImageDepth.h"
31 #include <image_transport/image_transport.h>
32 #include <cv_bridge/cv_bridge.h>
33
34 // Keypoint and descriptor type
35 std::string DETECTOR_TYPE = "SIFT";
36 std::string EXTRACTOR_TYPE = "SIFT";
37 // Resolution
38 const int VIDEO_WIDTH = 1280;
39 const int VIDEO_HEIGHT = 720;
40 // Path to camera parameters (K-matrix)
41 const std::string CAMERA_PARAMS = ros::package::getPath("image_processor") + "/"
42 // resources/calibration_reserve_camera.yml";
43 // Path to reference image storage
44 const std::string ref_path1 = ros::package::getPath("image_processor") + "/"
45 // resources/output/ref_keypoints1.jpg";
46 // Path to initial matching image
47 std::string temp_path1 = ros::package::getPath("image_processor") + "/resources
48 // /Lenna.png";
49 // Holds the object pose
50 geometry_msgs::Pose2D object_pose_msg;
51 // Homography method

```

```
48 int homographyMethod = CV_RANSAC; // CV_LMEDS
49 // Loop frequency
50 double FREQ = 60;
51
52 /*!
53 * \brief Initializes the object matcher image, resolution, detector and
  extractor.
54 * \param video_width The horizontal video resolution (pixel).
55 * \param video_height The vertical video resolution (pixel).
56 */
57 void initializeMatcher(const int video_width, const int video_height);
58
59 /*!
60 * \brief Detect and compute keypoints and descriptors for a given image matrix
  .
61 * \param object The query image.
62 * \param keypoints_object Reference to the keypoints storage object.
63 * \param descriptor_object Reference to the descriptor storage object.
64 */
65 void detectAndComputeReference(cv::Mat &object, std::vector<cv::KeyPoint> &
  keypoints_object, cv::Mat &descriptor_object);
66
67 /*!
68 * \brief Draws keypoints on a chosen image object and stores it to a desired
  file path.
69 * \param object The query image.
70 * \param keypoints_object The keypoints.
71 * \param ref_path The storage file path.
72 */
73 void writeReferenceImage(cv::Mat object, std::vector<cv::KeyPoint>
  keypoints_object, std::string ref_path);
74
75 /*!
76 * \brief Read an image from a desired file path.
77 * \param path The file path.
78 * \return The image matrix.
79 */
80 cv::Mat readImage(std::string path);
81
82 /*!
83 * \brief Callback method for toggling the object detection through ROS service
  .
84 * \param req The service request. True for image processed video stream. False
  for raw video stream.
85 * \param res The service response. Not in use.
86 */
87 bool setProcessRunningCallback(image_processor::setProcessRunning::Request &req
  , image_processor::setProcessRunning::Response &res);
88
89 /*!
90 * \brief Callback method for object detection running status through ROS
  service.
91 * \param req The service request.
92 * \param res The service response. Returns the state of the image processing.
  True if running. False otherwise.
93 */
94 bool getProcessRunningCallback(image_processor::getProcessRunning::Request &req
```

```

    , image_processor::getProcessRunning::Response &res);
95
96 /*!
97 * \brief Callback method for toggling of binary/non-binary matching through
ROS service.
98 * \param req The service request. True for matching of binary descriptors.
False for real-valued.
99 * \param res The service response. Not in use.
100 */
101 bool setBinaryMatchingCallBack(image_processor::setBinaryMatching::Request &req
    , image_processor::setBinaryMatching::Response &res);
102
103 /*!
104 * \brief Callback method for binary/non-binary matching status through ROS
service.
105 * \param req The service request.
106 * \param res The service response. Returns the state of the matching control
boolean.
107 */
108 bool getBinaryMatchingCallBack(image_processor::getBinaryMatching::Request &req
    , image_processor::getBinaryMatching::Response &res);
109
110 /*!
111 * \brief Callback method for toggling between bruteforce and FLANN matching
through ROS service.
112 * \param req The service request. True for bruteforce matching. False for
FLANN.
113 * \param res The service response. Not in use.
114 */
115 bool setBruteforceMatchingCallBack(image_processor::setBruteforceMatching::
    Request &req, image_processor::setBruteforceMatching::Response &res);
116
117 /*!
118 * \brief Callback method for bruteforce/FLANN matching status through ROS
service.
119 * \param req The service request.
120 * \param res The service response. Return the status of matching approach in
use.
121 */
122 bool getBruteforceMatchingCallBack(image_processor::getBruteforceMatching::
    Request &req, image_processor::getBruteforceMatching::Response &res);
123
124 /*!
125 * \brief Callback method for setting keypoint detector through ROS service.
126 * Sets the detector based on a string input. Detects keypoints in the query
image and outputs an image file with
127 the new keypoints.
128 * \param req The service request. String as an acronym for wanted detection, e
.g. SIFT, SURF, BRISK, ORB.
129 * \param res The service response. Not in use.
130 */
131 bool setKeypointDetectorTypeCallBack(image_processor::setKeypointDetectorType::
    Request &req, image_processor::setKeypointDetectorType::Response &res);
132
133 /*!
134 * \brief Callback method for getting the keypoint detector type through ROS
service.

```

```
135 * \param req The service request.
136 * \param res The service response. Return the keypoint detector in use.
137 */
138 bool getKeypointDetectorTypeCallBack(image_processor::getKeypointDetectorType::
    Request &req, image_processor::getKeypointDetectorType::Response &res);
139
140 /*!
141 * \brief Callback method for setting descriptor extractor through ROS service.
142 * \param req The service request. String as an acronym for wanted extractor, e
    .g. SIFT, SURF, BRISK, ORB.
143 * \param res The service response. Not in use.
144 */
145 bool setDescriptionTypeCallBack(image_processor::setDescriptionType::Request &req
    , image_processor::setDescriptionType::Response &res);
146
147 /*!
148 * \brief Callback method for getting descriptor extractor type through ROS
    service.
149 * Sets the extractor based on a string input. New descriptors are computed for
    the matching image.
150 * Further matching with the new descriptor can be performed instantaneously.
151 * \param req The service request.
152 * \param res The service response. Return the descriptor extractor in use.
153 */
154 bool getDescriptorTypeCallBack(image_processor::getDescriptorType::Request &req
    , image_processor::getDescriptorType::Response &res);
155
156 /*!
157 * \brief Callback method for setting color/grayscale video capture through ROS
    service.
158 * \param req The service request. True for color. False for grayscale.
159 * \param res The service response. Not in use.
160 */
161 bool setVideoColorCallBack(image_processor::setVideoColor::Request &req,
    image_processor::setVideoColor::Response &res);
162
163 /*!
164 * \brief Callback method for getting current video color mode through ROS
    service.
165 * \param req The service request.
166 * \param res The service response. Returns the color status of the video
    stream. True for color. False for grayscale.
167 */
168 bool getVideoColorCallBack(image_processor::getVideoColor::Request &req,
    image_processor::getVideoColor::Response &res);
169
170 /*!
171 * \brief Callback method for setting video undistortion of video stream
    through ROS service.
172 * Undistortion will use distortion parameters from .XML/.YAML file output from
    camera calibration.
173 * \param req The service request. True if correction for lens distortion.
    False for no correction.
174 * \param res The service response. Not in use.
175 */
176 bool setVideoUndistortionCallBack(image_processor::setVideoUndistortion::
    Request &req, image_processor::setVideoUndistortion::Response &res);
```



```
177
178 /*!
179 * \brief Callback method for getting undistortion status through ROS service.
180 * \param req The service request.
181 * \param res The service response. Get the status of lens correction.
182 */
183 bool getVideoUndistortionCallBack(image_processor::getVideoUndistortion::
    Request &req, image_processor::getVideoUndistortion::Response &res);
184
185 /*!
186 * \brief Callback method for setting the image to match with in the video
    scene through ROS service.
187 * Reads the new image, detects keypoints and computes descriptors, and outputs
    an image with keypoints.
188 * \param req The service request. Path as string to the new query image.
189 * \param res The service response. Not in use.
190 */
191 bool setMatchingImage1CallBack(image_processor::setMatchingImage1::Request &req
    , image_processor::setMatchingImage1::Response &res);
192
193 /*!
194 * \brief Callback method for setting the image depth (lambda), used for
    scaling the normalized image coordinates
195 * through ROS service.
196 * \param req The service request. Double value of distance from camera lens to
    object along the optical axis.
197 * \param res The service response. Not in use.
198 */
199 bool setImageDepthCallBack(image_processor::setImageDepth::Request &req,
    image_processor::setImageDepth::Response &res);
200
201
202 #endif //IMAGE_PROCESSOR_OBJECT_2D_MATCHER_HPP
```

openCV_matching.cpp

Listing B.3: Source file - code/image_processor/openCV_matching.cpp

```
1 //
2 // Original author: Asgeir Bjoerkedal. Created: 10.03.16. Last edit: 30.05.16.
3 //
4 // The class implements methods from OpenCV and is designed for use in an
5 // object detection application.
6 // It encompasses capturing of video frames, processing video frames by
7 // numerous keypoint
8 // detectors and descriptor extractors, matching algorithms, visualization and
9 // computation
10 // of object image coordinates and orientation.
11 //
12 // Created as part of the software solution for a Master's thesis in Production
13 // Technology at NTNU Trondheim.
14 //
15 #include "../include/image_processor/openCV_matching.hpp"
16
17 namespace robotcam
18 {
19     cv::Mat OpenCVMatching::getCameraMatrix(const std::string path) {
20         cv::Mat temp;
21         cv::FileStorage fs(path, cv::FileStorage::READ);
22         fs["camera_matrix"] >> temp;
23         fs.release();
24         return temp;
25     }
26
27     cv::Mat OpenCVMatching::getDistortionCoeff(const std::string path) {
28         cv::Mat temp;
29         cv::FileStorage fs(path, cv::FileStorage::READ);
30         fs["distortion_coefficients"] >> temp;
31         fs.release();
32         return temp;
33     }
34
35     std::string OpenCVMatching::type2str(int type) {
36         std::string r;
37         uchar depth = type & CV_MAT_DEPTH_MASK;
38         uchar chans = 1 + (type >> CV_CN_SHIFT);
39         switch (depth) {
40             case CV_8U:
41                 r = "8U";
42                 break;
43             case CV_8S:
44                 r = "8S";
45                 break;
46             case CV_16U:
47                 r = "16U";
48                 break;
49             case CV_16S:
50                 r = "16S";
51                 break;
52             case CV_32S:
```

```

49         r = "32S";
50         break;
51     case CV_32F:
52         r = "32F";
53         break;
54     case CV_64F:
55         r = "64F";
56         break;
57     default:
58         r = "User";
59         break;
60     }
61     r += "C";
62     r += (chans + '0');
63     // USAGE
64     //     std::string ty = type2str( H.type() );
65     //     printf("Matrix: %s %d x %d \n", ty.c_str(), H.cols, H.rows );
66     return r;
67 }
68
69 cv::Mat OpenCVMatching::captureFrame( bool color, bool useCalibration, cv::
VideoCapture capture, cv::Mat cameraMatrix, cv::Mat distCoeffs) {
70     cv::Mat inFrame, outFrame;
71     capture >> inFrame;
72     if (color == false && useCalibration == false) {
73         cv::cvtColor(inFrame, outFrame, CV_RGB2GRAY); // grayscale
74     } else if (color == false && useCalibration == true) {
75         cv::Mat temp;
76         cv::undistort(inFrame, temp, cameraMatrix, distCoeffs);
77         cv::cvtColor(temp, outFrame, CV_RGB2GRAY); // grayscale
78     } else if (color == true && useCalibration == false) {
79         outFrame = inFrame;
80     } else {
81         cv::undistort(inFrame, outFrame, cameraMatrix, distCoeffs);
82     }
83     return outFrame;
84 }
85
86 cv::Mat OpenCVMatching::captureFrame( bool color, cv::VideoCapture capture)
87 {
88     cv::Mat inFrame, outFrame;
89     capture >> inFrame;
90     if(color) {
91         outFrame = inFrame;
92     } else {
93         cv::cvtColor(inFrame, outFrame, CV_RGB2GRAY);
94     }
95     return outFrame;
96 }
97
98 std::vector<cv::DMatch> OpenCVMatching::knnMatchDescriptors( cv::Mat
descriptors_object, cv::Mat descriptors_scene, float nnratio) {
99     cv::FlannBasedMatcher matcher;
100     std::vector<std::vector<cv::DMatch> > matches;
101     // Find the 2 best descriptor matches
102     matcher.knnMatch(descriptors_object, descriptors_scene, matches, 2);
103     // Ratio test the matches

```

```

103     std::vector<cv::DMatch> good_matches;
104     good_matches.reserve(matches.size());
105     for (size_t i = 0; i < matches.size(); ++i) {
106         if (matches[i].size() < 2) continue;
107         const cv::DMatch &m1 = matches[i][0];
108         const cv::DMatch &m2 = matches[i][1];
109         if (m1.distance <= nnratio * m2.distance) good_matches.push_back(m1
110             );
111     }
112     return good_matches;
113 }
114
115 std::vector<cv::DMatch> OpenCVMatching::knnMatchDescriptorsLSH(cv::Mat
116     descriptors_object, cv::Mat descriptors_scene, float nndrRatio) {
117     cv::FlannBasedMatcher matcher(new cv::flann::LshIndexParams(20, 10, 2))
118     ;
119     std::vector<std::vector<cv::DMatch> > matches;
120     // Find the 2 best descriptor matches
121     matcher.knnMatch(descriptors_object, descriptors_scene, matches, 2);
122     // Ratio test the matches
123     std::vector<cv::DMatch> good_matches;
124     good_matches.reserve(matches.size());
125     for (size_t i = 0; i < matches.size(); ++i) {
126         if (matches[i].size() < 2) continue;
127         const cv::DMatch &m1 = matches[i][0];
128         const cv::DMatch &m2 = matches[i][1];
129         if (m1.distance <= nndrRatio * m2.distance) good_matches.push_back(
130             m1);
131     }
132     return good_matches;
133 }
134
135 std::vector<cv::DMatch> OpenCVMatching::matchDescriptors(cv::Mat
136     descriptors_object, cv::Mat descriptors_scene) {
137     cv::FlannBasedMatcher matcher;
138     std::vector<cv::DMatch> matches;
139     // Match descriptors
140     matcher.match(descriptors_object, descriptors_scene, matches);
141     // Compute the max and min distance of the matches in current
142     // videoFrame
143     double max_dist = 0;
144     double min_dist = 100;
145     for (int i = 0; i < descriptors_object.rows; i++) {
146         double dist = matches[i].distance;
147         if (dist < min_dist) min_dist = dist;
148         if (dist > max_dist) max_dist = dist;
149     }
150     // Filter out the good matches
151     std::vector<cv::DMatch> good_matches;
152     double k = 2;
153     for (int i = 0; i < descriptors_object.rows; i++) {
154         if (matches[i].distance <= cv::max(k * min_dist, 0.02)) {
155             good_matches.push_back(matches[i]);
156         }
157     }
158     return good_matches;
159 }

```

```

154
155     std::vector<cv::DMatch> OpenCVMatching::bruteForce(cv::Mat
156     descriptors_object, cv::Mat descriptors_scene, int normType) {
157         cv::BFMatcher matcher(normType);
158         std::vector<std::vector<cv::DMatch> > matches;
159         // Find the 2 best descriptor matches
160         matcher.knnMatch(descriptors_object, descriptors_scene, matches, 2);
161         // Ratio test the matches
162         std::vector<cv::DMatch> good_matches;
163         for (int i = 0; i < matches.size(); ++i) {
164             const float ratio = 0.9; // 0.8 in Lowe's paper on SIFT. Can be
165             tuned
166             if (matches[i][0].distance < ratio * matches[i][1].distance) {
167                 good_matches.push_back(matches[i][0]);
168             }
169         }
170         return good_matches;
171     }
172
173     cv::Ptr<cv::Feature2D> OpenCVMatching::setKeyPointsDetector(std::string
174     typeKeyPoint) {
175         cv::Ptr<cv::Feature2D> detector;
176         if (typeKeyPoint == "SURF") {
177             detector = cv::xfeatures2d::SURF::create(1000,4,5,false,false);
178             ROS_INFO("Keypoint detector: %s", typeKeyPoint.c_str());
179         } else if (typeKeyPoint == "SIFT") {
180             detector = cv::xfeatures2d::SIFT::create(0,5,0.04,10,1.6);
181             ROS_INFO("Keypoint detector: %s", typeKeyPoint.c_str());
182         } else if (typeKeyPoint == "STAR") {
183             detector = cv::xfeatures2d::StarDetector::create(45,30,10,8,5);
184             ROS_INFO("Keypoint detector: %s", typeKeyPoint.c_str());
185         } else if (typeKeyPoint == "BRISK") {
186             detector = cv::BRISK::create(30,3,1.0f);
187             ROS_INFO("Keypoint detector: %s", typeKeyPoint.c_str());
188         } else if (typeKeyPoint == "FAST") {
189             detector = cv::FastFeatureDetector::create(10,true,cv::
190             FastFeatureDetector::TYPE_9_16);
191             ROS_INFO("Keypoint detector: %s", typeKeyPoint.c_str());
192         } else if (typeKeyPoint == "ORB") {
193             detector = cv::ORB::create(1000,1.2f,8,31,0,2,cv::ORB::FAST_SCORE
194             ,31,20);
195             ROS_INFO("Keypoint detector: %s", typeKeyPoint.c_str());
196         } else if (typeKeyPoint == "AKAZE") {
197             detector = cv::AKAZE::create(cv::AKAZE::DESCRIPTOR_MLDB,0,3,0.001f
198             ,4,4,cv::KAZE::DIFF_PM_G2);
199             ROS_INFO("Keypoint detector: %s", typeKeyPoint.c_str());
200         } else {
201             ROS_ERROR("Could not find keypoint detector: %s\n\tChoosing default
202             : SURF", typeKeyPoint.c_str());
203             detector = cv::xfeatures2d::SURF::create(1000);
204         }
205         return detector;
206     }
207
208     cv::Ptr<cv::Feature2D> OpenCVMatching::setDescriptorsExtractor(std::string
209     typeDescriptor, bool &binary) {
210         cv::Ptr<cv::Feature2D> extractor;

```

```

203     if (typeDescriptor == "SURF") {
204         binary = false;
205         extractor = cv::xfeatures2d::SURF::create(1000,4,5,false,false);
206         ROS_INFO("Descriptor: %s", typeDescriptor.c_str());
207         ROS_INFO("Binary matching: %d", binary);
208     } else if (typeDescriptor == "SIFT") {
209         binary = false;
210         extractor = cv::xfeatures2d::SIFT::create(0,5,0.04,10,1.6);
211         ROS_INFO("Descriptor: %s", typeDescriptor.c_str());
212         ROS_INFO("Binary matching: %d", binary);
213     } else if (typeDescriptor == "BRISK") {
214         binary = true;
215         extractor = cv::BRISK::create(30,3,1.0f);
216         ROS_INFO("Descriptor: %s", typeDescriptor.c_str());
217         ROS_INFO("Binary matching: %d", binary);
218     } else if (typeDescriptor == "FREAK") {
219         binary = true;
220         extractor = cv::xfeatures2d::FREAK::create(true,true,22.0f,4);
221         ROS_INFO("Descriptor: %s", typeDescriptor.c_str());
222         ROS_INFO("Binary matching: %d", binary);
223     } else if (typeDescriptor == "ORB") {
224         binary = true;
225         extractor = cv::ORB::create(1000,1.2f,8,31,0,2,cv::ORB::FAST_SCORE
226             ,31,20); // WTA_K = 3-4 -> HAMMING2
227         ROS_INFO("Descriptor: %s", typeDescriptor.c_str());
228         ROS_INFO("Binary matching: %d", binary);
229     } else if (typeDescriptor == "AKAZE") {
230         binary = true;
231         extractor = cv::AKAZE::create(cv::AKAZE::DESCRIPTOR_MLDB,0,3,0.001f
232             ,4,4,cv::KAZE::DIFF_PM_G2);
233         ROS_INFO("Descriptor: %s", typeDescriptor.c_str());
234         ROS_INFO("Binary matching: %d", binary);
235     } else if (typeDescriptor == "BRIEF") {
236         binary = true;
237         extractor = cv::xfeatures2d::BriefDescriptorExtractor::create(32,
238             true);
239         ROS_INFO("Descriptor: %s", typeDescriptor.c_str());
240         ROS_INFO("Binary matching: %d", binary);
241     } else {
242         binary = false;
243         ROS_ERROR("Could not find keypoint detector: %s\n\tChoosing default
244             descriptor: SURF", typeDescriptor.c_str());
245         extractor = cv::xfeatures2d::SURF::create(1000);
246     }
247     return extractor;
248 }
249
250 CurrentMatch OpenCVMatching::visualizedMatch(cv::Mat searchImage, cv::Mat
251     objectImage, std::vector<cv::KeyPoint> keypointsObject, std::vector<
252     cv::KeyPoint> keypointsScene, std::vector<cv::DMatch> good_matches,
253     bool showKeypoints, int homographyType) {
254     cv::Mat image_matches;
255     if (showKeypoints) {
256         cv::drawKeypoints(searchImage, keypointsScene, image_matches,
257             CV_RGB(0,0,255), cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);
258     } else {
259         image_matches = searchImage.clone();

```

```

252     }
253     std::vector<cv::Point2f> obj;
254     std::vector<cv::Point2f> scene;
255     for (size_t i = 0; i < good_matches.size(); i++) {
256         // Retrieve the keypoints from good matches
257         obj.push_back(keypointsObject[good_matches[i].queryIdx].pt);
258         scene.push_back(keypointsScene[good_matches[i].trainIdx].pt);
259     }
260     // Perform Homography to find a perspective transformation between two
261     // planes.
262     cv::Mat H;
263     if (!obj.size() == 0 && !scene.size() == 0) {
264         H = cv::findHomography(obj, scene, homographyType); // CV_LMEDS //
265         CV_RANSAC
266     }
267     // Put object corners in a vector
268     std::vector<cv::Point2f> objectCorners(4);
269     objectCorners[0] = cvPoint(0, 0); //Upper left corner
270     objectCorners[1] = cvPoint(objectImage.cols, 0); //Upper right corner
271     objectCorners[2] = cvPoint(objectImage.cols, objectImage.rows); //Lower
272     // right corner
273     objectCorners[3] = cvPoint(0, objectImage.rows); //Lower left corner
274     // Find the corresponding object corners in the scene perspective
275     std::vector<cv::Point2f> sceneCorners(4);
276     if (!H.rows == 0 && !H.cols == 0) {
277         cv::perspectiveTransform(objectCorners, sceneCorners, H);
278         if (checkObjectInnerAngles(sceneCorners, 60, 120)) {
279             // Draw lines surrounding the object
280             cv::line(image_matches, sceneCorners[0], sceneCorners[1], cv::
281             Scalar(0, 255, 0), 2); //TOP line
282             cv::line(image_matches, sceneCorners[1], sceneCorners[2], cv::
283             Scalar(0, 255, 0), 2); //RIGHT line
284             cv::line(image_matches, sceneCorners[2], sceneCorners[3], cv::
285             Scalar(0, 255, 0), 2); //BOTTOM line
286             cv::line(image_matches, sceneCorners[3], sceneCorners[0], cv::
287             Scalar(0, 255, 0), 2); //LEFT line
288             // Draw diagonals
289             cv::line(image_matches, sceneCorners[0], sceneCorners[2], cv::
290             Scalar(0, 255, 0), 1); //DIAGONAL 0-2
291             cv::line(image_matches, sceneCorners[1], sceneCorners[3], cv::
292             Scalar(0, 255, 0), 1); //DIAGONAL 1-3
293             // Center
294             cv::Point2f cen(0.0, 0.0);
295             if (intersection(sceneCorners[0], sceneCorners[2], sceneCorners
296             [1], sceneCorners[3], cen)) {
297                 cv::circle(image_matches, cen, 10, cv::Scalar(0, 0, 255),
298                 2);
299             }
300         }
301     }
302     // Draw circles in center pixel of the video stream
303     if (searchImage.rows > 60 && searchImage.cols > 60) {
304         cv::circle(image_matches, cv::Point(searchImage.cols / 2,
305         searchImage.rows / 2), 5, CV_RGB(255, 0, 0));
306         cv::circle(image_matches, cv::Point(searchImage.cols / 2,
307         searchImage.rows / 2), 10, CV_RGB(0, 255, 0));
308         cv::circle(image_matches, cv::Point(searchImage.cols / 2,

```

```

        searchImage.rows / 2), 15, CV_RGB(0, 0, 255));
296     }
297     CurrentMatch cm;
298     cm.outFrame = image_matches;
299     cm.sceneCorners = sceneCorners;
300     return cm;
301 }
302
303 bool OpenCVMatching::intersection(cv::Point2f o1, cv::Point2f p1, cv::
Point2f o2, cv::Point2f p2, cv::Point2f &r) {
304     // The lines are defined by (o1, p1) and (o2, p2).
305     cv::Point2f x = o2 - o1;
306     cv::Point2f d1 = p1 - o1;
307     cv::Point2f d2 = p2 - o2;
308     float cross = d1.x * d2.y - d1.y * d2.x;
309     if (fabsf(cross) < /*EPS*/1e-8) return false;
310     double t1 = (x.x * d2.y - x.y * d2.x) / cross;
311     r = o1 + d1 * t1;
312     return true;
313 }
314
315 int OpenCVMatching::innerAngle(cv::Point2f a, cv::Point2f b, cv::Point2f c)
{
316     cv::Point2f ab(b.x - a.x, b.y - a.y);
317     cv::Point2f cb(b.x - c.x, b.y - c.y);
318     double dot = (ab.x * cb.x + ab.y * cb.y); // dot product
319     double cross = (ab.x * cb.y - ab.y * cb.x); // cross product
320     double alpha = atan2(cross, dot);
321     int angle = (int) floor(alpha * 180. / PI + 0.5);
322     return abs(angle);
323 }
324
325 bool OpenCVMatching::checkObjectInnerAngles(std::vector<cv::Point2f>
scorner, int min, int max) {
326     bool out = false;
327     int c0 = innerAngle(scorner[3], scorner[0], scorner[1]);
328     int c1 = innerAngle(scorner[0], scorner[1], scorner[2]);
329     int c2 = innerAngle(scorner[1], scorner[2], scorner[3]);
330     int c3 = innerAngle(scorner[2], scorner[3], scorner[0]);
331     if (c0 > min && c0 < max && c1 > min && c1 < max && c2 > min && c2 <
max && c3 > min && c3 < max) out = true;
332     return out;
333 }
334
335 double OpenCVMatching::getXoffset(cv::Mat frame, std::vector<cv::Point2f>
scorner) {
336     cv::Point2f cen;
337     double xoffset = 0.0;
338     if (intersection(scorner[0], scorner[2], scorner[1], scorner[3], cen))
339     {
340         xoffset = cen.x - frame.cols / 2;
341     }
342     return xoffset;
343 }
344
345 double OpenCVMatching::getYoffset(cv::Mat frame, std::vector<cv::Point2f>
scorner) {

```



```

345     cv::Point2f cen;
346     double yOffset = 0.0;
347     if (intersection(scorner[0], sccorner[2], sccorner[1], sccorner[3], cen))
348     {
349         yOffset = cen.y - frame.rows / 2;
350     }
351     return yOffset;
352 }
353
354 double OpenCVMatching::getXpos(std::vector<cv::Point2f> sccorner) {
355     cv::Point2f cen;
356     intersection(scorner[0], sccorner[2], sccorner[1], sccorner[3], cen);
357     double x = cen.x;
358     return x;
359 }
360
361 double OpenCVMatching::getYpos(std::vector<cv::Point2f> sccorner) {
362     cv::Point2f cen;
363     intersection(scorner[0], sccorner[2], sccorner[1], sccorner[3], cen);
364     double y = cen.y;
365     return y;
366 }
367
368 double OpenCVMatching::getObjectAngle(cv::Mat frame, std::vector<cv::
369     Point2f> sccorner) {
370     double centerX = frame.cols / 2;
371     double diffX = centerX - sccorner[1].x;
372     double x = (centerX - diffX) - sccorner[0].x;
373     double y = sccorner[0].y - sccorner[1].y;
374     double angle = atan2(y, x) * 180 / PI;
375     return angle;
376 }
377
378 Eigen::Vector3d OpenCVMatching::getNormImageCoords(double x, double y,
379     double lambda, cv::Mat camera_matrix) {
380     Eigen::Vector3d pixelCoords;
381     Eigen::Vector3d normCoords;
382     Eigen::Matrix3d camMat;
383     camMat << camera_matrix.at<double>(0,0),0,camera_matrix.at<double>(0,2)
384     ,
385     0,camera_matrix.at<double>(1,1),camera_matrix.at<double>(1,2)
386     ,
387     0,0,1;
388     pixelCoords(0) = x;
389     pixelCoords(1) = y;
390     pixelCoords(2) = 1;
391     Eigen::Matrix3d icamMat = camMat.inverse();
392     normCoords = icamMat*pixelCoords;
393     return lambda*normCoords;
394 }

```

openCV_matching.hpp

Listing B.4: Source file - code/image_processor/openCV_matching.hpp

```
1 //
2 // Original author: Asgeir Bjoerkedal. Created: 10.03.16. Last edit: 30.05.16.
3 //
4 // The class implements methods from OpenCV and is designed for use in an
5 // object detection application.
6 // It encompasses capturing of video frames, processing video frames by
7 // numerous keypoint
8 // detectors and descriptor extractors, matching algorithms, visualization and
9 // computation
10 // of object image coordinates and orientation.
11 //
12 // Created as part of the software solution for a Master's thesis in Production
13 // Technology at NTNU Trondheim.
14 //
15 #ifndef IMAGE_PROCESSOR_OPENCV_MATCHING_HPP
16 #define IMAGE_PROCESSOR_OPENCV_MATCHING_HPP
17
18 #include <iostream>
19 #include <math.h>
20 #include <ros/ros.h>
21 #include "opencv2/core.hpp"
22 #include "opencv2/imgcodecs.hpp"
23 #include "opencv2/highgui.hpp"
24 #include "opencv2/features2d.hpp"
25 #include "opencv2/calib3d.hpp"
26 #include "opencv2/imgproc.hpp"
27 #include "opencv2/xfeatures2d.hpp"
28 #include <eigen3/Eigen/Dense>
29
30 #define PI 3.14159265
31
32 namespace robotcam {
33
34     struct CurrentMatch {
35         /*! The frame with visualized keypoints and matching. */
36         cv::Mat outFrame;
37         /*! The corners of the matched object in the scene. */
38         std::vector<cv::Point2f> sceneCorners;
39     };
40
41     class OpenCVMatching {
42     public:
43         /*!
44          * \brief Get a camera matrix from XML or YAML file.
45          * \param path The path of the file.
46          * \return The camera matrix.
47          */
48         cv::Mat getCameraMatrix(const std::string path);
49
50         /*!
51          * \brief Get the distortion coefficients from XML or YAML file.
52          * \param path The path of the file.
53          */
54     };
55 }
```

```

49     * \return The distortion coefficients.
50     */
51     cv::Mat getDistortionCoeff(const std::string path);
52
53     /*!
54     * \brief Check the actual type openCV cv::Mat.
55     * \param type The type of a matrix.
56     * \return The matrix type as string.
57     */
58     std::string type2str(int type);
59
60     /*!
61     * \brief Capture a frame from a connected web camera.
62     * \param color True for RGB. False for grayscale.
63     * \param undistort True for correction for lens distortion. False for
64     *       no correction.
65     * \param cameraMatrix The camera matrix (K-matrix) of the web camera.
66     * \param distCoeffs The distortion coefficients of the web camera.
67     * \return The current video frame.
68     *
69     * Capture a frame in color/grayscale and with or without lens
70     *       distortion.
71     */
72     cv::Mat captureFrame(bool color, bool undistort, cv::VideoCapture
73     capture, cv::Mat cameraMatrix, cv::Mat distCoeffs);
74
75     /*!
76     * \brief Capture a frame from a connected web camera.
77     * \param color The boolean determining RGB or grayscale video frame.
78     * \param capture The object capturing the video stream from the camera
79     *
80     * Capture either with color or grayscale.
81     */
82     cv::Mat captureFrame(bool color, cv::VideoCapture capture);
83
84     /*!
85     * \brief Flann based nearest neighbour matching.
86     * \param descriptors_object The descriptors of the query image.
87     * \param descriptors_scene The descriptors of the training scene image
88     *
89     * \param nnratio The nearest neighbour ratio for distance filtering.
90     * \return The good matches.
91     */
92     std::vector<cv::DMatch> knnMatchDescriptors(cv::Mat descriptors_object,
93     cv::Mat descriptors_scene, float nnratio);
94
95     /*!
96     * \brief Flann based nearest neighbour with LSH index for binary
97     *       matching.
98     * \param descriptors_object The descriptors of the query image.
99     * \param descriptors_scene The descriptors of the training scene image
100    *
101    * \param nndrRatio The nearest neighbour ratio for distance filtering.
102    * \return The good matches.

```

```

98     */
99     std::vector<cv::DMatch> knnMatchDescriptorsLSH(cv::Mat
100         descriptors_object, cv::Mat descriptors_scene, float nndrRatio);
101
102     /*!
103     * \brief Flann based matching.
104     * \param descriptors_object The descriptors of the query image.
105     * \param descriptors_scene The descriptors of the training scene image
106     *
107     * \return The good matches.
108     */
109     std::vector<cv::DMatch> matchDescriptors(cv::Mat descriptors_object,
110         cv::Mat descriptors_scene);
111
112     /*!
113     * \brief Bruteforce nearest neighbour matching.
114     * \param descriptors_object The descriptors of the query image.
115     * \param descriptors_scene The descriptors of the training scene image
116     *
117     * \param normType The distance type, e.g. NORM_L1, NORM_L2,
118     *       NORM_HAMMING.
119     */
120     std::vector<cv::DMatch> bruteForce(cv::Mat descriptors_object, cv::Mat
121         descriptors_scene, int normType);
122
123     /*!
124     * \brief Set a keypoint detector based on a input string.
125     * \param typeKeyPoint The input string as an acronym for wanted
126     *       algorithm, e.g. SIFT, SURF.
127     * \return The keypoint detector.
128     */
129     cv::Ptr<cv::Feature2D> setKeyPointsDetector(std::string typeKeyPoint);
130
131     /*!
132     * \brief Set a descriptor extractor based on a input string.
133     * \param typeDescriptor The input string as an acronym for wanted
134     *       algorithm, e.g. SIFT, SURF.
135     * \param binary Reference to a matching control boolean. True if real-
136     *       valued descriptor, False if binary.
137     * \return The descriptor extractor.
138     */
139     cv::Ptr<cv::Feature2D> setDescriptorsExtractor(std::string
140         typeDescriptor, bool &binary);
141
142     /*!
143     * \brief Visualize a object matching using homography.
144     * \param searchImage The training scene image.
145     * \param objectImage The query image.
146     * \param keypointsObject The keypoints of the query image.
147     * \param keypointsScene The keypoints of the training scene image.
148     * \param good_matches The good matches between query and training
149     *       image.
150     * \param showKeypoints True for visualized keypoints. False for no
151     *       drawn keypoints.
152     * \param homographyType The homography type, e.g. CV_RANSAC or
153     *       CV_LMEDS.
154     * \return The current match holding an image with visualized matching

```

```

142         and the object corners in training scene.
143     */
CurrentMatch visualizedMatch(cv::Mat searchImage, cv::Mat objectImage,
    std::vector<cv::KeyPoint> keypointsObject, std::vector<cv::KeyPoint
    > keypointsScene, std::vector<cv::DMatch> good_matches, bool
    showKeypoints, int homographyType);

144
145     /*!
146     * \brief Check if the inner angles of a square or rectangle is within
147     * \param scorners The training scene corners of the matched object.
148     * \param min The minimum angle in degrees.
149     * \param max The maximum angle in degrees.
150     * \return True if angle is within min and max. False otherwise.
151     */
152     bool checkObjectInnerAngles(std::vector<cv::Point2f> scorners, int min,
    int max);

153
154     /*!
155     * \brief Get the pixel offset in x-direction of the matched object
156     * \param frame The training scene image.
157     * \param scorners The scene corners of the matched object.
158     * \return The object offset in x-direction.
159     */
160     double getXoffset(cv::Mat frame, std::vector<cv::Point2f> scorners);
161
162     /*!
163     * \brief Get the pixel offset in y-direction of the matched object
164     * \param frame The training scene image.
165     * \param scorners The scene corners of the matched object.
166     * \return The object pixel offset in y-direction.
167     */
168     double getYoffset(cv::Mat frame, std::vector<cv::Point2f> scorners);
169
170     /*!
171     * \brief Get the pixel coordinate x of the matched object center.
172     * \param scorners The scene corners of the matched object.
173     * \return The pixel coordinate x.
174     */
175     double getXpos(std::vector<cv::Point2f> scorners);
176
177     /*!
178     * \brief Get the pixel coordinate y of the matched object center.
179     * \param scorners The scene corners of the matched object.
180     * \return The pixel coordinate y.
181     */
182     double getYpos(std::vector<cv::Point2f> scorners);
183
184     /*!
185     * \brief Get the angle of in-plane rotation of the matched object.
186     * \param frame The training scene image.
187     * \param scorners The scene corners of the matched object.
188     * \return The object angle in degrees.
189     */
190     double getObjectAngle(cv::Mat frame, std::vector<cv::Point2f> scorners);

```

```
191
192     /*!
193     * \brief Get the normalized image coordinates of the matched object
194       scaled with lambda.
195     * \param x The pixel coordinate x.
196     * \param y The pixel coordinate y.
197     * \param lambda The depth to object along optical axis from camera
198       lens.
199     * \param camera_matrix The K-matrix of the camera.
200     * \return The normalized image coordinates scaled with lambda.
201     */
202     Eigen::Vector3d getNormImageCoords(double x, double y, double lambda,
203       cv::Mat camera_matrix);
204
205 private:
206     /*!
207     * \brief Get the intersection point of two lines.
208     * \param o1 The origin point of the first line.
209     * \param p1 The end point of the first line.
210     * \param o2 The origin point of the second line.
211     * \param p2 The end point of the second line.
212     * \param r The intersection point referenced.
213     * \return The boolean whether an intersection was found. True if found
214       . False otherwise.
215     */
216     bool intersection(cv::Point2f o1, cv::Point2f p1, cv::Point2f o2, cv::
217       Point2f p2, cv::Point2f &r);
218
219     /*!
220     * \brief Get the inner angle using three points.
221     * \param a The first point.
222     * \param b The origin of the angle.
223     * \param c The second point.
224     * \return The angle in degrees.
225     */
226     int innerAngle(cv::Point2f a, cv::Point2f b, cv::Point2f c);
227 };
228 }
229 #endif //IMAGE_PROCESSOR_OPENCV_MATCHING_HPP
```


Appendix C: The `agilus_planner` Application

This appendix contains the source code for the `agilus_planner` application.

robot_movement.cpp - This class advertises the services used for robotic manipulator control.

Pose.srv - This is the `.srv` file that is used to define the pose service object used to control the robotic manipulators.

robot_movement.cpp

Listing C.1: Source file - `code/agilus_planner/robot_movement.cpp`

```
1 //
2 // Original author: Adam Leon Kleppe. Last edit by: Asgeir Bjoerkedal at
3 // 30.05.16.
4 // A ROS node advertising the trajectory planning and execution, as defined in
5 // robot_planning_execution.hpp,
6 // as ROS services for simple interfacing with other ROS nodes.
7 //
8 #include "agilus_planner/Pose.h"
9 #include "../include/agilus_planner/robot_planning_execution.hpp"
10
11 ih::RobotPlanningExecution *robot;
12
13 /*!
14 * \brief Callback method for planning of a trajectory. The plan will only be
15 * visualized in MoveIt!.
16 * \param req The service request. Set the desired pose of the manipulator.
17 * \param res The service response. Returns the fraction of the trajectory
18 * which is feasible.
19 */
20 bool planPoseService(agilus_planner::Pose::Request &req, agilus_planner::Pose::
21 Response &res) {
22     if ((bool) !req.relative) {
23         if ((bool) req.set_position && (bool) !req.set_orientation) {
24             res.progress = robot->planPoseByXYZ(
25                 (double) req.position_x, (double) req.position_y, (double)
26                 req.position_z);
27         }
28         if ((bool) !req.set_position && (bool) req.set_orientation) {
29             res.progress = robot->planPoseByRPY(
30                 (double) req.orientation_r, (double) req.orientation_p, (
31                 double) req.orientation_y);
32         }
33     }
34 }
```



```

27     if ((bool) req.set_position && (bool) req.set_orientation) {
28         res.progress = robot->planPoseByXYZRPY(
29             (double) req.position_x, (double) req.position_y, (double)
30             req.position_z,
31             (double) req.orientation_r, (double) req.orientation_p, (
32             double) req.orientation_y);
33     }
34     else {
35         if ((bool) req.set_position && (bool) !req.set_orientation) {
36             res.progress = robot->planRelativePoseByXYZ(
37                 (double) req.position_x, (double) req.position_y, (double)
38                 req.position_z);
39         }
40         if ((bool) !req.set_position && (bool) req.set_orientation) {
41             res.progress = robot->planRelativePoseByRPY(
42                 (double) req.orientation_r, (double) req.orientation_p, (
43                 double) req.orientation_y);
44         }
45         if ((bool) req.set_position && (bool) req.set_orientation) {
46             res.progress = robot->planRelativePoseByXYZRPY(
47                 (double) req.position_x, (double) req.position_y, (double)
48                 req.position_z,
49                 (double) req.orientation_r, (double) req.orientation_p, (
50                 double) req.orientation_y);
51         }
52     }
53 }
54
55 /*!
56 * \brief Callback method for planning and execution of a trajectory. The
57 trajectory be executed.
58 * \param req The service request. Set the desired pose of the manipulator.
59 * \param res The service response. Returns the fraction of the trajectory
60 which is feasible.
61 */
62 bool goToPoseService(agilus_planner::Pose::Request &req, agilus_planner::Pose::
63 Response &res) {
64     if ((bool) !req.relative) {
65         if ((bool) req.set_position && (bool) !req.set_orientation) {
66             res.progress = robot->goToPoseByXYZ(
67                 (double) req.position_x, (double) req.position_y, (double)
68                 req.position_z);
69         }
70         if ((bool) !req.set_position && (bool) req.set_orientation) {
71             res.progress = robot->goToPoseByRPY(
72                 (double) req.orientation_r, (double) req.orientation_p, (
73                 double) req.orientation_y);
74         }
75         if ((bool) req.set_position && (bool) req.set_orientation) {
76             res.progress = robot->goToPoseByXYZRPY(
77                 (double) req.position_x, (double) req.position_y, (double)
78                 req.position_z,
79                 (double) req.orientation_r, (double) req.orientation_p, (
80                 double) req.orientation_y);
81         }
82     }
83 }

```

```

71     else {
72         if ((bool) req.set_position && (bool) !req.set_orientation) {
73             res.progress = robot->goToRelativePoseByXYZ(
74                 (double) req.position_x, (double) req.position_y, (double)
                    req.position_z);
75         }
76         if ((bool) !req.set_position && (bool) req.set_orientation) {
77             res.progress = robot->goToRelativePoseByRPY(
78                 (double) req.orientation_r, (double) req.orientation_p, (
                    double) req.orientation_y);
79         }
80         if ((bool) req.set_position && (bool) req.set_orientation) {
81             res.progress = robot->goToRelativePoseByXYZRPY(
82                 (double) req.position_x, (double) req.position_y, (double)
                    req.position_z,
83                 (double) req.orientation_r, (double) req.orientation_p, (
                    double) req.orientation_y);
84         }
85     }
86 }
87
88 int main(int argc, char **argv) {
89     ros::init(argc, argv, "robot_movement_service");
90     ros::NodeHandle node_handle("~");
91
92     // Initializing robot arguments used if ROS server has no parameters
93     std::string group_name = "manipulator";
94     double max_vel_scale_factor = 0.1;
95     int planning_time = 10;
96     int num_planning_attempts = 5;
97     int options = 2;
98
99     // Get or set group_name
100    if (node_handle.getParam("group_name")) {
101        node_handle.getParam("group_name", group_name);
102        ROS_INFO("Got group_name: %s", group_name.c_str());
103    }
104    else {
105        node_handle.setParam("group_name", group_name);
106        ROS_INFO("No group_name found. Default used: %s", group_name.c_str());
107    }
108    // Get or set max_vel_scale_factor
109    if (node_handle.getParam("max_vel_scale_factor")) {
110        node_handle.getParam("max_vel_scale_factor", max_vel_scale_factor);
111        ROS_INFO("Got max_vel_scale_factor: %f", max_vel_scale_factor);
112    }
113    else {
114        node_handle.setParam("max_vel_scale_factor", max_vel_scale_factor);
115        ROS_INFO("No max_vel_scale_factor found. Default used: %f",
            max_vel_scale_factor);
116    }
117    // Get or set planning_time
118    if (node_handle.getParam("planning_time")) {
119        node_handle.getParam("planning_time", planning_time);
120        ROS_INFO("Got planning_time: %d", planning_time);
121    }
122    else {

```

```
123     node_handle.setParam("planning_time", planning_time);
124     ROS_INFO("No planning_time found. Default used: %d", planning_time);
125 }
126 // Get or set planning_time
127 if (node_handle.hasParam("num_planning_attempts")) {
128     node_handle.getParam("num_planning_attempts", num_planning_attempts);
129     ROS_INFO("Got num_planning_attempts: %d", num_planning_attempts);
130 }
131 else {
132     node_handle.setParam("num_planning_attempts", num_planning_attempts);
133     ROS_INFO("No num_planning_attempts found. Default used: %d",
134             num_planning_attempts);
135 }
136 // Set options regardless of server parameter
137 node_handle.setParam("options", options);
138
139 // Initializing robot
140 robot = new ih::RobotPlanningExecution(
141     group_name,
142     max_vel_scale_factor,
143     planning_time,
144     num_planning_attempts,
145     ih::RobotOptionFlagFromInt(options));
146
147 // Advertise the services
148 ros::ServiceServer goto_service = node_handle.advertiseService("go_to_pose",
149     , goToPoseService);
150 ros::ServiceServer plan_service = node_handle.advertiseService("plan_pose",
151     , planPoseService);
152
153 ROS_INFO("robot_movement_service ready to use for: %s", group_name.c_str());
154 ;
155 ros::spin();
156
157 return 0;
158 }
```

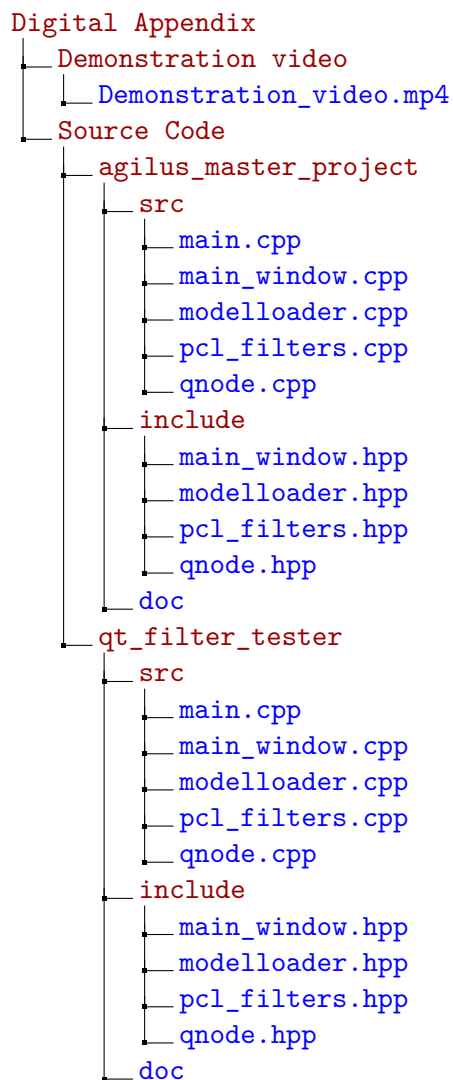
Pose.srv

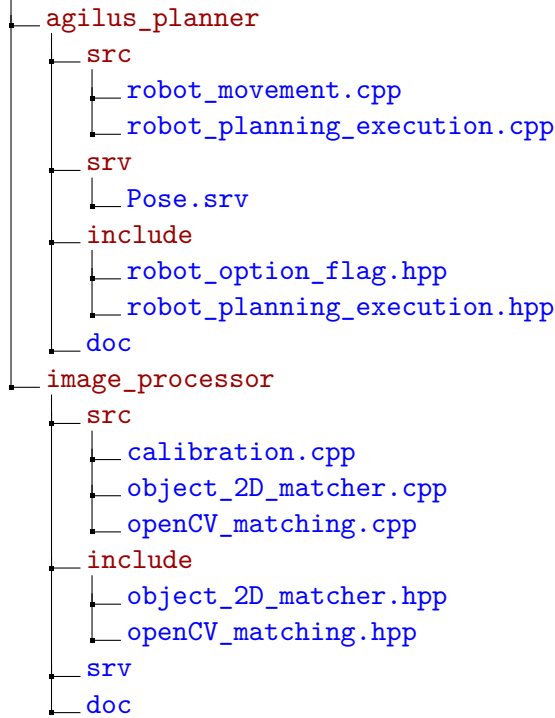
Listing C.2: Source file - code/agilus_planner/Pose.srv

```
1 | Header header
2 | bool relative
3 | bool set_position
4 | float64 position_x
5 | float64 position_y
6 | float64 position_z
7 | bool set_orientation
8 | float64 orientation_r
9 | float64 orientation_p
10 | float64 orientation_y
11 | ---
12 | float64 progress
```


Appendix D: Digital Appendix

This section describes the purpose and contents of all the files available through the digital appendix for this thesis. Located below is a directory tree illustrating the location of each individual file available. Listings in the directory tree with the color red are folders used for sorting. The different applications are placed in their own folders, containing the corresponding source code and code documentation (the code documentation is located in the **doc** folder). The documentation is generated using **doxygen**. The generated documentation can be viewed by opening the *annotated.html* file located in **/doc/html/*. Launching this file will open a new page in the default web browser containing the interactive documentation page.





The following is a brief description of the files available through the digital appendix:

Demonstration_video.mp4 - A video demonstrating the capabilities of the system produced throughout this thesis.

agilus_master_project - This folder contains all source code and auto-generated documentation for the *agilus_master_project* application produced for this thesis.

qt_filter_tester - This folder contains all source code and auto-generated documentation for the *qt_filter_tester* application produced for this thesis.

agilus_planner - This folder contains all source code and auto-generated documentation for the *agilus_planner* application produced for this thesis.

image_processor - This folder contains all source code and auto-generated documentation for the *image_processor* application produced for this thesis.