NTNU
Norwegian University of Science and Technology

Real-time point cloud registration from RGB-D camera mounted on a robot arm using GPU acceleration

Ulrich Johan Isachsen

Master of Science in Computer Science
Submission date: June 2018
Supervisor: Theoharis Theoharis, IDI
Co-supervisor: Ekrem Misimi, SINTEF Ocean

Norwegian University of Science and Technology
Department of Computer Science

Preface

This document is a master thesis conducted at NTNU at the department of Computer Science as part of the Computer Science study program. The master thesis is conducted in cooperation with Sintef Ocean at their Robot lab. This work is part of the Work Package 3 of the iProcess project lead by Sintef Ocean (http://iprocessproject.com/). The work was done during the spring semester of 2018.

Trondheim, 2018-06-12

Ulrich Johan Isachsen

Acknowledgment

I would like to thank the following persons for their great help during the course of the project.

Ekrem Misimi For project suggestion and guidance of project.

Theoharis Theoharis For project guidance.

This project is carried out in cooperation with Sintef Ocean and I would like to thank them for providing equipment, workplace and supervision by Ekrem Misimi.

O.N.

UJI

Summary and Conclusions

Recent development of 3D scanners have provided small, precise and cheap consumer grade scanners operating in real-time. Especially because of their low weight, they are being considered for usage in visual servoing (vision aided robot control). These 3D scanners take images that are represented as an unstructured list of points represented by their position in 3D space and color called point clouds. The goal of this project has been to perform real-time point cloud registration using Graphics Processing Unit (GPU) acceleration (Registration is the task of aligning separate 3D scans into a single point cloud that minimize the distance between common features in the point clouds) for a camera mounted in an eye-in-hand configuration to perform scans of food and food-like objects. This is different from a lot of previous work that look at room scale scans.

This thesis is based on state of the art algorithms for GPU accelerated point to point based registration and extended this with a point to plane linear least squares optimizer, which is one of the novel contributions of this thesis. This also involved an implementation of a kD-tree search algorithm on the GPU. The point-to-plane optimizer is compared to a recent grid based registration algorithm that is also extended with a pre-search strategy and a multi-resolution search. In this project, a new point correspondence rejector based on a boundary rejection strategy that improves registration quality was evaluated, showing that it could greatly improve registration quality. Apart from the specific implementations, one of the main contributions of this master thesis is the use of a Relative Pose Error (RPE) based error metric to evaluate algorithms that are conceptually different.

Both performance and registration correctness has been evaluated for the GPU implementations of the 3D registration methods in this thesis, showing that real-time point to point based registration is possible for high resolution point clouds without noticeable loss of registration correctness. The GPU implementations performed are 33 times faster than similar CPU implementations and 8 times faster than the PCL-based CPU implementations. The point to plane linear least squares optimizer was proven to perform well both performance wise and correctness wise. The grid based search with its suggested improvements was observed to be fast, but didn't match the quality of the point to point based algorithms on the GPU. The algorithm correctness was evaluated using a reference trajectory generated with the robot arm. The point to point distance evaluation strategy that is typically used when evaluating registration algorithms is in this work shown to be insufficient to compare different algorithms to each other, in which the reference path strategy is an external evaluation that is shown to be an efficient metric when comparing different algorithms to each other. The 3D registration methods and implementations used in this thesis are evaluated against a number of rigid and non-rigid properties varying textural and optical properties (the desired application for the project is to be used with compliant objects and food objects).

Contents

Preface i
Acknowledgment ii
Summary and Conclusions iii

1 Introduction **3**
1.1 Background and motivation 3
1.2 Problem Formulation 4
1.2.1 Real-time point cloud registration 4
1.2.2 Problem formulation 5
1.3 Problem solution 6
1.4 Structure of the Report 8

2 Background **9**
2.1 RGB-D cameras 9
2.1.1 RGB-D sensors 9
2.1.2 Alternatives to the Intel RealSense camera 11
2.2 Point Cloud Library (PCL) 12
2.3 Point Cloud representation 13
2.3.1 2D-grid representation (organized point cloud) 13
2.3.2 Unstructured cloud 13
2.3.3 Voxel representation 13
2.4 Downsampling 14
2.5 Normal calculations 15
2.5.1 Problem formulation 16
2.5.2 Solving the normal vector 16

2.5.3 Getting the correct orientation 17
2.6 Segmentation 18
2.6.1 Geometric model fitting 18
2.6.2 Clustering 20
2.7 RGB-D images 21
2.8 GPU programming 23
2.8.1 Comparing a GPU to a CPU 23
2.8.2 GPU restrictions 24
2.8.3 GPU memory 25
2.8.4 Programming concepts 25
2.9 kD-Trees 26
2.9.1 Contructing a kD-tree 26
2.9.2 kD-tree search 28
2.9.3 kD-tree Optimizations 28
2.10 Point cloud registration 32
2.10.1 The ICP algorithm 32
2.10.2 Non linear ICP (accelerated ICP) 36
2.10.3 Using point to plane as transformation minimization 36

3 Related Work **41**
3.1 Variations of the ICP algorithm 41
3.1.1 Selection of points 42
3.1.2 Matching points 42
3.1.3 Weighting of pairs 44
3.1.4 Point pairs rejection 44
3.1.5 Error metric 45
3.1.6 Minimization 45
3.2 Voxelized space for approximate nearest neighbours 46
3.2.1 Description of algorithm 46
3.3 kD-tree acceleration on GPU 47
3.3.1 GPU accelerated kD-tree search for registration 47
3.3.2 Other GPU based kD-tree implementations 49
3.4 Kinect fusion 49
3.4.1 Truncated Signed Distance Function (TSDF) 50
3.4.2 Measurement 50
3.4.3 Pose estimation 50
3.4.4 Update reconstruction 51
3.4.5 Surface prediction 51

3.4.6 Summary 51
3.5 Kinfu 52
3.6 A PCL based 3D scanner 52
3.6.1 Voxel grid filter 53
3.6.2 Crop box filter 53
3.6.3 Plane model segmentation 53
3.6.4 Euclidean cluster 53
3.6.5 Initial alignment 53
3.7 Other alternatives to ICP 54

4 Implementations 57
4.1 Robot control for camera movement 57
4.2 Camera capture methodology 61
4.3 Registration framework 63
4.4 Registration algorithms 67
4.4.1 Brute-force correspondence estimation CPU 67
4.4.2 Brute-force correspondence estimation GPU 68
4.4.3 Kd-tree correspondence estimation CPU 68
4.4.4 kD-tree correspondence estimation on GPU 69
4.4.5 Voxelized search space for correspondence estimation on CPU 69
4.4.6 Voxelized search space for correspondence estimation on GPU 70
4.5 Point correspondence rejector 70
4.5.1 Rejection box construction 70
4.5.2 Finding the transform 71
4.5.3 Doing the correspondence rejection 72
4.6 Other implementation considerations on GPU. 73
4.6.1 CPU and GPU interaction 73
4.6.2 Point cloud transformation 74
4.6.3 Reduction trees 74
4.6.4 Convergence/termination criteria 75
4.6.5 Evaluation tool 75

5 Evaluation 77
5.1 Experimental setup 77
5.1.1 Robot and camera setup 77
5.1.2 Test sets 77
5.1.3 Evaluation metrics 82
5.1.4 Experimental setup 84

6 Results **91**
6.1 Registration results 91
6.2 Camera differences 91
6.3 Camera and robot paths 95
6.3.1 Robot trajectories 96
6.3.2 Estimated free-hand trajectories 96
6.4 ICP variations 96
6.4.1 Trajectory estimation 99
6.4.2 Varying number of iterations 99
6.4.3 Varying fitness epsilon 102
6.4.4 Varying the transformation epsilon 103
6.4.5 Varying number of images 106
6.5 Scan method results 107
6.6 kD-tree search stack size 109
6.6.1 Performance 109
6.7 Voxel search variations 111
6.8 Correspondence rejection 116
6.9 Registration of translation scans 118
6.10 Real time registration video 118
6.11 Other results 119

7 Discussion **121**
7.1 Registration quality and performance 121
7.1.1 Registration quality 121
7.1.2 Registration performance 122
7.1.3 Free-hand scans 123
7.1.4 Wall time 123
7.2 Voxelized ICP variations 123
7.2.1 Voxelized ICP algorithm 123
7.2.2 Voxel size 124
7.2.3 Multilevel search 124
7.2.4 Pre-search method 125
7.2.5 Race condition in Voxelized ICP 125
7.2.6 Comparing the Voxelized CPU and GPU implementations 125
7.3 Dataset quality 126
7.3.1 Using a custom dataset 126
7.3.2 The robot based datsets 126
7.3.3 Free-hand datasets 127

7.4 Robot arm movement . . . 127
7.5 Evaluation metrics . . . 128
7.6 Camera differences . . . 128
7.7 Point correspondence rejector . . . 128
7.8 ICP variations . . . 129
7.8.1 Stack size . . . 129
7.9 Scan methods and object variations . . . 129

8 Conclusion and further work 131
8.1 Conclusion . . . 131
8.2 Further work . . . 134

A Acronyms 137

B Quaternion operators 139
B.1 Quaternion/matrix rotation magnitude . . . 139
B.2 Converting from a quaternion to a matrix . . . 140
B.3 Quaternion circular interpolation (slerp) . . . 140
B.4 Quaternion to euler conversion . . . 140

C Registration framework arguments 141
C.1 Program usage . . . 141
C.2 Arguments . . . 142

D Registration suggestions 147
D.1 Scanning . . . 147
D.2 Filtering . . . 147
D.2.1 Box filter . . . 147
D.2.2 Voxel grid filter . . . 148
D.2.3 Plane model segmentation filter . . . 148
D.2.4 Euclidean cluster filter . . . 148
D.3 Transformation and correspondence estimation . . . 148
D.4 Convergence criteria . . . 148
D.5 Point correspondence rejectors . . . 149
D.6 Additional advice . . . 149
D.6.1 Trial and error . . . 149
D.6.2 Initial alignment . . . 149
D.6.3 Using GPU for registration . . . 150

Bibliography 151

CHAPTER 1

Introduction

This section will describe the background for the project as well as the problem formulation this master thesis is based on. This master thesis is a continuation of the autumn project conducted at NTNU fall 2017 (Isachsen (2017); IDI (2018)). The pre project was about evaluation of some ICP based registration algorithms, while this master thesis is based on those results to perform GPU implementation of a few registration algorithms to obtain real-time registration. Since this MSc thesis is a natucal continuation of the pre-project, certain parts of the text is taken from the pre-project (Isachsen (2017)) (section 1.4 contains a list of the sections that is taken from the pre-project).

1.1 Background and motivation

The IProcess project (www.iprocessproject.com) is a research project headed by Sintef-Ocean (2018). The aim of the IProcess project is to develop novel concepts for flexible robot based automation in the food processing industry. One part of this project is the WP3 research (Sintef Ocean (2018)). The goal of WP3 is to develop vision guided and machine learning based robot manipulation of compliant food objects. This involves developing a grasping concept that can find suitable grasping poses for a robot arm gripper. In order to do this, the robot arm is given point clouds obtained from an RGB-D camera mounted on the arm itself (eye-in-hand configuration).

As the robot arm moves, the RGB-D camera will obtain scans of the object from different robot arm poses. This thesis will explore the registration step, which is about merging the different scans obtained into one model which represents the model and/or the environment. The goal of the project is to evaluate different variations of registration algorithms implemented on the graphics processing unit and evaluate the possibility of doing registration in real-time.

The WP3 research have multiple subtasks which rely on obtaining good 3D-registrations. Having a good point cloud registration will first of all enable 3D model construction of arbitrary objects. The meshes obtained from 3D model construction can then be used further to for example perform tracking of the objects while they are deformed or changed (as described in section 1.2.1). However, for 3D model construction, real-time registration is only convenient but not required. Other related tasks are related to object reconstruction such as next-best-view estimation. This task is about finding the next view a camera should have of an object to minimize occlusion. In this task, it is very beneficial to have fast feedback from a registration algorithm, such that it can quickly find a trajectory that efficiently scans the entire object. In this case, it is especially to have a real-time registration to help the robot make decisions on the fly. Other task might include robot grasping. In this case, the robot is presented with an object and should determine how the object should be grasped and handled. In this case, a fast registration algorithm is beneficial to give the robot a more complete 3D view of the object and quickly make a decision regarding grasping and handling.

1.2 Problem Formulation

This section will first discuss why real-time point cloud registration is a problem and why this is important before it gives a clear problem formulation.

1.2.1 Real-time point cloud registration

Food objects issue some challenges in robotics for some of the following reasons:

- They might be soft, and change shape when interacted with.
- Items differ in color, shape and size.
- The robot might perform actions which can significantly change the properties of the object (for example cutting).

As well as facing these challenges, the robot should also be able to process information and perform it's tasks efficiently.

As food objects have individual variations in shape, color and size, a general registration algorithm should be considered. This is because the prior knowledge of the individual objects are more limited compared to other industrial applications which have more similar objects. As the robot is supposed to operate efficiently, it is important that it scans the object in an efficient manner. This is where the importance of real-time registration comes in. If the registration is real-time, the processing time itself is reduced as well as it can enable the robot to quickly evaluate next best view, which is the next position the robot should move to in order to obtain an optimal view of the entire object. One way to obtain faster registration time is to perform point cloud simplifications such as voxelization and downsampling. However, some of the actions the robot perform might require high accuracy and tracking of certain features as it performs the action (for example cutting). For these kind of tasks, a high resolution description of the point cloud might be required. This will make point cloud simplifications a less attractive option as the maximum resolution might beneficial.

With both real-time requirements and full-resolution point-clouds, powerful hardware is required. In order to have less compromises on the resolution and registration time, Graphics Processing Units (GPUs) are used in the implementation of the registration. The Point Cloud Library (PCL) will be used for the project (Point Cloud Library (2017); Rusu and Cousins (2011)). While there exists one GPU-based implementation of registration in PCL (Kinfu (Pirovano et al. (2013))), it is highly dependent on voxelization. The processing pipeline is also quite interwoven compared to the otherwise modular design of PCL. This motivates that some other GPU based approaches of registration could be implemented into PCL (a voxelization strategy is evaluated in this report to give a comparison betweeen the two approaches).

1.2.2 Problem formulation

The main goal of the master thesis is to evaluate state-of-the-art registration algorithms and implement the most optimal and relevant ones on the GPU. The optimality and relevance is based on the interoperability with existing libraries (PCL), performance, registration quality, stability and generality (having few requirements about the point dataset). The implementations support the Point Cloud Library in such a way that it can be used in a similar way as other subsystems in the Point Cloud Library.

An Intel RealSense SR300 camera and an Intel RealSense D435 camera is provided as 3D scanners for the project (Intel (2018b,c)). These cameras are relatively cheap considering relatively good image quality (both RGB and depth), small size and low weight. This makes it a good choice for eye-in-hand configuration.

Franka is the robot arm that is provided with the project (Franka (2017)). The Franka robot is a relatively cheap robot-arm that can be controlled through a C++ library. One of the main advantages of the robot is the 7 joints that all have force-torque control. While the thesis includes implementations for controlling the robot, this is not the main focus of the project.

Given the goal above, the project was set with the following objectives

1. Implement robot control software that is able to follow certain trajectories while capturing depth and RGB images with the Intel RealSense SR300 or D435 camera.
2. Create benchmark tests. These should be based on relevant scans of objects and realistic scan trajectories of the robot.
3. Implementations of registration algorithms
   (a) Implement a brute-force point correspondence estimation version of the Iterative closest point (ICP) algorithm on CPU.
   (b) Implement a brute-force point correspondence estimation version of the ICP algorithm on GPU.
   (c) Implement a kD-tree based point correspondence estimation version of the ICP algorithm on CPU.
   (d) Implement a kD-tree based point correspondence estimation version of the ICP algorithm on CPU.
   (e) Implement a kD-tree based point correspondence estimation version of the ICP algorithm on GPU.
   (f) Implement a Voxelization based point correspondence estimation version of the ICP algorithm on CPU. This is mainly for comparison with the GPU implementation.
   (g) Implement a Voxelization based point correspondence estimation version of the ICP algorithm on GPU.
4. Evaluation, the evaluation will be a comparison of the implemented algorithms as well as some implementations already implemented in PCL.

1.3 Problem solution

Based on the problem formulation described in section 1.2.2, the work done in this thesis can be divided into two major tasks. The first task is about obtaining relevant 3D data to perform registration on. The second task is to do implementations of the algorithms described in section

1.2.2 (Brute force search, kD-tree search and voxelized search strategies for both CPU and GPU). The first task was solved by implementing a robot control program that moves the robot to user specified waypoints. This is further described in section 4.1. An Intel RealSense SR300 camera is mounted on the hand of the robot arm, and while the robot is moving between the waypoints, the camera captures depth frames to a custom file format as well as RGB-images. This is further described in section 4.2.

As for the second major task, the algorithms described in section 1.2.2 were implemented. This is done through command-line based framework that reads the data generated from the camera mounted on the robot arm, and performs registration based on multiple parameters that can be specified. Through this framework, it is possible to select a variety of ICP-based registration algorithms, including the algorithms mentioned in section 1.2.2. This is a powerful tool that enables parameter testing and evaluation of the registration algorithms.

The kD-tree based implementation is able to perform registrations in real-time under realistic circumstances. This algorithm can shortly be summarized as a standard kD-tree search algorithm with a fixed stack size. This enables the GPU perform recursion like behaviour, with some limitations. If this fixed stack size is smaller than the tree-depth, the nearest neighbour search performed on the kD-tree will be an approximation. This approximation have never been evaluated for the task of registration. This implementation is explained further in section 4.4.4.

The suggested voxelized registration method is an implementation of a O(N) search algorithm based the work done by Marden and Guivant (2012) (where N is the size of the point cloud that is to be aligned). This project contains a version of this algorithm implemented on the GPU. To avoid relying on the requirement of a close initial alignment in the algorithm presented by Marden and Guivant, additional suggested improvements are implemented. The first extension is to change the voxel size between the individual iterations of the iterative closest point algorithm, and the other suggestion is to perform a pre-search in the voxel grid to enable the closest point operator to look further than the algorithm presented by Marden an Guivant. The suggested algorithm is further explained in section 4.4.6.

The Franka robot provides precise pose estimations of the robot arm position. This enables comparison of the estimated poses obtained from the registration with the reference trajectory of the robot. This is the most important measure used in this project for evaluation of the correctness of the registrations. The other important measure is the performance of the algorithms, which is measured as the time it takes to align two point clouds, where the sizes and other properties of the point clouds are varied. This is further explained in section 5.1.3.

1.4 Structure of the Report

While this section has presented the problem setting for the task, section 2 will focus on the theoretical background for point cloud registration. Section 3 will look at similar work that looks at similar problems from different angles, which serves as an inspiration for the work in this MSc thesis. Further, section 4 will describe the implementations done for this project, and section 5.1 will focus on the experimental setup done to evaluate and compare the implemented algorithms and library implementations in PCL. Section 6 shows the results from these experiments and section 7 will give an evaluation and discussion of the presented results. Finally section 8.1 will summarize the results and conclude the findings from the experiments. Section 8.2 will be about suggested work that can extend on the work that is done here.

As this project is an extension of the pre-project, some of the text have been taken from the project (Isachsen (2017)). In the background section, sections 2.1 (the part about the SR300 camera), 2.1.1, 2.1.2, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7 and 2.10 (except the part about the point to plane linear least squares optimizer chapter) are taken from the pre-project. In the related work section, sections 3.1. In the Implementations section, section 4.3 is extended from the pre-project. Section 1.1 is also somewhat extended from the pre-project. Note that a few modifications have been done and that many of the figures are created for this document and doesn't exist in the pre-project report.

CHAPTER 2

Background

This chapter will go through the theory that is the foundation for doing point cloud registration. The first sections are about 3D sensors. The following sections are about foundations in point cloud processing in general. The section ends with a summary of the ICP algorithm.

2.1 RGB-D cameras

The issue of getting 3D data is an important one regarding robotics. Example of 3D sensors are stereo cameras, LIDAR (Light Detection and Ranging), 3D cameras, etc. 3D cameras (RGB-D) are a great compromise between a LIDAR and a stereo camera. Stereo cameras have the problem of determining the distance to featureless surfaces, while the LIDAR is very large and expensive. An RGB-D camera is relatively small and cheap, while giving relative good 3D data. This project is based around using the Intel RealSense SR300 camera and the Intel RealSense D435 to some extent.

2.1.1 RGB-D sensors

There are many existing RGB-D sensors. One such example is the Intel RealSense camera seen in figure 2.1 (Intel (2018b)). The Microsoft Kinect and ASUS Xiton Pro are examples of similar cameras (Microsoft developer (2018); ASUS (2018)). Another more expensive alternative is the Zivid camera (Zivid labs (2018)). These cameras work in a similar way using a technique called structured light 3D scanning (Intel® RealSense™ (2016); Pirovano et al. (2013); Zivid labs (2016)).

(a) Intel RealSense SR300

(b) Intel RealSense D435

Figure 2.1: Figure showing an image of the Intel RealSense SR300 camera (a) taken from the web (Intel (2017)). (b) is showing an image of the Intel RealSense D435 camera taken from the web (Intel (2018c)).

Figure 2.2 shows the internals of a structured light 3D scanner, here represented by a RealSense SR300. The most important components in regards of this chapter is the IR laser projector, the IR camera and the Color camera. The IR laser projector projects a laser pattern into the world in the infrared spectrum. This pattern is known to the system, and is captured by the IR camera. Because this pattern is known, it's possible to quantify the distortions in space and calculate the depth of each pixel from that. The color camera is applied on top of the depth image in order to get an RGB-D image consisting of pixels with a known color and the distance to the camera (Intel® RealSense™ (2016)).

The Intel RealSense SR300 camera uses the projector to illuminate the scene using multiple vertical bar patterns with increasing spatial frequency to both get near and far objects. An example of one such pattern is shown in figure 2.3. One problem with structured light scanners is that they are dependent on the reflective properties of the objects they are scanning. If the object absorbs the emitted light, it makes it impossible for the camera to deduce the depth data. Another problem occurs if the objects are reflective. If they have a lot of specular reflection, the structured light pattern is not visible at that spot, and it can also reflect to other surfaces and distort the depth data at other positions in the image. This section will not go any further into how structured light scanning is performed because it's not the concern of the project. More details on structured light scanning can be found in Skotheim and Couweleers (2004).

The Intel RealSense D435 is slightly different from the SR300 camera. It uses a fixed dotted pattern that is projected into the scene Intel® RealSense™ (2018). It uses a stereo camera pair to match features which in turn is used to deduce the 3D information. This can make this camera better at reflective and absorbing materials, as it isn't really dependent on having the projected

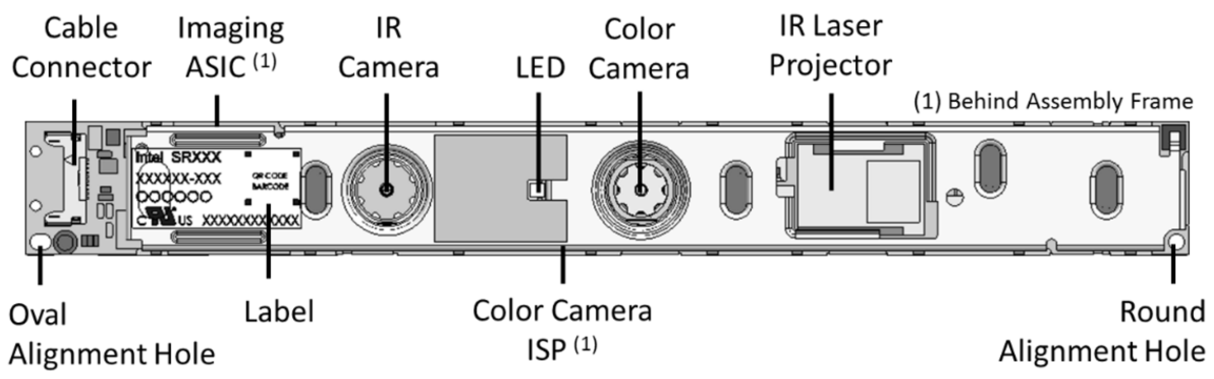


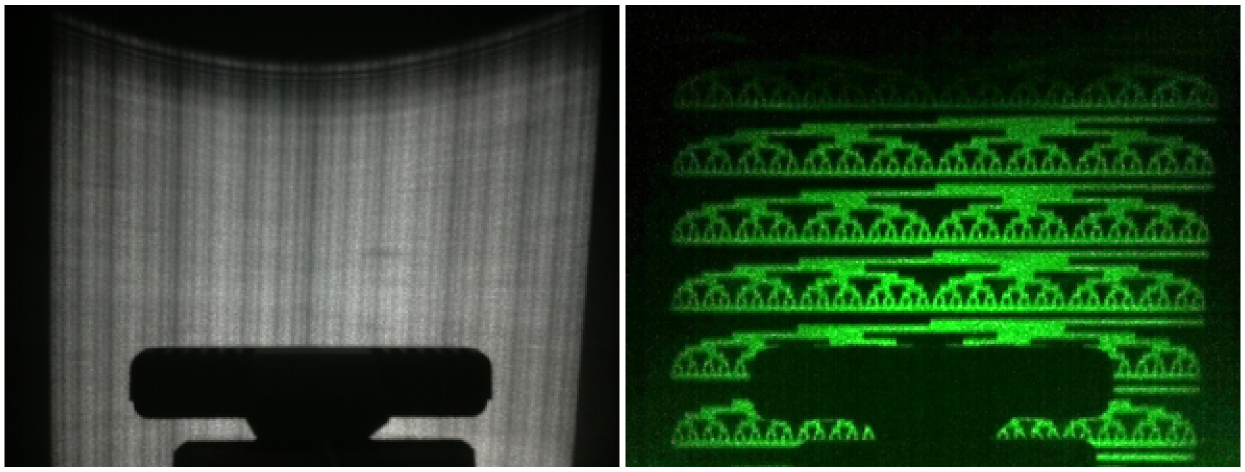
Figure 2.2: Figure showing the main components of the Intel RealSense SR300 camera. The IR Laser projector projects a pattern into the world which is captured by the IR camera. The depth image is found by inspecting the distortion of the projected pattern. This is then joined with the image from the RGB camera. Figure taken from the Intel RealSense SR300 manual (Intel® RealSense™ (2016))

pattern. The camera is also expected to work better at greater distances than the SR300 camera. The main issue with this technology is that it might lose some accuracy compared to the SR300 camera.

2.1.2 Alternatives to the Intel RealSense camera

As mentioned in section 2.1.1, there are many alternatives to the RealSense camera. Comparing the RealSense to the Kinect, they are both based on the same technology and are in the same price range. However, the Kinect has a larger blind area in front of the camera compared to the RealSense Intel (2018b); Microsoft dev Center (2017). It's also larger and heavier, making it less practical for an eye-in-hand setup. Another thing to note is that Microsoft is advising to use the RealSense instead of the Kinect, as they are working with the RealSense team for future applications Microsoft developer (2018). The Xiton Pro has more or less the same dimensions and price as the RealSense, but seems to be less popular and unavailable at the moment. It also has a large blind-area as the Kinect.

The Zivid seems to be a great alternative with great resolution and high quality. However, it's expensive, heavy and relatively large in size making it unpractical for usage in a eye-in-hand configuration (Zivid labs (2018)). Two similar examples is the Artec scanners (Artec3D (2018)) and the Creaform scanners (Creaform (2018)), which is also based on structured light scanning. They claim that they have resolution of the same scale as the Zivid camera, but has some of the same drawbacks like price, weight and scale. With all things considered, The Intel RealSense seems to be the best choice for this application (at least if price is a concern).



(a) Long exposure time. (b) Short exposure time.

Figure 2.3: Figure showing a pattern emitted by the IR projector of the Intel RealSense SR300 camera. (a) shows a longer exposure time than (b). (b) shows one of the many patterns that are emitted by the IR projector. Images are taken from a forum post on the Intel forum samotab (2015). Note that the Intel RealSense D435 camera works differently from the SR300 camera, and therefore use a different pattern Intel® RealSense™ (2018).

2.2 Point Cloud Library (PCL)

The Point Cloud Library is a large scale open source project for 2D/3D image and point cloud processing (Point Cloud Library (2017)). PCL is a popular library including libraries supporting the whole point cloud processing pipeline from input, filtering, registration, output, etc. It is designed around the principle of modularity, where the different processing stages are loosely coupled. PCL is written in C++ which is well suited for real-time registration and GPU development regarding performance. While development have been slow the last years there is still active contributions, and has an active forum (Open Perception Foundation (2017b, 2018b, 2017c)). Given the popularity and number of active contributors, PCL seems to be high quality software that is well maintained. Another advantage of PCL is that it has a relatively good documentation with examples of common operations done with PCL (Open Perception Foundation (2017b)). One problem with PCL is that it lacks GPU implementations of some of the modules. The Kinfu project was a promising extension, but was not entirely integrated into the standard PCL registration framework, and had a quite specific use case of large scale scene reconstruction (Pirovano et al. (2013)). This motivates the work done in this thesis as it includes a modular point to point based solution for registration implemented on the GPU. To summarize, PCL is a library that seems promising regarding point cloud processing.

2.3 Point Cloud representation

In terms of the Point Cloud Library (PCL), there are two ways to store point clouds (Open Perception Foundation (2017a)). One is referred to as a organized cloud while the other is referred to as an unstructured cloud. The structured cloud is dividing the space into some structure, where the points are placed inside this structure based on one or more rules. An unstructured point cloud can be considered an array of single points with no overall structure.

2.3.1 2D-grid representation (organized point cloud)

Another way to represent point clouds is to structure them in a 2D grid as they are captured by the input device (Open Perception Foundation (2017a)). As described in section 2.1.1, the point clouds are captured as an RGB image with corresponding depth values. The points are ordered in a 2D grid, where each pixel represents the 3D position of that point after deprojection. In this way, the point cloud is represented in a compact (dense) way given that it is obtained from a camera, while still having some structure which can be exploited by algorithms. While avoiding loss of accuracy of the point clouds (compared to voxelized point clouds), the disadvantage of this representation is that it's less flexible when adding and removing points. When transforming the cloud, it might lose the validity based on the transform (for non-rigid transforms for example), but it's still simple to convert to an unstructured cloud if these operations are required. In PCL, point clouds can be represented as either a 2D array like described here, or as an unstructured cloud.

2.3.2 Unstructured cloud

Finally, the unstructured point cloud is simply a list of points without any specified structure (Open Perception Foundation (2017a)). This is the most general and compact representation of a point cloud, but because of lacking structure, it usually involves more computations on multiple algorithms that depend on the relative positions of the individual points.

2.3.3 Voxel representation

One way to structure the point cloud is called voxelization. This involves dividing the 3D space into cells of a fixed size ordered in a grid structure, which can be stored into a 3D array. One such cell is called a voxel. A voxel is either empty or is filled with a value. For colored point clouds, a voxel can consists of values representing the color (for example RGB values). Voxelized point clouds can be considered as an image in 3D space instead of 2D. Voxelization is a method that is frequently used when constructing objects and scenes with examples like Kinfu and Kinect fusion (Newcombe et al. (2011); Pirovano et al. (2013)). They are especially useful when doing operations like ray-tracing, which is about finding where a line intersects the surface of some object (or similar). The disadvantage of voxel represented point clouds is that they might contain a lot of redundant information. Depending on the scene, most voxels are empty as standard 3D cameras capture surfaces and not volumes. Figure 2.4 shows a visualization of a vozelized space.

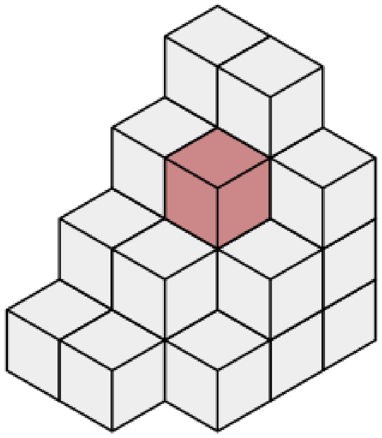
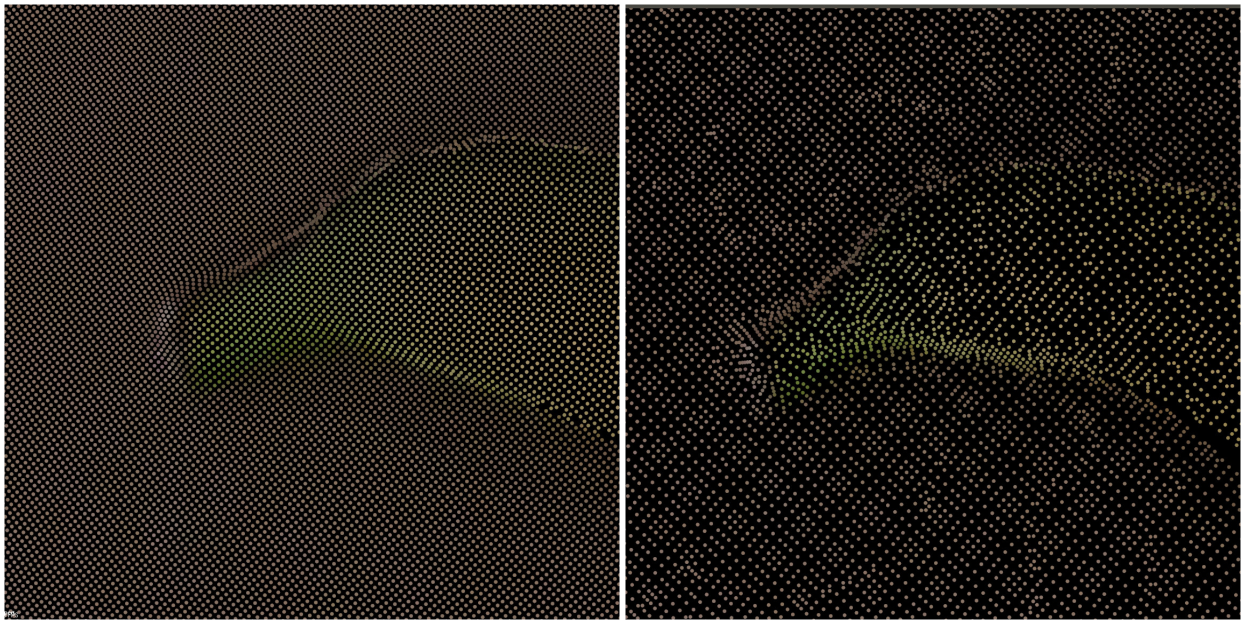


Figure 2.4: Figure showing how 3D space can be divided into voxels. Figure is taken from wikipedia (Wikipedia (2017))

2.4 Downsampling

Given the computation time required by of some of the PCL algorithms, certain algorithms would require downsampling in order to meet real-time constraints. While this project aims at avoiding downsampling, it's still a common technique to use. Downsampling can be done in multiple ways. One example of downsampling is to reduce the resolution of the input images obtained from the camera using standard image downsampling techniques. Another technique is to use a voxel grid filter (Open Perception Foundation (2017b)). A voxel grid filter starts by dividing the space into equally sized quadratic cells stacked tightly in a grid. Points are mapped to their closest voxel, and if multiple points are mapped to the same voxel, all those points are replaced by their centroid. An example of voxel grid downsampling can be seen in figure 2.5. See section 2.5.2 for a definition of a centroid. Note that this technique differs from voxel point cloud representation because it doesn't change the representation of the point clouds (they can still be organized or unorganized). The voxelization is an intermediate step to reduce the number of points.



(a) Full resolution banana (b) Downsampled banana

Figure 2.5: Figure (a) is showing the full resolution point cloud, while (b) is showing the exact same point cloud after a voxel-grid filter is applied. The resolution is clearly reduced. The figure show that (b) has fewer point than (a), but the main characteristics of the object is still maintained.

2.5 Normal calculations

A normal vector is a vector whose direction perpendicular to a surface. Since point clouds don't contain a surface in itself, the normals are calculated assuming there is a surface related to the points. Usually, this involves looking at the neighbours of the individual points and assume that they are part of the same surface. This assumption is usually valid when working with 3D scans, as the scanners only scan the surface of objects. This also introduces the notion of neighbourhoods. Rusu (2009) categorized neighbourhoods either as a radius search or a k-nearest search. In the radius search, all neighbours are considered neighbours if they are within a radius r of the point of interest. In the k-nearest neighbourhood, the k-nearest points are considered part of the neighbourhood. Rusu concludes that which metric to use and values for r and k depends on the application.

The normals in a point cloud can be viewed as a higher level feature in point cloud than the individual points as it also involves the the context of those points. Normal estimation can be done in multiple ways as there is no perfect way to determine a surface when the data only consists of independent points. A comparison of multiple normal estimators was done by Klaas Klasing (2009). This project however, will focus on the method used in PCL which is summarized by Rusu (2009). Figure 2.6 shows how the normals are computed for a 2D case. The method is similar to the 3D case.

2.5.1 Problem formulation

One simple way of calculating the normal of a point is to consider it's neighbourhood and fit a plane through this neighbourhood (J. Berkmann (1994)). The plane is described by a point **x** and a normal vector $\vec{n}$. The goal is to define a plane that minimizes the distance to the plane of all the points in the neighbourhood. The distance from a point to a plane can be described as

$$d_i = (p_i - \mathbf{x}) \cdot \vec{n}$$

where d_i is the distance between point p_i and plane described by point **x** and normal vector $\vec{n}$.

2.5.2 Solving the normal vector

The point **x** can be calculated as the centroid of the neighbourhood around the point of interest:

$$\mathbf{x} = \bar{p} = \frac{1}{k} \sum_{i=1}^{k} p_i$$

where k is the number of points in the neighbourhood of the point of interest, and p_i is point i in the same neighbourhood. Further, the normal of this plane can be calculated from the covariance matrix of the neighbourhood

$$C = \frac{1}{k} \sum_{i=0}^{k} w_i \cdot (p_i - \bar{p}) \cdot (p_i - \bar{p})^T$$

where k is the number of neighbours and w_i is the weight of point p_i, but is usually just set to 1.

The eigenvalues and eigenvectors of the covariance matrix form an orthogonal frame. The eigenvalues λ can be interpreted as the significance of the eigenvectors. This means that the eigenvector with the largest corresponding eigenvalue represents the direction with the largest variance when they are found from the covariance matrix. By taking the eigenvector of the smallest λ, we are selecting the unit direction in which the variance of the points are minimized. Given that the points are placed on a plane, this will also represent the normal of the plane that minimizes the distance of the neighbourhood points to the plane.

The eigenvectors are defined as

$$C \cdot \vec{v}_j = \lambda_j \cdot \vec{v}_j \in 0, 1, 2$$

By letting $0 \leq \lambda_0 \leq \lambda_1 \leq \lambda_2$, and $\vec{v}_0$ be the corresponding eigenvector to λ_0, $\vec{v}_0$ is the estimated normal of the plane, which is the estimated normal of the point of interest.

2.5.3 Getting the correct orientation

Solving the eigenvector problem doesn't necessary give the correct orientation of the normal vectors relative to each other or to the camera. Rusu (2009) discuss some methods for this, which will be summarized here. If the wanted viewpoint w_p is known, the following condition is what is required of the sign of the normal vector:

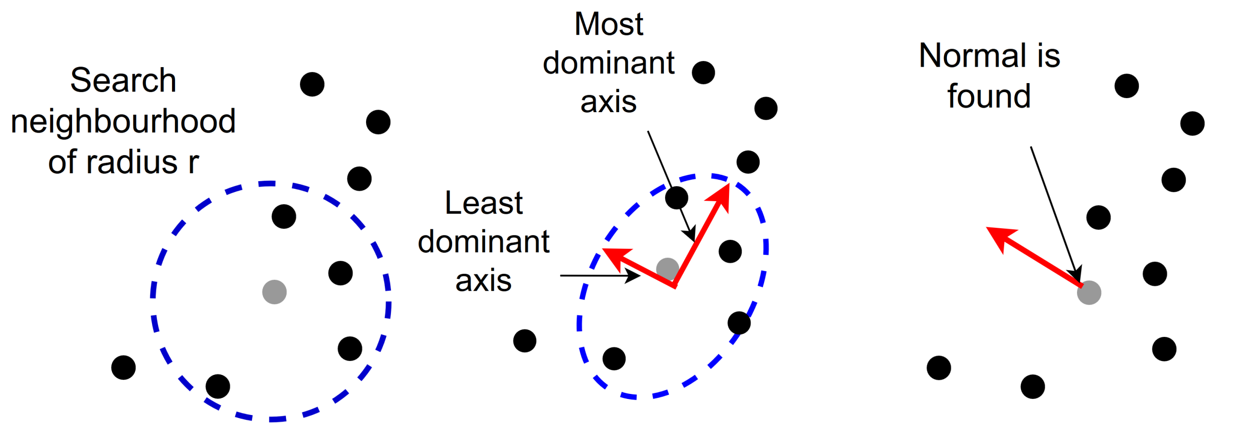
$$\vec{n}_i \cdot (v_p - p_i) > 0$$

If this condition is wrong, one can simply flip the sign of the normal vector. If the viewpoint is not known, the problem is slightly harder. Hoppe et al. (1992) suggested a solution to this which involves transforming a point cloud to a graph problem where each point is represented as a node with edges to all other nodes. Edge weights are set as the euclidian distance between the two points. The algorithm will construct a spanning tree from this graph. It will start with a point in the graph which has a known true normal orientation or is randomly chosen if no such point is known. It will traverse the graph from this point, and flip neighbours based on the following rule

$$\vec{n}_i \cdot \vec{n}_j < 0 \Rightarrow \vec{n}_j = -\vec{n}_j$$

Here, $\vec{n}_i$ is the normal of the point with a known orientation and $\vec{n}_j$ is the unknown point. After a point is evaluated, it can be considered to have a known orientation and the algorithm can continue.

While this algorithm shows a way to flip wrongly oriented normals, it's still quite computationally heavy, and often it gives just as good results just setting a viewpoint somewhere (Rusu (2009)). When using a 3D scanner, the viewpoint of the camera relative to the point cloud is known. When applying the registration step (section 2.10.1), this viewpoint can be transformed to it's position relative to the result point cloud. This is done by applying the same transformation found in the registration step used to transform the point cloud.



(a) Neighbourhood search (b) Eigenvectors (c) Normal of point

Figure 2.6: Figure showing how the normals of a points is computed in 2D. The method is similar for 3D. (a) shows the point, whose normals is to be computed marked as a gray point. (b) show a representation of the eigenvectors computed for the neighbourhood found in (a). Finally (c) show that the normal is found as the least least significant eigenvector, representing the axis of least variance.

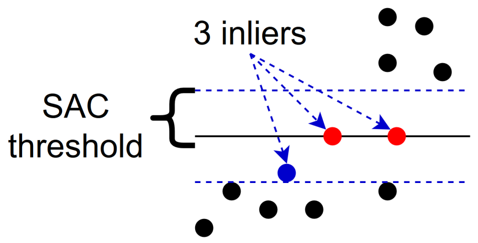
2.6 Segmentation

Segmentation is the task of separating the point cloud into different categories based on characteristics of the data. Rusu (2009) mentions two main categories of segmentation that is relevant for point cloud data. The first category is about fitting simplified geometric models, while the other is about clustering in general.

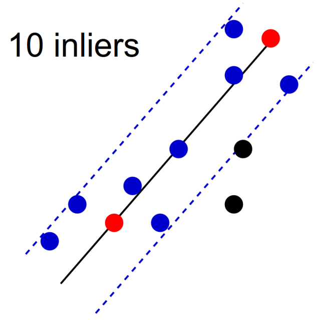
2.6.1 Geometric model fitting

Fitting the point cloud to some simple geometric model is one way of segmenting the data. In geometric model fitting, the input is a point cloud and a geometrical shape. The goal is to find a set of points as large as possible that fit the geometrical shape based on some error metric and a threshold on the error. One way of finding a set of points is the RAndom SAmple Consensus (RANSAC) algorithm introduced by Fischler and Bolles (1981) The algorithm is described below using a plane model in the next paragraph as done by Rusu (2009). The method can easily be extended to other models as well, but for this Project, only the plane model is used. Figure 2.7 shows the RANSAC algorithm applied on 2D spatial data. The concepts are the same for the 3D case.

RANSAC applied on a point cloud to find the best-fitting plane can be described with the following steps:



(a) First iteration with 3 inliers.



(b) Second iteration with 10 inliers.

Figure 2.7: Figure showing the RANSAC algorithm for plane fitting performed in a 2D case with two iterations. (a) shows the first iteration where two random points have been selected to describe a line, and 3 points are within the threshold (SAC threshold). (b) shows the second iteration where a different point pair has been selected to describe the line. In the second iteration 10 inliers are found, so this is kept as the current best line. The steps are similar for the 3D case, where the line is replaced with a plane. This also involves using 3 random points to describe the plane instead of the two required for a line.

1. Randomly select three non-collinear unique points from the point cloud. In other words, they should not lay on the same line or be the same points in any way. Given another model to fit, fewer or more points might be required.

2. Solve the model's equation based on the three points. For a plane this is $a\mathbf{x}+b\mathbf{y}+c\mathbf{z}+d=0$. This model must obviously be replaced if using another model than a plane.

3. Calculate the distance from all points in the point cloud to the model found in the previous step. For the plane model, this is just the euclidean distance from each point to the closest point on the plane.

4. Count the number of points whose distance is less than a user specified threshold.

This process is repeated k times, where the best model is kept and returned as the best fitting model. k can be chosen as an arbitrary number or based on some probability of selecting the best fitting model. The following equation can be used for just that (Fischler and Bolles (1981))

$$1-p=(1-(1-\epsilon)^s)^k \Rightarrow k=\frac{\log(1-p)}{\log(1-(1-\epsilon)^s)}$$

ϵ is the probability of choosing a sample that produces a bad estimate. s is the number of good samples that is desirable to have, p is the desired probability of success. For the plane model, $s=3$ as it's required to pick three points to solve the plane equation. ϵ is the probability that a randomly picked point belongs to the actual object (for example, if half the model consists of a plane: $\epsilon = 0.5$). It therefore depends on the scan application. However, for a scan of objects laying on a table, there is a high probability that a point belongs to the table, which means that it's not required to have a too large k to segment the table surface.

Rusu (2009) extends the model fitting method for general object as well, and can be used to segment objects of interest given that the model definition is known. However, given that the application of this project is to be used in the food processing industry, objects are unlikely to be similar, and segmenting objects based on some reference model will be impossible.

2.6.2 Clustering

The general goal of clustering is to divide a point cloud into smaller parts. This goal is very general, and can be achieved in multiple ways. While section 2.6.1 focused on segmenting data to best fit a mathematical model, this section will focus on object clustering. This means that the point cloud contains some kind of object structures, while the geometric properties are not necessarily known. The general assumption here is that the objects are spatially separated. Given that objects are not placed too close to each other, it's possible to separate the objects because they have significant gap between each other (Rusu (2009)).

The "gap" mentioned in the previous paragraph must be defined before proceeding. Though it can be defined in multiple ways, one simple but efficient way is to define it as the minimum distance between two or more object point clusters. If the minimum distance between these object point clusters are above a certain threshold, they are considered separate clusters, but if it is below this threshold they are considered the same object point cluster.

Rusu (2009) defines an object point cluster in the following way. $O_i = p_i \in P$ is a distinct point cluster from $O_j = p_j \in P$ if the following is true

$$\min ||p_i - p_j||_2 \geq d_{th}$$

d_{th} is a user specified distance threshold. This is the minimum allowable distance between the two point clusters if they should be considered as separate point clouds. P is the point cloud containing the object clusters if there are any. This must be valid for all $p_i \in O_i$ and $p_j \in O_j$ if they are to be considered as separate object point clusters.

Rusu (2009) also suggests an algorithm in order to find clusters such that this property is maintained. This algorithm is summarized here and visually shown in figure 2.8 for a 2D case.

1. Create a kd-tree representation of the point cloud P. This step is not really required, but is almost a requirement in order to get acceptable performance. kD-trees are explained in section 2.9.

2. $C \leftarrow$ An emtpy list of clusters,
   $Q \leftarrow$ A queue of points to be checked.

3. For each point $p_i \in P$, do the following steps:

   (a) Add p_i to Q and mark it as *processed*.

   (b) For every point p_i in Q do:

   - Search for neighbours P_i^k of p_i with radius $r < d_{th}$ that have not been processed.
   - Add all points in P_i^k (which is not marked as *processed*) into Q. Mark all points in P_i^k as *processed*.

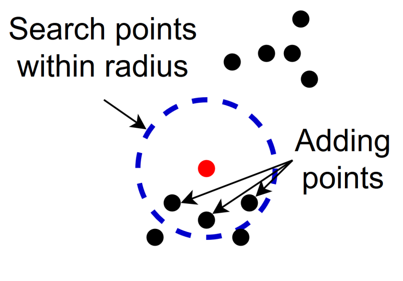
4. Termination happens when all points $p_i \in P$ have been evaluated. All points now belong to at least one of the clusters in C.

Each iteration of step 3 will add the points to an unique cluster. The algorithm described above will evaluate every point and search for neighbours within a radius. All those points are considered part of the same cluster as that point. For each of the newly found points, the same search is performed, possibly adding new points to the cluster. This is repeated until no new points can be added to the cluster without violating the distance threshold rule. Then this step is repeated until all points belong to some cluster.

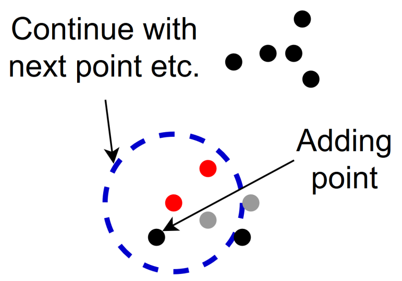
2.7 RGB-D images

An RGB-D image is a 2D array of pixels much like a standard camera. Unlike a normal camera, each pixel is represented by 4 values instead of 3. In addition to storing the color information as 3 RGB values, it also stores the depth that pixel has relative to the camera. This can be deprojected into a scene and produce a point cloud. This chapter will focus on why RGB-D images are preferred to standard RGB images and depth images without color data. Figure 2.9 shows an RGB-D image. The depth and color data is separated in the figure, but can easily be combined on the computer while processing them.

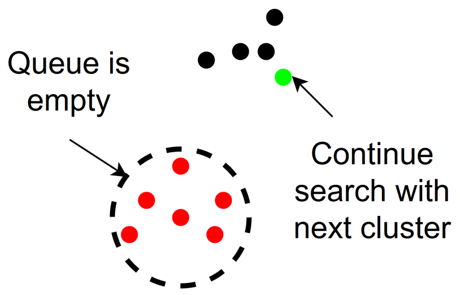
One of the advantages of RGB-D images is that it's able to combine standard 2D image techniques (which is often used in robotics) with 3D data. Using RGB data is for example very important when doing visual tracking, which is an important task in robotics. As another example, Chen et al. (2016) suggested a method combining 3D data and RGB-data to improve 3D object



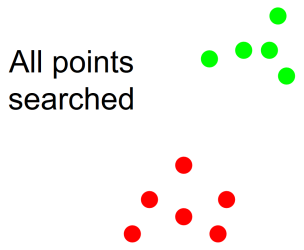
(a) Point selection



(b) Next point from queue

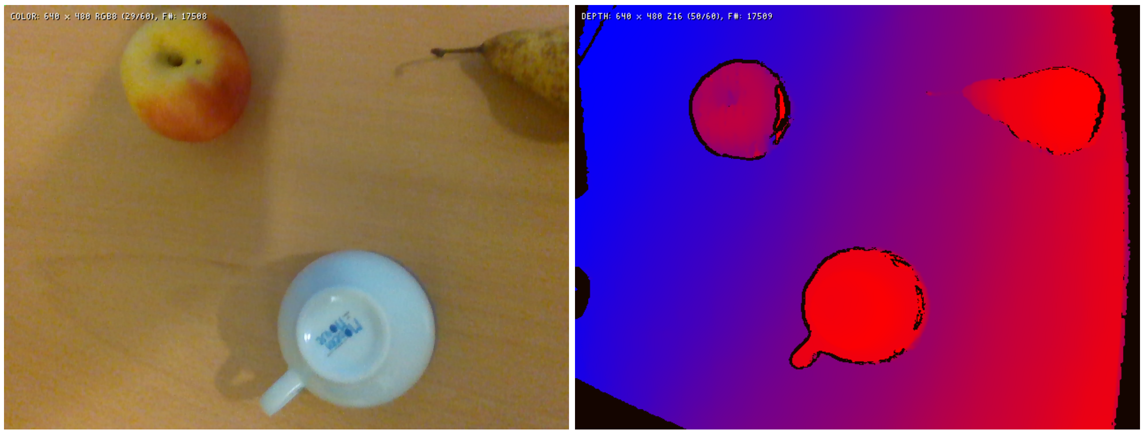


(c) Find new point and add to new cluster



(d) All points are found

Figure 2.8: Figure showing how the clustering method described by Rusu Rusu (2009) for 2D data. (a) starts with selection of some point and adding neighbours within a certain radius to the queue. (b) selects the next point from the queue and processes it the same way as for (a). In (c), the queue is empty and a new unvisited point is selected and processed as (a) and (b) but with a new cluster index. Finally, (d) shows that all points have been visited and added to a cluster, making the algorithm terminate. The method is the same for the 3D case.


COLOR: 640 × 480 RGB8 (29/60), F#: 17508
DEPTH: 640 × 480 Z16 (58/60), F#: 17509

(a) RGB image (b) Depth image

Figure 2.9: (a) shows the RGB content of a depth image. (b) shows the corresponding depth image. Red color is closer to the camera while blue color is further. Figure is created using the rs-capture tool provided with librealsense by Intel (2018a).

detection. At some of the test sets, using RGB data together with depth data gave significant improvements compared to using only depth data. It's also possible to think of applications where color data is necessary in addition to depth data when certain features are only visible in color.

2.8 GPU programming

This section will briefly introduce the necessary concepts required for GPU computing. The method outlined here is based on the CUDA programming guide by NVIDIA (2018). This guide is based on the parallel concepts rather than the actual architecture, though they are related. This section will use the terminology and concepts used in CUDA because this is the GPU programming language used for this project. Other frameworks like OpenCL might have a different terminology, but the main concepts are almost identical (Karimi et al. (2010)).

2.8.1 Comparing a GPU to a CPU

A graphics processing unit (GPU) is a massively parallel computational unit similar to a central processing unit (CPU). GPUs was originally designed to perform highly specialized tasks in graphics processing, but has in recent years become more general. The main difference is that GPUs come with a lot more processing cores than a standard CPU, but loses some generality when doing so. This makes GPUs better suited for algorithms that is computationally intensive and parallelizable, while the CPU is still better for serialized tasks due to their more advanced and general processing cores. A simplified comparison of CPU and GPU architecture is shown in figure 2.10.

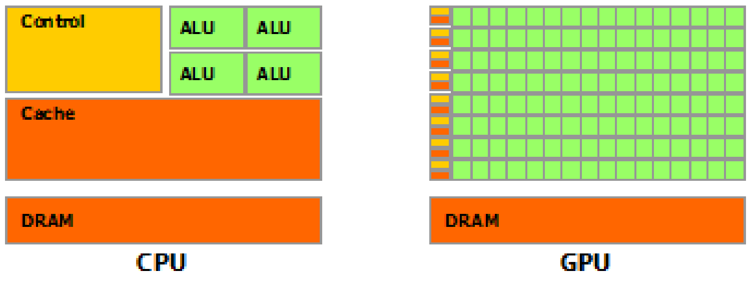


Figure 2.10: Figure showing a simplified architecture of a CPU (left) and a GPU (right) based on the die area. The figure is showing two architectures with the same die area, where the CPU typically dedicates more space for advanced control (more advanced instructions) and memory control (for more advanced memory accesses), while the GPU will dedicate more space to the arithmetical units. This enables massive parallelism for the GPU, while the CPU is better for serialized tasks and advanced branching patterns. The figure is taken from NVIDIA (2018).

2.8.2 GPU restrictions

Because of the priorities done on a GPU, there are certain limitations. The first limitations is that GPU and CPU memory is separated (this is not always the case, but is usually the case for dedicated graphics cards). This means that data must be transferred between CPU and GPU memory in order for them to share data. This introduces some overhead, meaning that the speedup obtained with the GPU might not be beneficial for data intensive applications because transfer becomes the bottleneck.

Another difference is that the individual cores are not able to execute individual instructions on the GPU. The programming model is known as single instruction multiple threads (SIMT, which is similar to single instruction multiple data (SIMD)). A group of threads that are interlocked is referred to as a warp in cuda terminology. Each warp typically consists of 32 threads, which have to operate on the same instruction at any time (they might operate on different data). The threads can have conditions that disables them (for example in a conditional branch in the code) temporarily, but must wait for the other threads to continue execution. This concept is referred to as branch divergence, and is an important consideration when programming the GPU. The programmer should think of this when programming, such that interlocked threads work on the same conditional branches when possible.

The Streaming multiprocessor (SM) is the unit that creates, manages and executes the warps. This can be viewed as a processing core that run multiple threads instead of a single one.

2.8.3 GPU memory

The memory hierarchy of the GPU is slightly different from that of the CPU. Both architectures have a main memory (referred to as the global memory on the GPU), while the GPU also has multiple specialzied caches. These caches must typically be manually managed by the GPU, while the caches of the CPU is for most practical reasons automatically configured. To obtain higher bandwidth, many of the GPU memories are banked. This also means that memory alignment is important because conflicting accesses (parallel accesses that go into the same bank) might require to be serialized. The memory is divided into the following stages according to NVIDIA (2018):

Registers These are the fastest memory available. The most special thing about them is that they are shared between all the threads, meaning that running fewer threads can enable more registers to each thread.

Local memory Memory local to an individual thread but is stored in global memory. This means that it is as slow as global memory, but is aligned in an efficient manner.

Shared memory This is a faster on-chip memory shared between the threads within a block (block is explained in section 2.8.4).

Constant memory This is a global memory cache but serializes different memory accesses instead of similar ones. This means that all accesses to the same memory address are obtained in one access. If all threads access the same element at a time, constant memory is a good choice to put this value.

Texture and surface memory A device memory cache that optimizes certain access types that are typically found when working with textures in graphics. Can be good for 2D and 3D striding in memory.

The usage of the different memory and memory caches are typically domain specific and based on a trial-and-error method to improve the memory bandwidth.

2.8.4 Programming concepts

While Streaming Multiprocessors (SMs) and warps are a hardware concepts. In software, the notion of blocks and threads are more common NVIDIA (2018). A block is a collection of threads. A thread is an individual execution unit, while a block is a collection of threads. Each block can consist of threads arranged in a 1D, 2D or 3D structure. Each block can also be arranged in a 1D, 2D or 3D structure. What this means is that each individual thread get their per-block unique identifier which is organized in an x, y and z index (Based on how many dimensions there are). The same applies for the block.

The GPU code is written in functions called kernels. These are functions that each individual thread will run in parallel. They must be designed such that the scheduler is free to select any order of execution of the threads. Inside the kernel, the x, y, and z indices for both the threads and block are available. The scheduler organizes these indices when launching the separate blocks and threads, which is automatically managed. The programmer has to specify how many blocks and threads are wanted in each dimension such that the scheduler is able to assign indices to each thread and block.

2.9 kD-Trees

Many registration algorithms and related algorithms are based on neighbourhood searches. Examples of such is solving normals (section 2.5), clustering (section 2.6.2) and the ICP algorithm itself (section 2.10.1). A brute force nearest-neighbour search has an asymptotic running time of $O(NM)$, given two point clouds where N is the number of points in the source cloud and M is the number of points in the target cloud. $N = M$ in the case of searching through the same cloud.

kD-trees are binary search trees that can speed up neighbourhood searches. The concept was introduced by Bentley (Bentley (1975)), but this section will focus on a specialized case only involving spatial points. The idea is that each point has k dimensions, and they are ordered by one dimension at a time. Each layer in the tree is ordering the points based on a single dimension. The tree structure is based on that a node has two children, where the value of the left child's $i'th$ dimension is smaller than that of the node itself. The value of the right child's $i'th$ dimension is larger than the node. This is valid for some $0 \leq i \leq k-1$. The ordering dimension i is typically circulated between 0 and $k-1$ for each level of the tree. Figure 2.11 shows the most basic structure of a kD-tree.

2.9.1 Contructing a kD-tree

A kD-tree can be constructed by inserting one point at a time into it. The construction method described here is based on the method proposed by Bentley Bentley (1975), but has a few simplifications as it only deals with spatial points. Algorithm 1 shows how the kD-tree is constructed.

The expected height of this tree is expected to be $O(\log(N))$, where N is the number of points inserted. For each insertion, the expected running time is therefore $\log(N)$, meaning that total kD-tree construction time is expected to be $O(N\log(N))$.

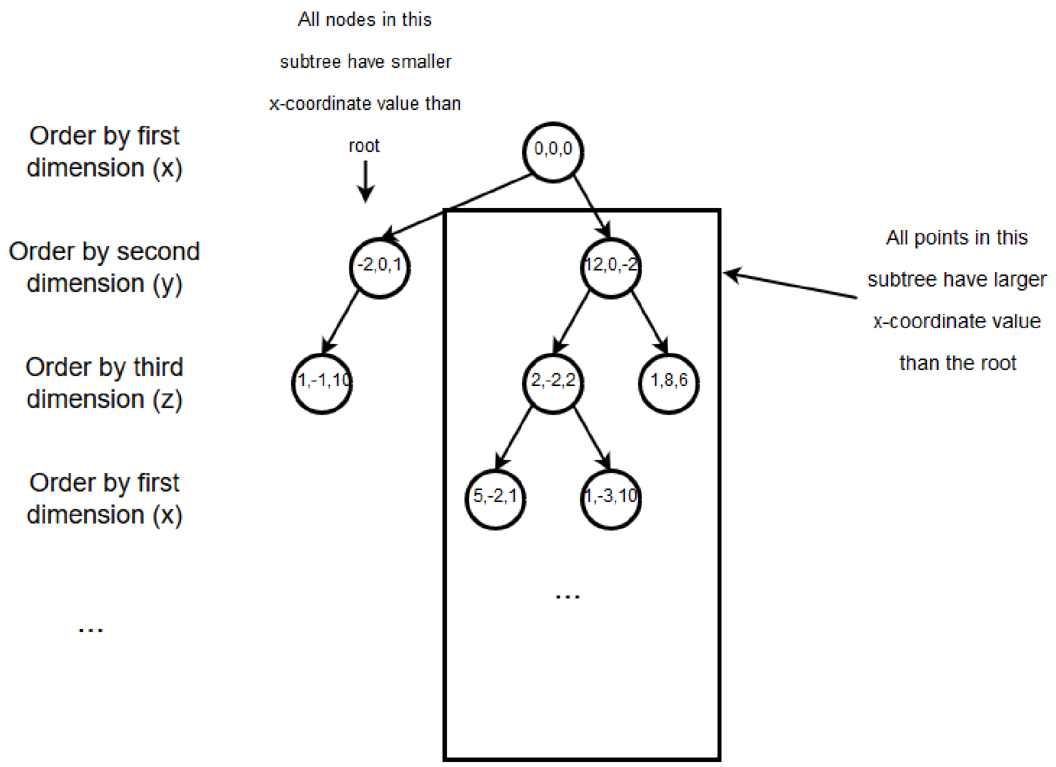

All nodes in this subtree have smaller x-coordinate value than root
Order by first dimension (x)
Order by second dimension (y)
Order by third dimension (z)
Order by first dimension (x)
...
0,0,0
-2,0,1
12,0,-2
1,-1,10
2,-2,2
1,8,6
5,-2,1
1,-3,10
...
All points in this subtree have larger x-coordinate value than the root

Figure 2.11: Figure showing the ordering of a kD-tree with 3D spatial points. The figure is showing how the kD-tree is order at one dimension at each level of the tree. The tree guarantees that all left children are smaller on the dimension of that level, and that the right children are larger on that dimension. Figure is taken from the pre-project by Isachsen (2017).

Algorithm 1 constructTree

```
Require: A point list $L$ to create kD-tree from
Require: The current dimension $0 \leq d \leq k-1$, typically starting at 0
  if len($L$) = 0 then
    return nil
  end if
  if len($L$) = 1 then
    return $L[0]$
  end if
  $i \leftarrow$ median_index($L$)
  node $\leftarrow L[i]$
  $d = ((d+1) \mod k)$
  node.lchild $\leftarrow$ constructTree($L[:i]$, $d$)
  node.rchild $\leftarrow$ constructTree($L[i+1:]$)
```

2.9.2 kD-tree search

While the method described in section 2.9.1 for kD-tree construction is relatively simple, a good searcing strategy is slightly harder. The idea is to use the kD-tree structure to limit the search, such that only a subset of the points need to be visited. The most common way to search a kD-tree is either to do a radius search or a k-nearest search. The radius search starts with a point p and a fixed radius r around that point. The search should return all points in the kD-tree within this radius of the point. The k-nearest search (k is not referring to the number of dimensions here) is searching the k nearest points of a point p. This is similar to a radius search, where the radius r starts as infinite and is reduced to the furthest of the currently closest k points. Since point cloud registration generally uses a $k = 1$-nearest search, the k-nearest search strategy is summarized in algorithm 2. Note that the algorithm easily can be extended to a radius search by fixing the radius r. The listed algorithm is based on a basic kD-tree search algorithm (Friedman et al. (1977)). The main difference is that it uses a size-limited stack (the circular stack) to backtrack the search. It also avoids using the hyper-rectangle bounds and instead compares the node to the distance of the hyperplane of the current dimension. Figure 2.12 shows how a kD-tree search is done using an example.

2.9.3 kD-tree Optimizations

The kD-tree described in section 2.9.1 is not necessarily efficient for storage or search. One important optimization that can be done on kD-trees is called left-balancing. This structure is shown in figure 2.13. The main idea is that nodes are related to each other by their index in the tree. The following equation can be used to find the left child of a node

$$i_{\text{leftchild}} = (2 \cdot i_{\text{node}}) + 1$$

and the right child can be found using the formula

$$i_{\text{leftchild}} = (2 \cdot i_{\text{node}}) + 2$$

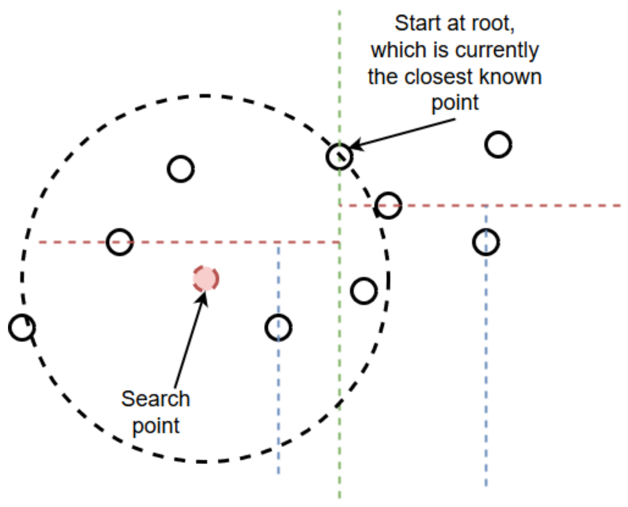
. Similarily, the parent node can be found with the formula

$$\frac{i_{child} - 1}{2}$$

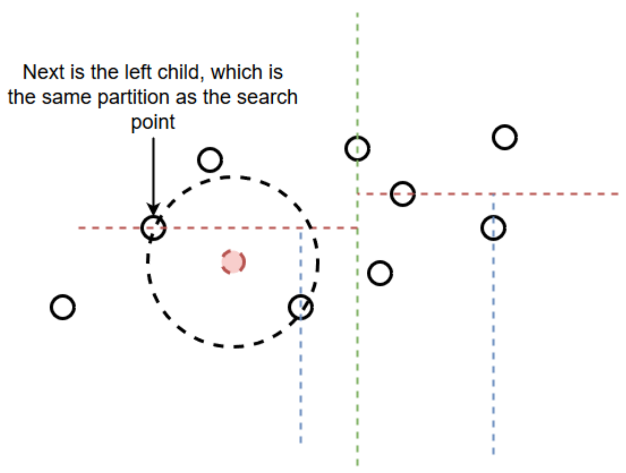
. This structure will have size exponentially in terms of tree depth. Therefore, the depth should be limited.

Algorithm 2 Kd-tree search.

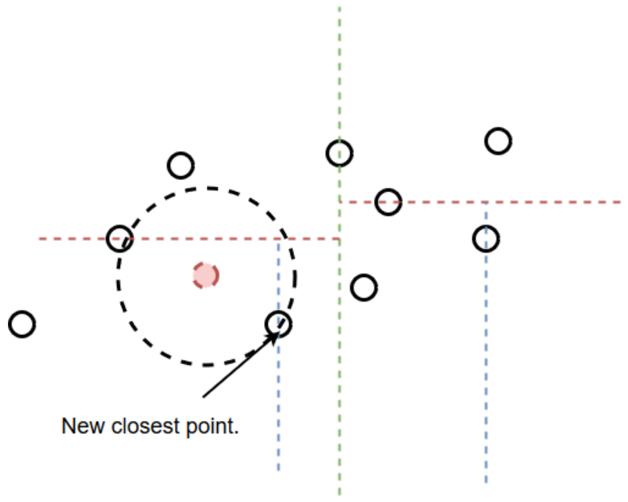
```
Require: A point $p$ to find nearest neighbour of
Require: root, the root of the kD-tree.
  $d \leftarrow 1$
  $s \leftarrow$ Initialize stack
  $s$.push(root)
  $c \leftarrow$ inf
  while $s$ is nonempty do
    Skip this node if it is nil
    $n \leftarrow s$.pop()
    if First discovery of $n$ then
      $c \leftarrow \text{dist}(p, n)$ if $\text{dist}(p, n) \leq c$
      $s$.push($n$)
      if $p[d] \leq n[d]$ then
        $s$.push($n$.leftchild)
      else
        $s$.push($n$.rightchild)
      end if
      $d = ((d + 1) \mod k)$
    else
      if $|p[d] - n[d]| \leq c$ then
        if $p[d] \leq n[d]$ then
          $s$.push($n$.rightchild)
        else
          $s$.push($n$.leftchild)
        end if
        $d = ((d + 1) \mod k)$
      else
        $d = ((d - 1) \mod k)$ {Assuming mod operator gives positive values}
      end if
    end if
  end while
  $c$ is the shortest distance for point $p$
```



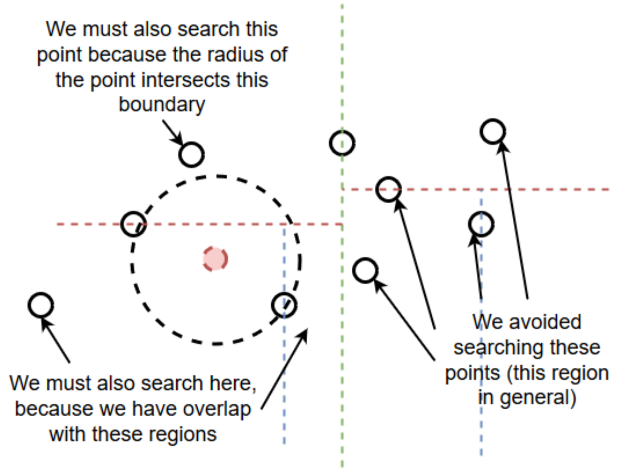
(a) Step 1


Next is the left child, which is the same partition as the search point

(b) Step 2

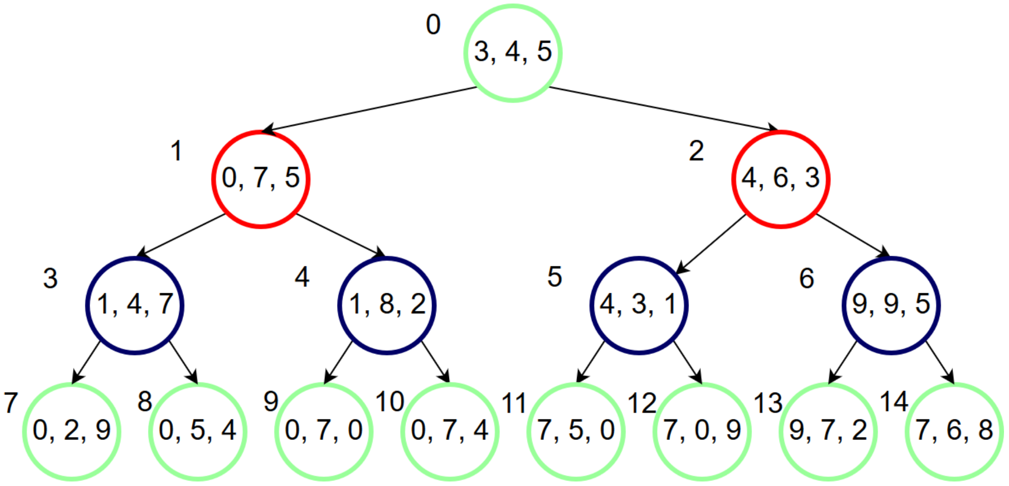


(c) Step 3

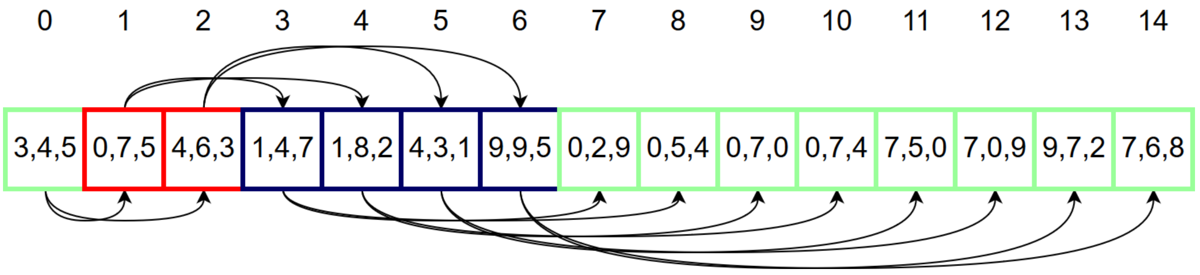

We must also search this point because the radius of the point intersects this boundary
We must also search here, because we have overlap with these regions
We avoided searching these points (this region in general)

(d) Step 4

Figure 2.12: Figure showing some steps of a k-nearest search using a kD-tree. (a) shows that the search starts at the root, which is the currently known closest point indicated with the search radius. The search continues to evaluate the root left child, as the x-dimension value is smaller than that of the root in (b). Again, the same procedure is applied for (c) in the y-dimension. As these steps are performed, the current closest point is updated. When going up the tree, the current best radius has been updated such that a portion of the tree doesn't have to be searched in (d). This particular case has 2D data, but the idea is similar for 3D. This figure is copied from the pre-project by Isachsen (2017)



(a) kD-tree represented as a tree



(b) kD-tree represented as an array

Figure 2.13: Figure showing a left-balanced kD-tree in (a), and the same tree represented as an array (b). Each subtree is separated on the median, which enables this dense representation of the tree. The array structure in (b) is easily stored in computer memory without using pointer structures and can then easily be transferred to the GPU memory.

The depth can be minimized by balancing the tree. While constructing the tree, instead of adding points by the order they are listed, it is possible to partition the tree on the median of each dimension. The median is inserted as the root of that subtree, meaning that an equal amount of elements are placed in the left and right subtree of that node. With this optimization, the tree will at maximum have a size of $2 \cdot n$ where n is the number of points in the point cloud.

2.10 Point cloud registration

Point cloud registration is about aligning a measured point cloud to a reference point cloud or reference model. This section will explain the basics of registration in general and outline some of the research done in the field. It will also introduce the ICP algorithm suggested by Besl and McKay (1992).

Point cloud registration have been an active research since the 1990's and there have been many suggested algorithms to solve the problem. An early and very popular approach is the ICP algorithm which works by finding corresponding points between an object model and a measured point cloud (Besl and McKay (1992)). It does so by regarding the pairs of points from all points in the measured point cloud to it's closest point in the model object. Based on these correspondences, it finds a matrix that transform the measured cloud in such a way that it will minimize the distance of the correspondences. The algorithm will converge to some local minimum after applying these two steps iteratively (closest point correspondence and transformation).

The reminder of this document will use the terminology used by Besl and McKay (1992). They consider registration of two point clouds. One of the point clouds is referred to as the point shape denoted as P and a model shape denoted as X (in PCL terminology, they are ferred to as source cloud and target cloud respectively). These clouds are also denoted as a test/data set and reference shape. P_t and P_m will also be used in this project as test shape and model shape respectively. The P notation is used to emphasize that the sets are in fact point clouds in this project (but doesn't have to be that in the general formulation of the ICP algorithm). The goal of a registration algorithm is to find a transformation that is applied to P_t such that P_t and P_m have an overlap that minimizes some error function based on the distance between points in the two point clouds.

2.10.1 The ICP algorithm

Point cloud registration have been an active research since the 1990's. The Iterative Closest Point (ICP) algorithm was first suggested by Besl and McKay (1992). The algorithm works by minimizing the squared distance between a registration cloud P and a model cloud X (or P_m). The following formula is used:

$$f(\vec{q}) = \frac{1}{N_p} \sum_{i=0}^{N_p} \| \vec{x}_i - R(\vec{q}_R)\vec{p}_i - \vec{q}_T \| \tag{2.1}$$

$f(\vec{q})$ is the objetive function to minimize. $\vec{x}_i$ is the i'th point in the model set corresponding to a point $\vec{p}_i$ in the measurement point cloud. The correspondence is found using closest point which will be explained later. $R(\vec{q}_R)$ is transformation matrix based on the rotational quaternion q_R and $\vec{q}_T$ is a translation vector. The goal is to find $R(\vec{q}_R)$ and $\vec{q}_T$ that minimizes the objective function $f(\vec{q})$.

The next equations are some preliminary equations used later to solve the unknowns.

$$\vec{\mu}_p = \frac{1}{N_p} \sum_{i=0}^{N_p} \vec{p}_i \tag{2.2}$$

$$\vec{\mu}_x = \frac{1}{N_x} \sum_{i=0}^{N_x} \vec{p}_x \tag{2.3}$$

These equations is simply the centroid of the point clouds P and X. By finding the cross-covariance matrix, it's possible to determine the transformation that will reduce the error. The idea is based on the eigenvectors of the cross-covariance matrix. The cross-covariance matrix is defined by:

$$\Sigma_{px} = \frac{1}{N_p} \sum_{i=0}^{N_p} [(\vec{p}_i - \vec{\mu}_p)(\vec{x}_i - \vec{\mu}_x)^t] = \frac{1}{N_p} \sum_{i=0}^{N_p} [\vec{p}_i \vec{x}_i] - \vec{\mu}_p \vec{\mu}_x{}^t \tag{2.4}$$

In order to find a matrix that gives a rotational quaternion as the unit egenvector corresponding to the largest eigenvalue, the following steps are required.

$$A = \Sigma_{px} - \Sigma_{px}^T \tag{2.5}$$

A is just a intermediate step to compute the column vector from the cyclic components of A.

$$\Delta = \begin{bmatrix} A_{23} & A_{31} & A_{12} \end{bmatrix}^T \tag{2.6}$$

The final matrix that is used to compute the rotation quaternion is the following 4x4 matrix:

$$Q(\Sigma_{px}) = \begin{bmatrix} tr(\Sigma_{px}) & \Delta^T \\ \Delta & \Sigma_{px} + \Sigma_{px}^T - tr(\Sigma_{px}) I_3 \end{bmatrix} \tag{2.7}$$

I_3 is simply a 3x3 identity matrix. $tr(a)$ is the trace of the matrix, which is simply the sum along the matrix diagonal. By solving the eigenvectors and eigenvalues of this matrix, it's possible to find the rotation quaternion. The rotation quaterion $\vec{q}_R = \begin{bmatrix} q_0 & q_1 & q_2 & q_3 \end{bmatrix}^t$ is the eigenvector corresponding to the largest eigenvalue.

The goal of this derivation was to obtain the optimal rotation of the point cloud. This is obtained from the rotation quaternion $\vec{q}_R$ using the following definition

$$R = \begin{bmatrix} q_0^2+q_1^2-q_2^2-q_3^2 & 2(q_1q_2-q_0q_3) & 2(q_1q_3+q_0q_2) \\ 2(q_1q_2+q_0q_3) & q_0^2+q_2^2-q_1^2-q_3^2 & 2(q_2q_3-q_0q_1) \\ 2(q_1q_3-q_0q_2) & 2(q_2q_3+q_0q_1) & q_0^2+q_3^2-q_1^2-q_2^2 \end{bmatrix} \tag{2.8}$$

Finally the translation component can be found with the following:

$$\vec{q}_T = \vec{\mu}_x - R(\vec{q}_R)\vec{\mu}_p \tag{2.9}$$

The operation that transforms this cloud is denoted as the least squares quaternion operation and is denoted

$$(\vec{q}, d_{ms}) = Q(P, X) \tag{2.10}$$

d_{ms} is the mean square point matching error. Before stating the algorithm itself, some words on closest point estimation is required.

The problem of finding closest points is an involved question Rusinkiewicz and Levoy (2001). More of this is explained in section 3.1.2, but for now, the method described by Besl and McKay (1992) will be used. The closest point operator is denoted by $C(P, X)$ which ends in a new point set Y. The closest point in X of a point $\vec{p}$ in P is simply the closest euclidean distance:

$$d(\vec{p}, X) = min_{\vec{x}\in X}||\vec{x}-\vec{p}|| \tag{2.11}$$

Based on all these definitions, the ICP algorithm itself can be stated. Figure 2.14 shows this algorithm in a 2D case, which conceptually works the same way as in 3D.

1. Initialize variables. Set $P_0 = P$, $q_0 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^t$ and $k = 0$. q_i is just a combined vector where the first 4 components are the rotation quaternion q_R and the last three is the translation component q_T.

2. Perform the following steps until convergence. That is when the change of mean square error $d_k - d_{k+1}$ drops below a threshold τ, or the maximum number of iterations is reached etc.

   (a) Compute the closest points $Y_k = C(P_x, X)$

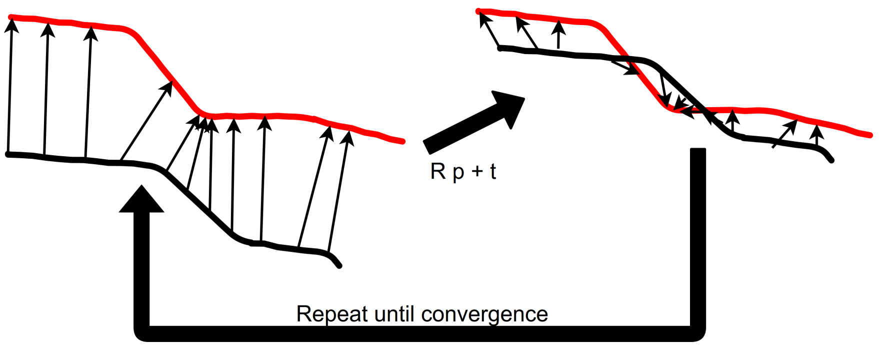


Figure 2.14: Figure showing the steps of the ICP algorithm in a 2D case. The concept is the same in the 3D case. The first step is the correspondence estimation indicated by the black arrows in on the left side. The rotation matrix and translation vector minimizes the error of the found correspondences. The results is shown in the right of the figure. This is repeated until a convergence condition is met.

(b) Compute the registration $(\vec{q}_k, d_k) = Q(P_0, Y_k)$

(c) Apply registration with $P_{k+1} = q_k(\vec{P}_0)$. $q_k(P)$ is just application of the transformation derived from the translation and rotation quaternion q_k on the point cloud P.

(d) Test if the mean square error is below the threshold: $d_k - d_{k+1} < \tau$. If this is the case, the algorithm terminates.

Besl and McKay (1992) gives a proof that this algorithm is guaranteed to converge. This proof as well as other parts of the paper is left out here because they are not that relevant for an implementation of the algorithm. Another large part of the paper involves an analysis of finding the global minimum as the algorithm tends to get stuck at local minima (Besl and McKay (1992)). This is usually not part of the basic ICP implementation (as with PCL (Open Perception Foundation (2017b))). This is because the initial alignment is an important question regarding the ICP registration, and depending on the application this might be known. Therefore, the general implementation of ICP in PCL does not perform this extra step to improve probability of finding the global minimum.

The performance of this algorithm is highly dependent on the individual steps. Most of the operators used are linear in terms of point cloud size. The exception is the closest point estimation which in the case of closest neighbour search runs in $N_p N_x$ in worst case and $N_p \log(N_x)$ on average by using a kD-tree as specified in section 2.9. This is without considering the number of iterations. However, as shown in section 6.4.2, this number is usually low compared to the size of the point clouds, and convergence is likely within 50 iterations.

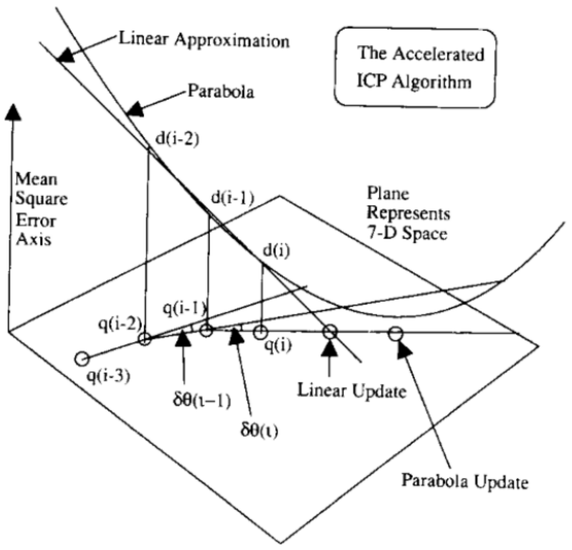


Figure 2.15: Figure showing an accelerated ICP algorithm as suggested by Besl and McKay (1992). The figure is taken from their paper on the ICP algorithm.

2.10.2 Non linear ICP (accelerated ICP)

As a final comment on the ICP algorithm by Besl and McKay (1992), they suggested an accelerated version of the ICP algorithm involving interpolation between consecutive iterations of the ICP algorithm. Figure 2.15 is taken from Besl and McKay (1992). The figure shows how the quaternion is interpolated between consecutive iterations to speed up the convergence of the algorithm.

2.10.3 Using point to plane as transformation minimization

As described in section 3.1, there are many variations of the ICP algorithm, and one such variation can be obtained by changing the transformation estimation step with a linear least squares optimization like Chen and Medioni (1992). Previous work by Isachsen (2017) showed that the linear least squares transformation estimator gave good results. Though section 3.1.5 is about ICP variations of different transformation estimators, this section will cover the basic mathematical background for transformation estimation using hte linear least squares estimator. This section is based on the derivation performed by Low (2004).

The main idea of the point to plane linear least squares error metric is that instead of minimizing the distance from point-to-point (like Besl and McKay (1992)), it minimizes the distance between a point and the tangent plane of the corresponding point. The tangent plane is represented by the normal vector of that point, which can be found using the technique described in section 2.5. The optimization becomes the following:

$$M_{\text{opt}} = \text{argmin}_Q \sum_i ((M \cdot p_{t,i} - p_{m,i}) \cdot n_i)^2 \tag{2.12}$$

Where Q_{opt} and Q are 4x4 homogeneous matrices that transforms the data cloud to the model cloud. $p_{t,i}$ and $p_{c,i}$ is the i'th point in data cloud and closest point cloud Y (as done in section 2.10.1). n_i is the normal of point $p_{m,i}$. As mentioned this equation is about finding the best M such that the distance from $p_{t,i}$ and the tangent plane of $p_{c,i}$ is minimized.

As the transformations of M are rigid, it can be written as a combination of a translation T and a rotation R.

$$R = T \cdot R \tag{2.13}$$

T and R are standard matrices used for homogeneous transformations where R is divided into a rotation done on each axis (α around first axis β around second axis and γ around third axis).

$$R = R(\alpha) \cdot R(\beta) \cdot R(\gamma) \tag{2.14}$$

As these are quite standard matrices for 3D transformations, I will not describe them further, but a complete description can be found in Low (2004). As these matrices contain sinusoidals, they are quite hard to solve. In order to simplify them, an approximation $\theta \simeq 0$ is used: $\sin\theta \simeq \theta$ and $\cos\theta \simeq 1$. Because of these simplifications the rotation matrix can be simplified by the following when $\alpha, \beta, \gamma \simeq 0$:

$$R \simeq \hat{R} = \begin{bmatrix} 1 & -\gamma & \beta & 0 \\ \gamma & 1 & -\alpha & 0 \\ -\beta & \alpha & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.15}$$

Note that this also assumes that multiplications of values $\simeq 0$ is approximated to 0. Using the $\hat{R}$ matrix, the point to plane distance for a single point can be written extended as a single expression:

$$\begin{aligned}(\hat{Q}\cdot p_{t,i} - p_{c,i})\cdot n_i &= \left(\hat{Q}\cdot\begin{pmatrix} p_{t,i,x}\\ p_{t,i,y}\\ p_{t,i,z}\\ 1\end{pmatrix} - \begin{pmatrix} p_{c,i,x}\\ p_{c,i,y}\\ p_{c,i,z}\\ 1\end{pmatrix}\right)\cdot\begin{pmatrix} n_{i,x}\\ n_{i,y}\\ n_{i,z}\\ 0\end{pmatrix}\\ &= [(n_{i,z}p_{t,i,y} - n_{i,y}p_{t,i,z})\alpha + (n_{i,x}p_{t,i,z} - n_{i,z}p_{t,i,x})\beta + (n_{i,y}p_{t,i,x} - n_{i,x}p_{t,i,y})\gamma\\ &\quad + n_{i,x}t_x + n_{i,y}t_y + n_{i,z}t_z]\\ &\quad - \left[n_{i,x}p_{c,i,x} + n_{i,y}p_{c,i,y} + n_{i,z}p_{c,i,y} - n_{i,x}p_{t,i,x} - n_{i,y}p_{t,i,y} - n_{i,z}p_{t,i,z}\right]\end{aligned} \tag{2.16}$$

Here, the c and t indices are used to indicate which point cloud they belong to. The i is the point pair index, and x, y, z is the x, y and z components of the points. The notation is the same as previous equations. The next trick is to arrange all of these expressions into a matrix for each $1 \le i \le N$ into a matrix expression (N is number of correspondences):

$$Ax - b \tag{2.17}$$

The A, x and b values are taken from the equation and is defined as the following:

$$b = \begin{pmatrix} n_{1,x}p_{c,1,x} + n_{1,y}p_{c,1,y} + n_{1,z}p_{c,1,y} - n_{1,x}p_{t,1,x} - n_{1,y}p_{t,1,y} - n_{1,z}p_{t,1,z}\\ n_{2,x}p_{c,2,x} + n_{2,y}p_{c,2,y} + n_{2,z}p_{c,2,y} - n_{2,x}p_{t,2,x} - n_{2,y}p_{t,2,y} - n_{2,z}p_{t,2,z}\\ \vdots\\ n_{N,x}p_{c,N,x} + n_{N,y}p_{c,N,y} + n_{N,z}p_{c,N,y} - n_{N,x}p_{t,N,x} - n_{N,y}p_{t,N,y} - n_{N,z}p_{t,N,z}\end{pmatrix} \tag{2.18}$$

$$x = \begin{pmatrix}\alpha & \beta & \gamma & t_x & t_y & t_z\end{pmatrix} \tag{2.19}$$

and A is defined as:

$$A = \begin{pmatrix} n_{1,z}p_{t,1,y} - n_{1,y}p_{t,1,z} & n_{1,x}p_{t,1,z} - n_{1,z}p_{t,1,x} & n_{1,y}p_{t,1,x} - n_{1,x}p_{t,1,y} & n_{1,x} & n_{1,y} & n_{1,z}\\ n_{2,z}p_{t,2,y} - n_{2,y}p_{t,2,z} & n_{2,x}p_{t,2,z} - n_{2,z}p_{t,2,x} & n_{2,y}p_{t,2,x} - n_{2,x}p_{t,2,y} & n_{2,x} & n_{2,y} & n_{2,z}\\ \vdots & & & & & \\ n_{N,z}p_{t,N,y} - n_{N,y}p_{t,N,z} & n_{N,x}p_{t,N,z} - n_{N,z}p_{t,N,x} & n_{N,y}p_{t,N,x} - n_{N,x}p_{t,N,y} & n_{N,x} & n_{N,y} & n_{N,z}\end{pmatrix} \tag{2.20}$$

While these matrices might look complicated, they are simply a matrix representation of equation 2.16. Each of the elements of the A and b matrices are taken from certain parts of equation 2.16. The advantage of representing the data as equation 2.17 is that Singular value decomposition can be used to solve the unknowns. This is because

$$\min_{\hat{M}} \sum_i \left(\left(\left(\hat{M} \cdot p_{t,i} - p_{c,i} \right) \cdot n_i \right)^2 \right) = \min_x |Ax - b|^2 \tag{2.21}$$

which again gives the following:

$$x_{\text{opt}} = \text{argmin}_x |Ax - b|^2 \tag{2.22}$$

The last step is now simply to solve this linear least-squares problem. This is done using a Singular-Value Decomposition (SVD) decomposition on A:

$$A = U\Sigma V^T \tag{2.23}$$

Where T denotes the transpose. The preudo-inverse of A ($A^+ = V\Sigma^+ U^T$) is used to solve the equation. Σ^+ is found by finding the inverse of the non-zero elements of Σ and otherwise leaving the other elements unchanged. This solves the optimal vector:

$$x_{opt} = A^+ b \tag{2.24}$$

which in turn in inserted into the transformation $M = T(t_{x,\text{opt}}, t_{y,\text{opt}}, t_{z,\text{opt}}) \cdot R(\alpha_{\text{opt}}, \beta_{\text{opt}}, \gamma_{\text{opt}})$. Note that the rotation matrix R is used instead of the approximated $\hat{R}$. Even with the small angle approximation in equation 2.15 Low (2004) claims that this method is relatively good even for angles up to 30 degrees.

CHAPTER 3

Related Work

While section 2 focused on the mathematical background and the basic formulation of point cloud registration, this section will focus more on work related to the goal of the project itself. The first section will be about different variations of the ICP algorithm based on the work done by Rusinkiewicz and Levoy (2001). Section 3.2 will go through an variation of the ICP algorithm based voxelization of the point cloud as described by Marden and Guivant (2012). Section will look at accelerating the nearest neighbour lookup by performing kD-tree search on the GPU. This is based on work by Qiu et al. (2009).

This section will also briefly discuss the Kinect fusion algorithm in section 3.4 and the Kinfu algorithm in section 3.5. These are based on work by Newcombe et al. (2011); Pirovano et al. (2013) respectively. Finally, section 3.7 will briefly discuss alternatives to the ICP algorithm, and is based on the list given by Magnusson (2013) in his PhD thesis. While these last sections have less direct relevance for the project, they are either included to give wider view of registration in general, and to show some thought that are included in this project.

3.1 Variations of the ICP algorithm

This section is based on the paper by Rusinkiewicz and Levoy (2001) called ”Efficient variants of the ICP algorithm”. The paper divides the ICP algorithm into several stages, each having multiple variants. This section will follow the same structure as this paper regarding the different stages and variations provided.

Rusinkiewicz and Levoy (2001) divides the ICP algorithm into the following stages: selection, matching, weighting, rejecting, assigning error metric and minimizing error metric. The algorithm performs the steps in that order.

3.1.1 Selection of points

This part is about selection of points in the data cloud (which is aligned with the model cloud). Besl and McKay (1992) simply use all points in the data cloud. Turk and Levoy (1994) and Masuda (2002) use a subsample of the points. While Masuda use a random subsample that is different for each iteration, Turk et al. use uniform subsampling. Weik (1997) use points with high intensity gradient based on the luminance of the points, using some color or gray scale intensity along the depth data from a scanner in order to aid the registration process. Another approach is to use both data cloud and model cloud in the selection stage, which was suggested by Guy Godin (1994).

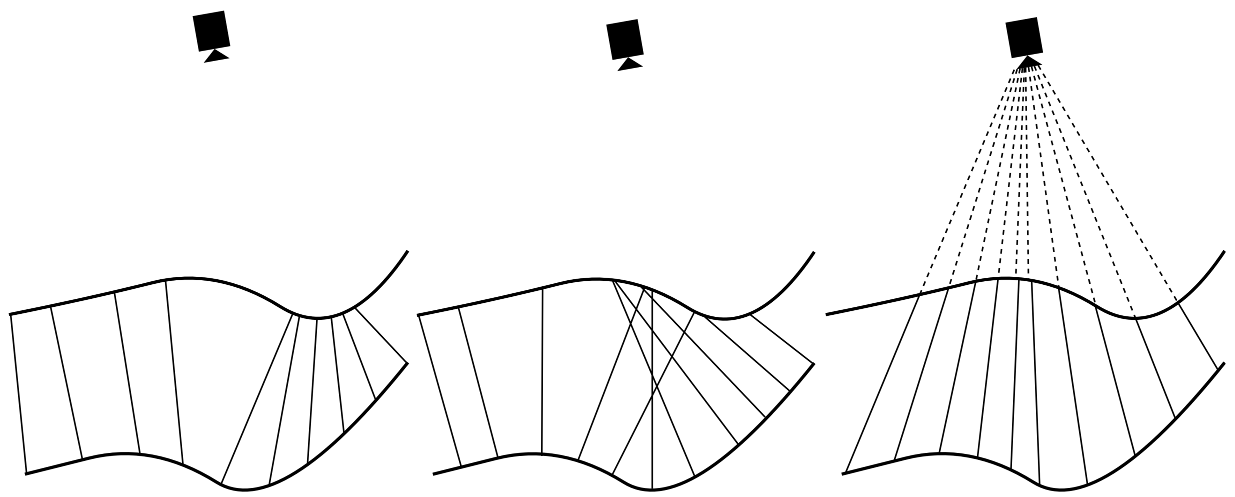
Finally Rusinkiewicz and Levoy (2001) have their own suggested point selection strategy that is based on selecting points in such a way that the distribution of normals is maximized. This method is based on the observation that in certain registration tasks, there are small features that are vital to the registration (like shallow grooves). This technique will try to distribute the points such that these smaller features have more importance if their normals are significantly different from the remaining model.

The results from this part is that all methods are equally good on simple point cloud sets, but the main difference is when the point clouds are featureless surfaces (like planes), except for a few significant features (like grooves). In this case, the subsampling strategies failed because they selected to few points in the significant parts of the model. Only the point selection strategy suggested by Rusinkiewicz and Levoy was able to get convergence on such test sets. Because of these results, all data cloud points are used for this project, though certain filters of the data cloud can utilize some of the benefits of the point selection method suggested by Rusinkiewicz and Levoy.

3.1.2 Matching points

After the points in the data cloud (or both data and model cloud in the case of the suggestion by Guy Godin (1994)), the ICP algorithms must find corresponding points in the model cloud. The following list gives a summary of some of the possible point correspondence methods. Some of them are shown in figure 3.1.

- Use the closest point in the model cloud. This is the method used by Besl and McKay (1992). See figure 3.1a.



(a) Closest point (b) Normal shooting (c) Back projection

Figure 3.1: Figure showing examples of different correspondence estimators as summarized by the work done by Rusinkiewicz and Levoy (2001). (a) shows the closest point estimator, (b) shows the normal shooting estimator, and (c) shows the point projection estimator. These are the correspondence evaluators used in this project which is implemented in PCL. Only the closest point correspondence estimator is used for the GPU implementation.

- Normal shooting. From a point in the data cloud, compute the normal and trace this along it's direction. Find the intersection point with the destination cloud. This was the method used by Chen and Medioni (1992). See figure 3.1b.

- Project the point in the source cloud onto the destination cloud from the point of view of the model cloud's camera (trace the ray going from the camera of the model cloud' through the data point cloud source point. Find the point where this ray intersects the model cloud). This method was used by Blais and D. Levine (1995) and is referred to as "reversed calibration". See figure 3.1c.

- Perform a search in the model cloud. Use the projected point as described in the previous method as a starting point of the search. Use some error metric when performing the search (for example point to point distance like Benjemaa and Schmitt (1997))

- Guy Godin (1994) also included some compability metrics. The methods described above must also be validated based on some compability measure. Godin did this based on the color, but other approaches can be used (Rusinkiewicz and Levoy (2001)).

To summarize the work done by Rusinkiewicz and Levoy (2001), the normal shooting variants are the fastest converging methods in terms of number of iterations. However, they are unable to give convergence for certain types of point clouds, where only the closest point matching gives convergence. When considering performance, the projective methods give the fastest convergence when considering the running time. All three are quickly evaluated in this project, but only the closest point correspondence estimator is used for the GPU implementations.

3.1.3 Weighting of pairs

When the point correspondences are found, it is possible to weight the pairs. Guy Godin (1994) suggested multiple weighting schemes. The simplest method is to use constant weighting (this is very common as weighting seems to have less impact on performance than many of the other measures Rusinkiewicz and Levoy (2001)). Godin suggested a weighting scheme using point distances and normal compabilites. Point distance weighting can be defined by Weight $= 1 - \frac{Dist(p1,p2)}{Dist_{\max}}$, and the normal compability can be defined by Weight $= n_1 \cdot n_2$. Rusinkiewicz and Levoy also suggests a scheme based on the uncertainty of the surface normals etc. (Rusinkiewicz and Levoy (2001)).

Rusinkiewicz and Levoy concluded that weighting has little effect on the end results of the registration. Therefore, weighting is not evaluated or implemented in this project.

3.1.4 Point pairs rejection

There are multiple approaches to doing point pairs rejection. The following list is based on the work done by Rusinkiewicz and Levoy (2001).

- Reject points that are further apart than a user specified distance.
- Rejection of a certain percentage of the points based on some error metric. Pulli (1999) rejected the worst 10 % of the points.
- Reject pairs which have a distance larger than some multiple of the standard deviation of distances. This was suggested by Masuda (2002).
- Reject points that are not consistent with neighbouring points. One such metric as used by Dorai et al. (1998), is to verify that the difference in distance between two neighbouring pairs is less than a threshold based on the largest distance of the point pairs.
- Reject pairs that contain points on the boundaries. Given partial overlaps, this metric makes the non-overlapping regions less significant for the final registration.

The idea behind point pair rejection is that the point clouds usually have partial overlaps. By setting a rejection criteria, the non-overlapping regions will have less impact on the final registration. Rusinkiewicz and Levoy (2001) suggested to always use the latter rejection metric. They also found that rejecting 10 % of the points gave slower convergence speed. Otherwise, there was little difference in using point rejection compared to not using it. The box correspondence rejector described in section 4.5 is based on the suggestion of rejecting boundary points, where a box is placed around the object such that boundary correspondences are rejected.

3.1.5 Error metric

The most common way to define the error metric is the squared distances of the all point pairs found in the matching stage. By using this metric, there are multiple ways to estimate the optimal transformation that reduces the error, but the main advantage is that there exists a closed form solution. The first approach was to use the singular-value decomposition method (SVD) (Arun et al. (1987)). Horn (1987) introduced the use of orthonormal matrices. Dual quaternions were introduced by Walker et al. (1991). Another possible improvement is to include color data into the error metric (Johnson and Hebert (1997)). The final suggestion is slightly different and was introduced by Chen and Medioni (1992). The idea is to sum the squared distances from the points in the data point cloud to the plane containing the matching point in the model cloud that is oriented perpendicular to the destination point. There is no closed for solution to this, but multiple non-linear methods are possible. This method was proposed by Chen and Medioni (1992).

In this project however, the the unit quaternion method described by Horn (1987) (which is the formulation used by Qiu et al. (2009)) is implemented for the GPU. Additionally the point to plane linear least squares optimization introduced by Low is implemented on the GPU (Low (2004)) (as described in section 2.10.3). PCL has implementations of the dual quaternion and SVD approach which is also briefly evaluated.

3.1.6 Minimization

Again, there are multiple alternatives here. However, Rusinkiewicz and Levoy indicates that multiple of the solutions described are very slow. These involve multiple starting tranformations (Simon (1996)), stochasic search using simulated annealing (Blais and D. Levine (1995)) and iterative minimization with randomly selected subsets of points, and then selecting the optimal result using a robust metric (Masuda (2002)).

Since the methods mentioned above are relatively slow, only two of the methods for finding a new transformation that minimizes the error metric will be considered. This includes the iterative method used in the standard ICP algorithm by Chen and Medioni (1992); Besl and McKay (1992). These methods provide a relatively light-weight search strategy while still obtaining convergence within a few iterations (for example 20 iterations for many normal point cloud sets) (Rusinkiewicz and Levoy (2001)).

3.2 Voxelized space for approximate nearest neighbours

Section 3.1 focused on the steps in the ICP algorithm, as well as multiple variations of each step. This section will go through a simplistic point correspondence estimator which is the foundation of the implementation in section 4.4.5 and 4.4.6. This registration algorithm was introduced by Marden and Guivant (2012).

3.2.1 Description of algorithm

The algorithm is based on the ICP formulation done by Besl and McKay (1992). The main contribution by Marden and Guivant (2012) is the point correspondence estimation based on voxelization of the model cloud, instead of the standard closest point estimator. The following steps are used for correspondence estimation:

- The x_{min}, x_{max}, y_{min}, y_{max}, z_{min} and z_{max} is found by inspecting the target cloud and finding the min and max values of each dimension.
- Divide the target point cloud space into an uniform voxel space. Each voxel is represented by the centroid of the points that fall within the voxel's bounds. This is simply done by calculating the mean of all points whose dimensions are within the bounds of the voxel.
- Correspondence estimation is done in a similar manner as the voxel space construction. Each point in the source cloud is matched with the centroid of the voxel which encapsulates the point. The centroid was found in the previous step.

The voxel index (i, j, k) of each point is found using the following equations.

$$i = \left\lfloor \frac{x - x_{min}}{l} \right\rfloor, j = \left\lfloor \frac{y - y_{min}}{l} \right\rfloor, k = \left\lfloor \frac{z - z_{min}}{l} \right\rfloor \tag{3.1}$$

where l is the side lengths of the voxels.

Finally, the voxel's can be updated with new points with the following equation.

$$N'_{ijk} = N_{ijk,existing} + N_{ijk,new} \tag{3.2}$$

$$\mu_{ijk} = \frac{\mu_{ijk,existing} N_{ijk,existing} + \mu_{ijk,new} N_{ijk,new}}{N_{ijk,existing} + N_{ijk,new}} \tag{3.3}$$

where μ_{ijk} is the voxel centroid at index i, j, k and N_{ijk} is the number of points that are inside voxel at index i, j, k. These equations are simply increasing the new point-count inside each voxel and updating the new centroid by considering the new points as well as the previous ones.

Summary of algorithm While the results from the work gives some good registrations, they don't have an empirical measure to prove that it's actually better than a PCL implementation which they compare it with. They show a few examples where they perform better than the PCL implementation of registration. While it is clear that the performance in improved using this strategy, they are unable to prove that their approach is better in the general case.

While the suggested approach is unable to fully convince that there is an improvement in registration quality, it suggests a really simple method with great performance. Given the quick lookup time, the algorithm might justify running more iterations of the ICP algorithm.

In their future work, they mention that a more robust testing framework could be applied to the method. Another thing they note is that they require a good initial transformation estimation in order to function. This is also the case for the standard ICP formulation, but is more extreme in the voxelized case. They also suggest that a GPU implementation can be done. Each of these points will be more or less investigated in this report. In this MSc thesis, a GPU implementation of an algorithm very similar to that of Marden and Guivant (2012) is implemented and evaluated using a reference trajectory comparison, which enables comparison with other registration algorithms. Additionally, a few extension to the voxelized ICP algorithm is suggested to lessen the initial alignment requirement by searching on multiple resolutions and performing a pre-search in the voxelized space.

3.3 kD-tree acceleration on GPU

While voxelization might aid the speed of registration algorithms, it does perform a significant approximation of the underlying data. This motivates the use of search structures like kD-trees. Section 2.9 show how kD-trees are constructed and searched. These algorithms are usually not considered ”GPU-friendly” due to their recursive nature. Still, there is some work that evaluated the use of GPU for kD-trees and kD-tree like structures that will be investigated in this section. This is to accelerate the nearest-neighbour lookups which is the most computationally expensive step of the ICP algorithm (Besl and McKay (1992)).

3.3.1 GPU accelerated kD-tree search for registration

Qiu et al. (2009) did a GPU implementation of a kD-tree search algorithm. The main idea is that the kD-tree is constructed on the CPU and transferred to the GPU where the search itself is performed. The search algorithm is based on the algorithm proposed by Arya and Mount (1993).

The algorithm starts with construction of a kD-tree on the CPU in a similar manner as section 2.9.1. The kD-tree is left balanced and stored in a compact format. This is then transferred to the GPU where the search itself is performed. A flow chart of their system is shown in figure 3.2. The search algorithm works slightly different from the method described in section 2.9.2, where it instead of using recursion or a stack, they use a dynamic priority queue. The search starts by searching to the closest leaf node in the tree. On the way down the tree, it evaluates the distance to the hyper-rectangle of each node it visits. The priority queue is updated with the node, where the priority is the distance to the hyper-rectangle. When a leaf-node is reached, it will perform backtracking by extracting the currently closest element in the priority queue. To make this algorithm more GPU friendly, the priority queue has a fixed size. This makes the algorithm an approximated closest distance. However, they apply this kD-tree search strategy to perform the closest point lookup in an ICP algorithm, where they show that this guarantee is not required. This search algorithm is based on the search algorithm presented by Arya and Mount (1993).

Other than the correspondence search on the GPU, the algorithm is similar to a standard ICP algorithm where they use an SVD matrix approach for error minimization (as described in section 4.5.2 and section 3.1.5). This project has implemented the unit quaternion approach (as done by Besl and McKay (1992)) and the point to plane linear least squares approach (as done by Low (2004)), which differs from the work done by Qiu et al. (2009).

Qiu et al. (2009) claims that their algorithm is 88 times faster than a standard and comparable CPU implementation. They also use convergence as a way to determine the termination of the algorithms, and show that even a queue size of 1 will converge to the same minima as a queue size of up to 10, suggesting that the ICP algorithm is not dependent on having precise nearest neighbours (the ICP algorithm itself approximates the corresponding points, so this is more or less expected). The convergence is slightly slower with a queue size of 1, but since the performance overall is increased, this is not a big problem.

The authors suggest a more fully-fledged registration as future work, as they only focused on the registration step itself. They also suggested that kD-tree construction might be possible in the future. Another thing to note is that the dataset they present is slightly different and limited compared to the eye-in-hand camera configuration used in this project. As this paper is a few year old, the 3D scanner they use works quite differently from the scanners used with this project. The correctness comparison is also quite limited when comparing to alternative algorithms.

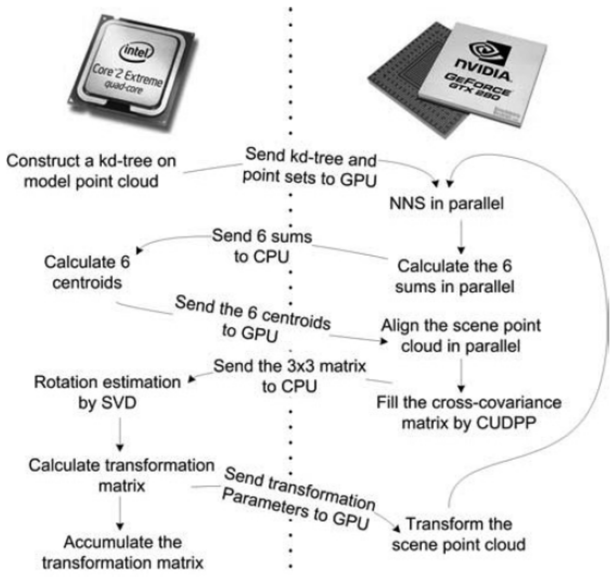

Construct a kd-tree on model point cloud
Send kd-tree and point sets to GPU
NNS in parallel
Send 6 sums to CPU
Calculate 6 centroids
Calculate the 6 sums in parallel
Send the 6 centroids to GPU
Align the scene point cloud in parallel
Send the 3x3 matrix to CPU
Rotation estimation by SVD
Fill the cross-covariance matrix by CUDPP
Calculate transformation matrix
Send transformation Parameters to GPU
Transform the scene point cloud
Accumulate the transformation matrix

Figure 3.2: Figure shows the data flow of the ICP algorithm implemented by Qiu et al. (2009). The figure shows where each steps of the ICP algorithm is computed. The figure is copied from their work (Qiu et al. (2009)). A similar architecture is used in this thesis, shown in figure 4.3.

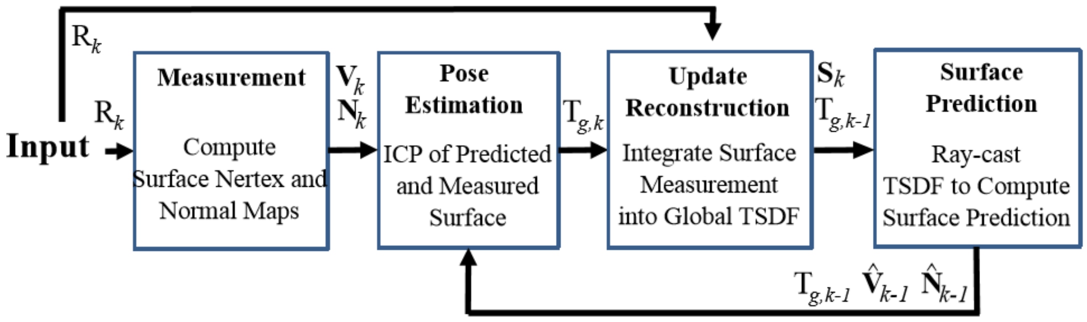
3.3.2 Other GPU based kD-tree implementations

Multiple other GPU based kD-tree search algorithms have been proposed. One such example is the work done by Hu et al. (2015) where they performed a kD-tree construction and search on GPU. The approach they use is based on the suggestions from Qiu et al. (2009), but they don't apply the algorithm for registration. The search method is similar to that of Qiu et al. (2009), which uses a priority queue based kD-tree search.

Zhou did used a surface area heuristic (SAH) kD-tree to perform ray-tracing (Zhou et al. (2008)). The main contribution is that the kD-tree search is performed without any stack. There also exist other similar work that use kD-tree searches to speed up certain tasks in graphics processing.

3.4 Kinect fusion

Another interesting work is the Kinect fusion algorithm. This is a real-time algorithm that enables 3D reconstruction of scenes, while a more conceptual description is given here. The implementation is based on the work by Newcombe et al. (2011). The architecture of the algorithm is shown in figure 3.3. Before briefly describing each of these steps, the Truncated Signed Distance Function (TSDF) will be described in section 3.4.1.



R_k
Input
R_k
Measurement
Compute Surface Nertex and Normal Maps
V_k
N_k
Pose Estimation
ICP of Predicted and Measured Surface
$T_{g,k}$
Update Reconstruction
Integrate Surface Measurement into Global TSDF
S_k
$T_{g,k-1}$
Surface Prediction
Ray-cast TSDF to Compute Surface Prediction
$T_{g,k-1}$ $\hat{V}_{k-1}$ $\hat{N}_{k-1}$

Figure 3.3: Figure showing the steps involved in the Kinect fusion algorithm. Figure is taken from the Kinect fusion paper by Newcombe et al. (2011).

3.4.1 Truncated Signed Distance Function (TSDF)

Multiple of the steps of the Kinect fusion algorithm is based around the Truncated Signed Distance Funciton (TSDF) structure. A Signed Distance Function (SDF) is a voxelized structure, where each voxel contains the signed distance to the nearest surface. Distances behind the surface is considered negative. In the Kinect fusion algorithm, the SDF measures the distance to the closest surface on a line going through the voxel and the camera. A TSDF is the same as an SDF, but the SDF is only computed in an interval $(-\mu, \mu)$, which is the expected measurement error of the sensor. This saves up a lot of space, as most of the space in a scene can be considered empty. This is shown in figure 3.4. This volume was the main inspiration for the pre-search strategy introduced in section 4.4.5

3.4.2 Measurement

This step is capturing depth images from a 3D scanner and performing some initial computations (Newcombe et al. (2011)). This involves image smoothing and normal computations which is computed from the pixel neighbours in the raw depth image.

3.4.3 Pose estimation

This is the most interesting step regarding this project. This step is the registration step, where an ICP algorithm is used to find the transformation of the camera. The ICP version used in Kinect fusion is using a projective data association algorithm which is based on work by Blais and D. Levine (1995) and uses a point-to-plane error metric as Chen and Medioni (1992); Newcombe et al. (2011). This requires a good initial alignment, which the authors argue is possible due to the high frame-rate (the camera is not able to move significantly between each frame). The Kinect fusion algorithm also applies a multi-resolution registration by downsampling the

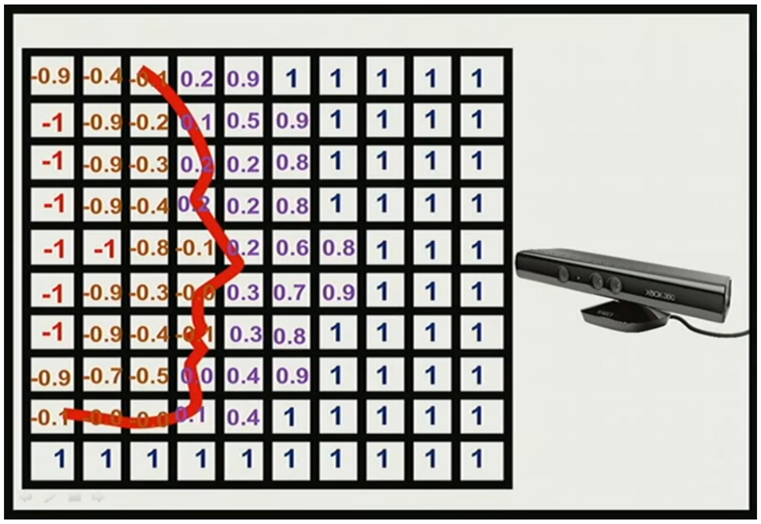


Figure 3.4: Figure showing a TSDF-volume in 2D. Each voxel represents the distance to the surface on a line going through the voxel and the camera. The sign is negative on the ”invisible” side of the surface. Note that the values are clamped between -1 and 1, which is considered empty space. This space can be simplified, saving a lot of space compared to an SDF. The figure is taken from the PCL website (perception foundation (2017)).

input image. The lower resolution image captures coarse camera movements, while the lower resolutions capture finer camera movements.

3.4.4 Update reconstruction

This step merges the registered image onto the existing TSDF, updating the estimated model (Newcombe et al. (2011)).

3.4.5 Surface prediction

The final step is the surface prediction step. This step is used to generate an estimated image that the next input image is aligned to. The ray-tracing is moving in steps of μ (where μ is the uncertainty of the camera as described in section 3.4.1) and is looking for sign changes in the TSDF, meaning that the surface is passed. If a sign change is found, a more precise search is performed in that area to find the voxel that is closest to the sign change.

3.4.6 Summary

The algorithm described is a relatively complicated and interwoven algorithm. Compared to the registration pipeline found in PCL, the Kinect fusion algorithm has less modularity, where each stage of the algorithm is heavily dependent on the other stages. The algorithm is also dependent of voxelization of the input space. As mentioned in section 1, this should probably be avoided if possible. The algorithm is not implemented or tested for this project, but the open source alternative Kinfu (section 3.5) could be a good alternative to evaluate for this project, as it could give real-time registration (Kinfu was also not implemented or evaluted in this project).

3.5 Kinfu

The Kinfu algorithm is an open-source alternative to the Kinect fusion algorithm (describe in section 3.4). The algorithm is based on the same architecture and concepts, though some differences are present as the Kinfu algorithm tries to use more of the existing GPU functionality in PCL. Because it is similar to the Kinect fusion algorithm, it has some of the same issues. It is more like a separate system in the PCL project, and is not based on the standard registration pipeline found in PCL. Therefore, it lack some of the modularity present in PCL, which in turn is one of the reasons to use PCL in the first place. A case study by Pirovano et al. (2013) shows how the Kinfu algorithm works, as well as an implementation of a 3D scanner (not using Kinfu) which will be explained later in section 3.6.

In Kinfu, a few algorithms have been replaced as it is later integrated into PCL, and want to utilize some of the built-in functions in PCL (Pirovano et al. (2013)). One difference is that the Kinfu algorithm uses eigenvalue normal estimation as described in section 2.5, instead of a vector cross-product between neighbouring pixels in the depth image as done in Kinect fusion. Rendering is performed using a marching cubes algorithm instead of the ray-casting method used in Kinect fusion. An additional difference is that the Kinfu algorithm works with any scanner supported in PCL, while Kinect fusion only works with the Microsoft Kinect (though this would probably be an easy extension for the Kinect fusion algorithm anyway).

3.6 A PCL based 3D scanner

Pirovano et al. (2013) also used PCL to implement a 3D scanner without using the Kinfu library. The registration framework developed for this project (section 4.3) is based on the method described by Pirovano et al. (2013). However, as Pirovano et al. (2013) doesn't use a GPU implementation for this, it depends on filtering to reduce the point cloud size. These filters are also implemented in the registration framework described in section 4.3 because they are generally usefull when doing object reconstruction and registration. These filters and processing stages will briefly be described here:

3.6.1 Voxel grid filter

This voxel grid filter works as described in section 2.4. This is the first step of the 3D scanner described by Pirovano et al. (2013). This filter can significantly reduce the size of the point cloud (which will improve performance).

3.6.2 Crop box filter

This is another simple filter that simply keeps the points that are inside the bounds of a box specified by the user. This is the second step of the scanner described by Pirovano et al. (2013).

3.6.3 Plane model segmentation

This filter removes the most significant plane in the point cloud based on the geometric model fitting method described in section 2.6.1. This filter improves performance by decreasing number of points. This could also help the registration if the author is using a transformation estimator that is sensitive to image boundaries, where the registration algorithm will prioritize to align boundaries of the point clouds instead of the actual objects.

3.6.4 Euclidean cluster

Pirovano et al. (2013) also included an euclidean cluster filter, which is described in section 2.6.2. After the other filters are applied, there are still some points left that are not part of the point cloud of interest. These can be removed by an euclidean filter, given that they are significantly smaller or larger than the object of interest.

3.6.5 Initial alignment

The scanner implemented by Pirovano et al. (2013) is based on using a few scans to align the model. This required to use some initial alignment algorithm before using the ICP algorithm as the inital alignment from the previous scan is insufficient. Pirovani used the SAmple Consensus Initial Alignment (SAC-IA) algorithm to do this with Fast Point Feature Histograms (FPFH) as feature descriptors for the SAC-IA algorithm. This step is in many cases not required in real-time registration as using the previous transform is good enough for the initial alignment. After the initial alignment step, the actual registration is performed with the ICP algorithm, which is the main focus for this MSc thesis.

3.7 Other alternatives to ICP

Though the ICP algorithm have become the most dominant approach for point cloud registration, there are many alternatives to this algorithm. Magnusson (2013) collected a few alternatives to registration in general which will be briefly summarized here.

One class of algorithms are based on the same concepts as the ICP algorithm, where the (Iterative Dual Correspondence) proposed by Lu and Milios (1994) is one such example. The core of the algorithm is the same, but it also includes another correspondence point which helps finding the correct rotation. This extra correspondence point is based on similarity in polar coordinates. Another group of algorithms are based on probability estimates of points r regions of the point cloud. The pIC algorithm proposed by Montesano et al. (2005) is similar to the ICP algorithm, but is based around uncertainty of initial pose estimate and scanner noise. Similarily, the point-based probabilistic registration algorithm by Hhnel2002ProbabilisticMF treats the point as probability distributions instead of individual points. The NDT algorithm is another work Biber and Strasser (2003) which is again based on a space representation that models the likelihood of finding a surface point as a linear combination of normal distributions. The advantage is that the problem can be solved using numerical optimization and the space representation avoid the expensive nearest neighbour lookup. Another class is based on numerical optimizations using Branch and Bound like algorithms for registration (like done by Nüchter et al. (2007)), which is slightly different approach to the problem, though they are better for 2D registration Magnusson (2013). A final suggestion is a more general class of algorithms that look at position invariant feature matching instead of the point based methods described so far. The main motivation behind such strategies is that initial poses become less relevant. Magnusson also mentions more NDT-like registration algorithms, but they are not mentioned here due to the similarity to the NDT algorithm, or lack of relevance (for example 2D registration).

While there are many alternatives to image and point cloud registration, the ICP is the dominant approach in the field. This is probably due to a relatively simple principle, yet very efficient. The modularity of the algorithm enables a quick setup of a domain specific variation of the algorithm, which works with high precision. The initial alignment requirement (though also present in many other algorithms) seems to be acceptable as many applications can use a different algorithm for initial alignment (As done by Pirovano et al. (2013)), assume the previous alignment as close enough (as done by Newcombe et al. (2011)), or maybe even a manual initial alignment. The ICP algorithm is a popular approach, and is implemented as the core registration algorithm in PCL (Open Perception Foundation (2017b)).

The robot scanner application presented in the project seems to favor the use of the ICP algorithm. The modularity of the algorithm enables exploration of multiple variations and properties. The initial pose can either be obtained from the robot itself.

CHAPTER 4

Implementations

This section is about the implementations that are done in the master thesis. It will give high-level descriptions of the algorithms and not deal with the details of the implementations. The implementations presented are a continuation of the work done in the pre-project by Isachsen (2017)

4.1 Robot control for camera movement

As a Franka robot arm was provided with the project, it was necessary to integrate the robot control software with the registration. The robot setup itself is explained further in section 5.1.1. While section 4.2 goes through the implementation for capturing point clouds, this section will go through the software controlling the robot itself.

According to the Franka examples and documentation, there are multiple ways to control the robot arm (Franka Emika GmbH (2018, 2017)). One such example is the ”Cartesian velocity controller”, which was used in this project. The ”cartesian velocity controller” guides the robot by specifying the wanted velocity of the end effector at regular intervals (1000 Hz). ”The Cartesian pose controller was initially attempted” (controlling the pose of the end effector instead of the velocity), but due to technical issues with the robot in this mode, the Cartesian velocity was the one that was selected. This worked well enough for simple movement for scanning objects. As the robot arm movement is not the main purpose of this project, linear accelerating movement was selected for the robot arm. This means that the robot arm moves the hand in a linear accelerating motion between user defined waypoints. The rotation of the robot is also accelerating, but not necessarily with linear angular acceleration.

The franka library requires samples at a rate of 1000 Hz. For the cartesian pose controller, it requires a new cartesian pose each millisecond (hand pos and orientation in x,y,z coordinates as well as orientation in pitch, roll and yaw). The cartesian velocity controller is similar, but it requires a cartesian velocity every millisecond (this is the velocity of the hand position in x, y, z as well as angular velocity in pitch, roll and yaw). As the cartesian velocity controller was selected, the following implementation is based on those principles.

Given a linear accelerating movement, the robot motion can be divided into two stages. The first stage is the acceleration stage, and the second is the deceleration stage. The acceleration is constant towards the destination point in the acceleration stage and constant away from the destination point in the deceleration stage. The acceleration stage is the first one, and switches to the deceleration stage halfway through the motion towards the destination point (both in distance and time). By using some classical mechanics, it is possible to calculate the expected timings. t_0^t is the starting time of the translation movement, t_1^t is the middle time of the translation movement and t_2^t is the end time of the translation movement.

$$t_1^t = \sqrt{2\frac{s_1 - s_0}{a_{max}}} + t_0^t \tag{4.1}$$

$$t_2^t = 2\sqrt{2\frac{s_1 - s_0}{a_{max}}} + t_0^t \tag{4.2}$$

s_1 is the distance from the starting point s_0 and the middle point on the path to the destination. a_{max} is the maximum allowed acceleration for translation movement. If s_2 is the destination point, the following equation can be used to calculate the distance.

$$s_0 = 0 \tag{4.3}$$

$$s_2 = \|P_2^t - P_0^t\| \tag{4.4}$$

$$s_1 = \frac{1}{2}(s2 - s0) \tag{4.5}$$

Where P_0^t is the translational component of the start pose, and P_0^t is the translational component of the destination pose.

Similar considerations can be done for rotation.

$$t_1^r = \sqrt{2\frac{\theta_1 - \theta_0}{\alpha_{max}}} + t_0^r \tag{4.6}$$

$$t_2^r = 2\sqrt{2\frac{\theta_1 - \theta_0}{\alpha_{max}}} + t_0^r \tag{4.7}$$

$$\theta_0 = 0 \tag{4.8}$$

$$\theta_2 = dq(P_0^r, P_2^r) \tag{4.9}$$

$$\theta_2 = \frac{1}{2}(\theta_2 - \theta_0) \tag{4.10}$$

t_1^r is the time of the middle point of the rotation from P_0^r to P_2^r. t_2^r is the time of the end point of the rotation from P_0^r to P_2^r. P_0^2 is the rotational component of the start pose as a quaternion and P_2^r is the rotational component of the destination pose as a quaternion. α_{max} is the maximum allowed rotational acceleration. θ_0 is the starting angle, θ_1 is the angle relative to θ_0 of the middle point of the rotation, and θ_2 is the angle relative to θ_0 of the destination rotation. Finally, the $dq(P_0^r, P_2^r)$ is the angular distance between quaternion P_0^r and P_2^r.

$$dq(a,b) = 2 \cdot arctan2(\| vec(a \cdot b*) \|, |w(a \cdot b*)|) \tag{4.11}$$

$$arctan2 = something \tag{4.12}$$

The vec function extracts the vector component of a quaternion (which is the i, j, and k components). The w function extracts the real part of the quaternion.

The timings obtained from rotation and translation is most likely different. The actual time of the movement is estimated as the maximum of the two.

$$t_2 = max(t_2^r, t_2^t) \tag{4.13}$$

$$t_1 = max(t_1^r, t_1^t) \tag{4.14}$$

The acceleration in translation (a) and rotation (α) is found using the following:

$$a = 2\frac{s1 - s0}{(t1 - t0)^2} \tag{4.15}$$

$$\alpha = 2\frac{\theta_1 - \theta 0}{(t1 - t0)^2} \tag{4.16}$$

The wanted velocity is computed from the current time t and distance from start point s.

$$\begin{cases} v = a(t - t_0) & \text{if} \quad s < s_1 \\ v = -a(t - t_2) & \text{otherwise} \end{cases} \tag{4.17}$$

The final translation velocity vector sent to the robot controller $\overrightarrow{v}$ is found by multiplying the velocity with the direction of movement $\overrightarrow{d}$.

$$\overrightarrow{v} = v \cdot \overrightarrow{d} \tag{4.18}$$

$$\overrightarrow{d} = \frac{P_2^t - P_0^t}{\|P_2^t - P_0^t\|} \tag{4.19}$$

The rotation is treated slightly differently as it needs to be converted from quaternions to euler angle velocity. As this is a quite involved conversion, the following definition is not accurate, but is good enough for the purpose of scanning objects. The idea is that it uses a circular interpolation (slerp) to find the previous expected rotation q_{prev} and current wanted rotation $q_{current}$. Then, it takes the difference between the two rotations and converts it to euler angles. By dividing by the time it takes between the current rotation and the previous, it is able to estimate the euler angle velocity of that step.

$$t_{prev} = t - \Delta_t \tag{4.20}$$

Δ_t is the time that have passed since the last robot pose update happened. t_{prev} is the time at the previous robot pose update and t is the current time.

$$\begin{cases} \beta_{current} = 2(\frac{t-t_0}{t_2-t_0})^2 & \text{if} \quad t < t_1 \\ \beta_{current} = 1-2(1-(\frac{t-t_0}{t_2-t_0})^2) & \text{otherwise} \end{cases} \tag{4.21}$$

$$\begin{cases} \beta_{prev} = 2(\frac{t_{prev}-t_0}{t_2-t_0})^2 & \text{if} \quad t_{prev} < t_1 \\ \beta_{prev} = 1-2(1-(\frac{t_{prev}-t_0}{t_2-t_0})^2) & \text{otherwise} \end{cases} \tag{4.22}$$

$\beta_{current}$ and β_{prev} are some temporary values that will be used for the circular interpolation. This is the interpolation amount.

$$q_{current} = \text{slerp}(P_0^r, P_2^r, \beta_{current}) \tag{4.23}$$

$$q_{prev} = \text{slerp}(P_0^r, P_2^r, \beta_{prev}) \tag{4.24}$$

Refer to section B.3 for how the slerp function is defined. The next step is to find the euler angle representation of the two current rotations.

$$\overrightarrow{r_{current}} = \text{quaternionToEuler}(q_{current}) \tag{4.25}$$

$$\overrightarrow{r_{prev}} = \text{quaternionToEuler}(q_{prev}) \tag{4.26}$$

Refer to section B.4 on how the quaternion is converted to euler angles. The final step is to find the corresponding euler-angle-velocity ω with the following.

$$\omega = \text{clamp}(\frac{\overrightarrow{r_{current}} - \overrightarrow{r_{prev}}}{\Delta_t}) \tag{4.27}$$

The clamp function simply converts the angle of each axis to the range $[-\pi, \pi]$. This is the translation and rotational velocities that is sent to the robot controller. This is done with a rate of 1000 Hz. For each update the robot does, the current pose is read. All poses are stored in a file with a corresponding timestamp. Note that the read value is not based on the wanted positions calculated, but is the measured position done by the robot.

Due to the bad quaternion to euler conversion described here, the rotation will drift significantly. This was fixed by measuring the drift from the wanted position. This position was then added to the destination position (it then wants to travel that distance extra). This gave a lot better movements, and gave a few degrees error for a typical movement. The drift-fix was also used for the position. At this point the movement was considered to be sufficient.

The waypoints can be read from text files. These files can either be generated by some program or be manually constructed. However, for this project, it was developed an additional program that enabled the user to move the robot to certain waypoints and obtain the pose of that waypoint. These can be read by the robot arm controller. The franka library enables this kind of interaction very simple because it enables the user to read the pose anytime. This is made possible becaues the Franka robot can easily be moved manually.

4.2 Camera capture methodology

While section 4.1 enables the robot to move in a specific pattern, this section will describe how the images are obtained while the robot is moving (the camera is not required to be mounted on a robot arm). The description shown here is based on the guide by Blacker (2016).

The output obtained from the realsense cameras through the librealsense library are standard image streams. There are three separate optional streams that can be configured to be obtained from the realsense cameras. These are the depth stream, color stream and infrared stream. This report will not consider the infrared stream, but the principles are the same as for the color stream.

The raw data from the camera are simple arrays of pixels. Each pixel in the depth stream is a 16-bit value representing depth, and each pixel in the color stream is a 3-byte value where each byte represents the red, green and blue channels. The width and height of the image are obtained from the intrinsics for each stream.

When the image is obtained, this must be converted to a point cloud. If only the depth stream is present, the point cloud can be generated with a simple deprojection. If the color stream is also present, the two streams need to be aligned. This is because the infrared camera used for depth estimation is offset from the rgb-camera with a translation and rotation. To align these streams, the depth image is first deprojected. Then it is transformed with the offset from the rgb-camera before it is projected back onto the rgb-camera. The transformation matrix representing the transform between the infrared and rgb-camera is called the extrinsics. This transformation and reprojection is done for each pixel in the depth stream.

The projection and deprojection is handled by librealsense, but the technique is summarized here. The method assumes the camera is positioned at $x = 0, y = 0, z = 0$ and pointing in the z-direction with the centerline of the camera going along the z-axis. The deprojected point is found with the following formula.

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} d \cdot u_x \\ d \cdot u_y \\ d \end{bmatrix} \tag{4.28}$$

d is the depth of the pixel. u_x and u_y are the x and y value of the pixel after considering the camera intrinsics and lens distortion. They are computed with the following assuming no lens distortion

$$\begin{bmatrix} u_x \\ u_x \end{bmatrix} = \begin{bmatrix} p_x \\ p_y \end{bmatrix} \tag{4.29}$$

however, for an inverse Brown Conrady distortion model, the following formula is used.

$$\begin{bmatrix} u_x \\ u_y \end{bmatrix} = \begin{bmatrix} p_x f + 2c_3 p_x p_y + c_4(p_x^2 + p_y^2 + 2p_x^2) \\ p_y f + 2c_4 p_x p_y + c_3(p_x^2 + p_y^2 + 2p_y^2) \end{bmatrix} \tag{4.30}$$

$$\begin{bmatrix} p_x \\ p_y \end{bmatrix} = \begin{bmatrix} \frac{i_x - ppx}{f_x} \\ \frac{i_y - ppy}{f_y} \end{bmatrix} \tag{4.31}$$

$$f = 1 + c_1(p_x^2 + p_y^2) + c_2(p_x^2 + p_y^2)^2 + c_5(p_x^2 + p_y^2)^3 \tag{4.32}$$

An image capturing tool was developed for the pre-project. This tool reads the raw images from the capturing device (realsense camera) and stores each of the streams in separate files. The color images are stored as bmp images, while the depth-stream is stored as dpt files which is a custom file format. The format contains the required intrinsics and extrinsics as well as the depth stream itself stored in a raw format. A corresponding tool was developed to read the files generated by the capturing tool. This reader tool generates the point clouds as described in this section, but is incorporated in the registration framework described in section 4.3. The reason for using this kind of architecture is first of all to generate datasets that can be read without using the camera. The reason for saving the raw depth file instead of a point cloud (which can be done through pcl) is to obtain flexibility for algorithms working directly on the raw image input. No algorithm implemented in this project utilizes the raw depth image, but in the pre-project, this was not certain. The registration framework also has a capturing tool to do the registration in real time.

4.3 Registration framework

This is the program that runs the actual registrations and take the timings of them. The program is configurable by inserting program arguments from the command line. It is possible to configure the registration program to select registration parameters, filtering and writing of results. The input to the program is the files to perform registrations on, as well as the arguments which describes how this is done. The program will output log messages to the command line or to a log file if specified. It can also output the final registration as a .pcd file if specified. Figure 4.1 shows the registration pipeline including all the steps.

The input files can either be the .dpt files (as described in section 4.2) or .pcd files. In the case of .dpt files, the program will look for a corresponding .bmp images based on the name of the .dpt file and .bmp files.

The other program arguments can specify most of the behaviour of the program. See appendix C for a list of all available arguments and their corresponding default value. It is possible to specify filters, filter arguments, registration algorithms, registration parameters and input/output (IO) based arguments. The following filters are supported: voxel grid, plane model segmentation, euclidean clustering, pass through filter and box filter. All filters have specialized arguments to configure the filter parameters. Though plane model segmentation and euclidean clusters are not strictly filters in terms of PCL terminology, they are referred to as filters in this sense because they remove certain parts of the point cloud based on some rules. Supported registration arguments are which registration algorithm to run and convergence criteria. IO arguments specify log files, output .pcd files, whether the point clouds should use color, and if the program should visualize the results.

When running the registration program, filtering is the first operation that is performed. The order of the filters is the same as the order they are specified in the arguments list. The voxel grid filter works as specified in section 2.4 by removing excess points that share the same voxel

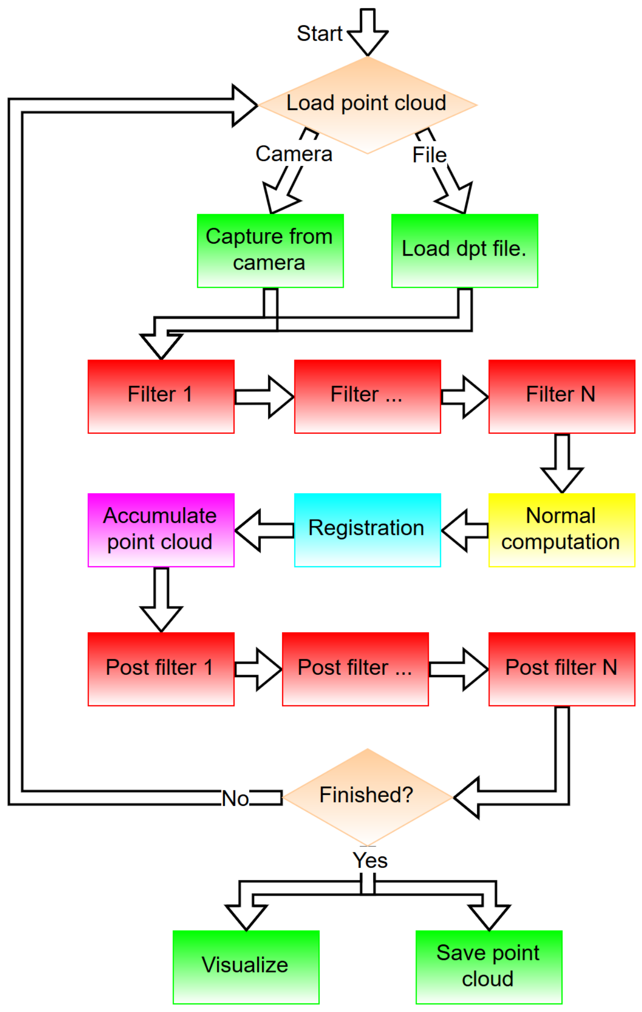


Figure 4.1: Figure showing the registration architecture. The point cloud is processed in all of the steps denoted in the figure. The figure starts by loading a point cloud, which can either be done from a saved file (.dpt or .pcd file formats) or from a RealSense camera connected to the computer. The following filtering steps are filters as explained in section 4.3. They are applied in the user specified order (see appendix C). The normal computation step is performed if it is specified by the user, or if the registration algorithm requires it. The registration step is the core of this project where the registration itself is performed. The captured and filtered point cloud is aligned to the accumulated point cloud. After this is done, the cloud itself is accumulated to the point cloud. The final step is the post-filtering step, that works like the filtering steps, only that they operate on the accumulated point cloud after the registration is performed. There is also the option to store the point cloud and visualize it.

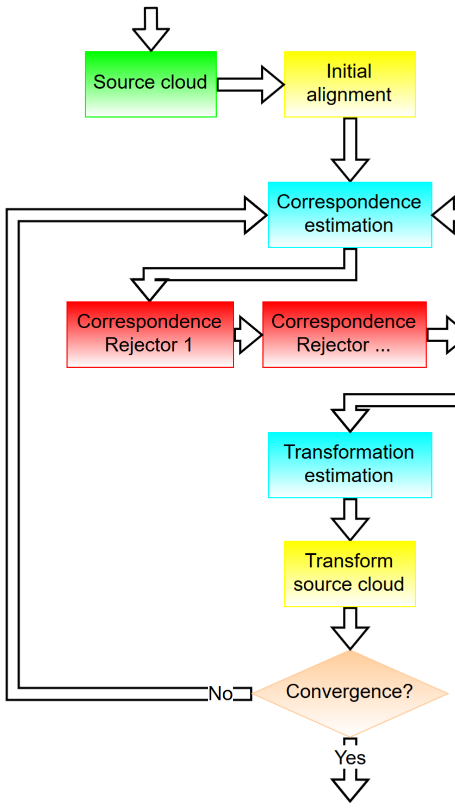

Source cloud
Initial alignment
Target cloud
Correspondence estimation
Build search structure
Correspondence Rejector 1
Correspondence Rejector ...
Correspondence Rejector N
Transformation estimation
Transform source cloud
Convergence?
No
Yes

Figure 4.2: Figure showing the steps involved in registration. These are basically the same steps that are outlined in sections 2.10.1 and 3.1. The registration is performed by finding a transformation that aligns the source cloud to the target cloud. To accelerate search, a search structure (like voxelized space of kD-tree) is constructed to speed up the registration. The source cloud is aligned with an initial transformation if provided. The correspondence estimation is performed as described in sections 2.10.1 and 3.1.2. The correspondence rejection is a list of correspondence rejectors that are applied in order (see section 3.1.4). The transformation estimation is done by a transformation estimation, that minimizes some error based on the correspondences found (section 3.1.5). Finally, the source cloud is transformed based on the transformation found in the previous step. If convergence is found (either by going below a fitness threshold, transformation threshold or reaching the iteration count), the algorithm terminates. Otherwise, the registration step is repeated.

when dividing the space into a 3D grid. Plane model segmentation works by removing all points belonging to the most dominant plane in the model using the method described in section 2.6.1. This is used to remove the table surface from the point clouds. The euclidean clustering works the same way as described in section 2.6.2, and will simply remove points that don't belong to a large enough cluster (based on a threshold which can be specified as an argument). This filter removes noisy irrelevant point data that is usually not part of the object of interest. The pass through filter is simply a filter that removes points that are either too close or too far away from the camera. These points usually contain noise which makes the registration harder. The box filter will filter away all points that are outside the bounds of the box specified in the filter.

If required, the next step is to calculate normals. The program determines if the registration algorithm will require that normals are computed and compute them if necessary. The normals are computed using the build-in functions of PCL, which is based on the method described in 2.5.

The following step is the actual registration. The registration steps are shown in figure 4.2. The program will select the registration algorithm, as well as the transformation method and correspondence estimation method, all specified by the user (otherwise, it is defaulted). Some of the algorithms, correspondence estimators and transformation estimators are already implemented in PCL. Some of those will be included in the project as a baseline for registration correctness/accuracy and performance. Some other algorithms are implemented for this project, and is further described in section 4.4. Some of the algorithms that can be selected have some limitations in transformation method and correspondence estimation. This is because not all of the transformation estimators are implemented for the GPU based registration algorithms, and because the correspondence estimation is more implicit to the name of the algorithm itself (which is what is tested). The reason for the implicit correspondence estimation is that the registration algorithm might require control of GPU data during the iterations, and might differ slightly from the standard registration pipeline described in PCL (Open Perception Foundation (2018a)).

Finally, after the registration is performed, the program adds the newly registered cloud to an accumulated cloud.

The post-filter stage is similar to the filtering stage. The same filters and arguments apply, but this filtering is performed on the accumulated cloud. This step might be required in a real-time scanner (depending on how long it is expected to run) in order to limit the accumulated point cloud size. If the point cloud grows to large, the running time of each iteration is increased. A post filter might remove excess points such that the accumulated point cloud is limited in size. Another reason to use a post-filter is for other filtering options that are not convenient to perform before the registration step.

The accumulated can then be visualized immediately. This means that the point-cloud can be visualized after each frame is registered to show the real-time behaviour as the scanner moved around the object. The visualized object is the accumulated cloud after the post-filtering stage.

The steps described here are run for each point cloud in the stream. The first point cloud is only filtered, as there is no current model cloud to compare it to. After all registrations are performed, the resulting cloud can be stored to disk, or be visualized within the registration program.

The registration program uses logging during the execution such that the user can obtain information about what it does. The log contains timings and registration results. Registration results are for example the fitness score of the ICP algorithm and the final transformation matrix. This log is designed such that it is simple to parse for a different program which can provide insights (section 4.6.5). The timings are done using wall time. The timings are given after every operation described in this section. This fine timing is nice to have when evaluating an improved algorithm. While one algorithm might improve the registration time, it might require more computationally intensive pre-processing. The fine grained timing gives finer control on this matter.

4.4 Registration algorithms

This section is about the actual algorithms implemented for the master thesis. This is the main work of this master thesis. Most of the work is based on changing the point-correspondence method as this usually is the bottleneck of the registration process. Chapter 4.6 will go through other activities done on the GPU, while this chapter will mainly focus on the correspondence estimation step of the registration algorithms.

4.4.1 Brute-force correspondence estimation CPU

This is the first and most simple correspondence estimation implemented. It is implemented as a PCL correspondence estimator which enables it to be used with other registration algorithms already implemented in PCL. The algorithm uses a brute-force all-to-all closest match without any search structure. This gives the algorithm a running time of $O(NM)$, where N is the number of points in the source cloud and M is the number of points in the target cloud. This algorithm is merely included as a baseline to compare the other implementations described in later chapters.

4.4.2 Brute-force correspondence estimation GPU

This is the same correspondence estimation as in section 4.4.1, but implemented on the GPU. The asymptotic performance is still expected to be $O(NM)$ where N is the number of points in the source cloud and M is the number of points in the target cloud. The performance is still expected to be drastically improved compared to the CPU implementation because of the massive parallelity of the GPU. The algorithm is not expected to suffer from branch divergence, or similar.

4.4.3 Kd-tree correspondence estimation CPU

A kd-tree is almost guaranteed faster than a brute-force correspondence estimation for most point clouds. This is why this is defaulted to be the search structure when using registration algorithms in PCL (Open Perception Foundation (2017b)). This project contains a CPU implementation of a kD-tree for nearest neighbour search in order to have a baseline to compare with the existing implementation is PCL. The implemented kD-tree is based on the description found in section 2.9. This section will further describe some of the implementation details of kD-tree construction and search.

Before the individual iterations in the ICP algorithm is run, the kD-tree is constructed. The kD-tree is constructed once for the target cloud, and then searched for each iteration of the ICP-algorithm. The tree uses a dense tree representation as described in section 2.9.3. In order to balance the tree, a quick-partition algorithm is used to find the median (this works the same way as quicksort, but does only one recursive call, giving it an expected running time of $O(N)$ when finding the median). The quick-partition algorithm guarantees that all elements in the list before the median is smaller than the median, and that all other elements are greater or equal. This must be done for each subtree with it's corresponding dimension. This will guarantee that the tree is balanced. This leads to a total running time of $N \log N$ asymptotic running time for kD-tree construction where N is the number of points in the target cloud.

The search is performed, and each element has expected running time of $\log N$, giving a total running time of $M \log N$ for a correspondence search. M is the number of points in the source cloud and N is the number of points in the target cloud.

4.4.4 kD-tree correspondence estimation on GPU

The kD-tree search algorithm is doing the same thing as the CPU version described in section 4.4.3. The only difference is that the search is performed on the GPU, while the kD-tree construction still is performed on CPU. Qiu et al. (2009) shows that most of the time consumption in the ICP algorithm is the point correspondence search. Another consideration is that construction of a kD-tree is the hardest part to perform on a GPU. All these factors suggests the method implemented here and Qiu et al. (2009). The idea is to construct the kD-tree on the CPU, and then transfer it to the GPU to perform the actual search there. While algorithm is similar to the one described in section 4.4.3, the actual search is performed on the GPU.

As in the work by Qiu et al. (2009) the main consideration when performing the search on GPU compared to a CPU implementation is the dynamic structures. In the case of a search, the dynamic structure is the stack to enable backtracking when searching through the kD-tree. Qiu et al. (2009) shows that the stack can have a fixed size which can be preallocated. If this size is shorter than the maximum depth of the kD-tree, the final result might differ slightly from the correct. However, as the ICP algorithm depends on correspondences for each point, the approximate nearest neighbour found using the limited stack search method can still give good registration results. Even if the search doesn't find the nearest neighbour, the found correspondence is likely to be relatively close to it.

4.4.5 Voxelized search space for correspondence estimation on CPU

The algorithm described in section 3.2 (based on the work by Marden and Guivant (2012)) describes the method implemented here. The space is divided into voxels, and the centroid is computed for each voxel. The search is done by finding which voxel each source points fall within, and use the centroid of that voxel (based on the target cloud) as the corresponding target point. The implemented algorithm is as described in section 3.2.

Two suggested improvements where suggested in this thesis. One improvement is to use a dynamic voxel size. The basis of this improvement is that using larger voxel sizes, the registration loses some accuracy due to loss of smaller features. Having to small voxels might require too many iterations to complete because fewer correspondences are found when the alignment is bad (typically at the earliest iteration of the algorithm). It might even be unable to find any correspondences at all if the initial alignment is not good enough. The idea is to use a larger voxel size for the earliest iterations to get the higher level features and then use a smaller voxel size for the later iterations in order to align using more low-level features.

The other suggested approach is to use an initial distance estimation on the available voxels. The first step is to construct the voxelized centroid structure as described in section 3. This voxelized space is called V_0. Then, a new voxel space V_1 is created by copying V_0. Then, each empty voxel in V_1, the the closest centroid is found by searching through V_0 with a maximum distance d. The voxel centroid in V_1 is simply copied from the closest nonempty voxel in V_0. This can enable the algorithm use slightly smaller voxel sizes, while still being able to search longer distances. This method is easily parallelizeable on the GPU by assigning a thread to each voxel in V_1.

4.4.6 Voxelized search space for correspondence estimation on GPU

This algorithm is based on the same concepts that are described in section 4.4.5, but is computed on the GPU. Both voxel space construction and voxel space search is implemented on the GPU. The only thing to note here is that there is a race condition when parallelizing this algorithm. When iterating over the target point cloud, multiple points can fall into the same voxel, and there is therefore a race condition. This race condition is not taken into consideration. Both voxelization extensions suggested in section 4.4.5 are also implemented on the GPU.

4.5 Point correspondence rejector

PCL comes with a lot of different point correspondence rejectors (Open Perception Foundation (2017b)). The correspondence rejector implemented for this project is based on the organized boundary correspondence rejector. Rusinkiewicz and Levoy (2001) suggest that the correspondences from the boundaries of the point cloud should be rejected because this is often where point clouds don't have overlapping features. However, the organized boundary rejector requires an organized point cloud (as described in section 2.3.1). This implementation is based on the same concept, but does not require the point cloud to be organized. The concept is that the user specifies a region of interest as a box. This box is then transformed together with the cloud through the iterations of the ICP algorithm. The box is specified in camera-frame, which enables the user to specify the region of importance in the point cloud. After the box-structure is configured, the rejection consists of two steps. The first step is to determine the transformation of the box, and the second step is to determine which points to reject.

4.5.1 Rejection box construction

The box consists of six planes specified by their normals and a point on the surface. The user specifies the bounds of the box using the minmum and maximum values of the box in each coordinate axis. The box is always aligned with the coordinate axes in the camera frame. The planes are specified from their corresponding normal vector and surface point (meaning that plane 1 is specified by $\vec{n}_1$ and p_1, plane 2 is specified by $\vec{n}_2$ and p_2, etc.)

$$\vec{n}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \vec{n}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad \vec{n}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \quad \vec{n}_4 = \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix}, \quad \vec{n}_5 = \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix}, \quad \vec{n}_6 = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix} \tag{4.33}$$

And similarily, the corresponding points are defined as the following.

$$p_1 = \begin{bmatrix} x_{min} \\ y_{min} \\ z_{min} \end{bmatrix}, \quad p_2 = p_1, \quad p_3 = p_1, \quad p_4 = \begin{bmatrix} x_{max} \\ y_{max} \\ z_{max} \end{bmatrix}, \quad p_5 = p_4, \quad p_6 = p_4 \tag{4.34}$$

4.5.2 Finding the transform

This is a simple task given the transformation from the registration algorithm. However, in PCL, the correspondence rejectors are not given the transformation from the registration. Therefore, the transformation is computed by observing the movement of the first three points in the source point cloud and computing the transformation from this. This is possible because the points don't change positions during registration, though the method described here is not very pretty, and would optimally be avoided if possible. The method uses SVD on the points to obtain the optimal rotation.

The first step is to compute the centroids of the original point position μ^{orig} and the new transformed point μ^{trans}.

$$\mu^{\text{orig}} = \frac{1}{3}(p_1^{\text{orig}} + p_2^{\text{orig}} + p_3^{\text{orig}}) \tag{4.35}$$

$$\mu\text{trans} = \frac{1}{3}(p_1^{\text{trans}} + p_2^{\text{trans}} + p_3^{\text{trans}}) \tag{4.36}$$

The matrix H is computed as the cross-covariance between the points after subtracting the centroid

$$X = \begin{bmatrix} p_1^{\text{orig}} - \mu^{\text{orig}} & p_2^{\text{orig}} - \mu^{\text{orig}} & p_3^{\text{orig}} - \mu^{\text{orig}} \end{bmatrix} \tag{4.37}$$

$$Y = \begin{bmatrix} p_1^{\text{trans}} - \mu^{\text{trans}} & p_2^{\text{trans}} - \mu^{\text{trans}} & p_3^{\text{trans}} - \mu^{\text{trans}} \end{bmatrix} \tag{4.38}$$

$$H = XY^T \tag{4.39}$$

X and Y are simply 3×3 matrices where each column is a point where the centroid is subtracted. The next step is to perform an SVD matrix decomposition.

$$U\Sigma V = H \tag{4.40}$$

The rotation R can then be found from this:

$$R = VDU^T \tag{4.41}$$

and the translation $\vec{t}$ is found using

$$\vec{t} = \mu^{\text{trans}} - (R\mu^{\text{orig}}) \tag{4.42}$$

The determinant of R is wanted to be positive, which is why the matrix D is introduced. The D matrix is an identity matrix where the lower right element is replaced by the determinant of VU^T.

The rotation matrix R and translation $\vec{t}$ is now applied to the box coordinates in the following way.

$$\vec{n}_i{}^{\text{trans}} = R\vec{n}_i | \text{for} i = 1..6 \tag{4.43}$$

$$p_i = (Rp_i) + \vec{t} | \text{for} i = 1..6 \tag{4.44}$$

This is basically the same steps required in the ICP algorithm, but multiple iterations are not required. This is because the corresponding points are certain to be correct.

4.5.3 Doing the correspondence rejection

Now, the rejection box is at it's correct place, and the actual rejection can take place. A point inside the box is indicated that the signed distance to all planes are positive. If the point is outside the box, the point is rejected. The following equation determines if the point is inside the box or not:

$$\vec{n}_i \cdot (p_0 - p_i) > 0 \quad | \quad \text{for} i = 1..6 \tag{4.45}$$

p_0 is the source point of a correspondence pair. If this condition is true, the point is on the inside of the box, otherwise it is outside, and the correspondence is rejected. This step is repeated for all point correspondences.

When implemented on the GPU, the point correspondence rejector can easily be extended to work on the GPU. The difference comes from the implementation details, where the GPU

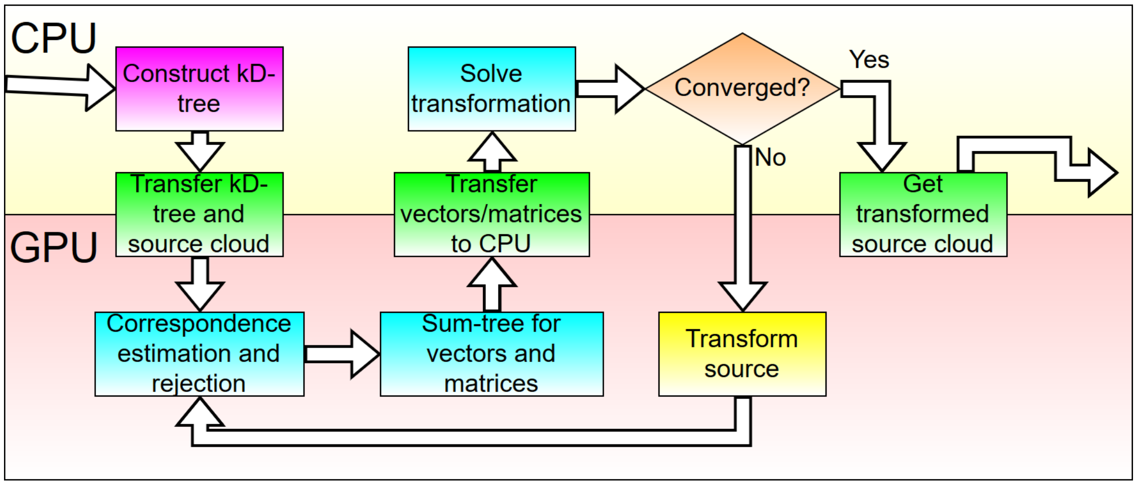

CPU
GPU
Construct kD-tree
Transfer kD-tree and source cloud
Correspondence estimation and rejection
Sum-tree for vectors and matrices
Transfer vectors/matrices to CPU
Solve transformation
Converged?
Yes
No
Get transformed source cloud
Transform source

Figure 4.3: Figure showing the work division and interaction between the CPU and GPU for the ICP algorithms implemented for this project. The developed 3D registration architecture for the GPU implementations used in this project is based on the architecture developed by Qiu et al. (2009). The figure shows the same steps as in figure 4.2, but also how the work is divided between the GPU and CPU. Looking at the step in the figure, the CPU is basically only solving the transformation for each iteration of the ICP algorithm. The data sent back and fourth between the GPU and CPU in the main loop is basically a few matrices and vectors (depending on the selected transformation estimator).f

implementation doesn't change the length of the correspondence list, and would instead set a flag, indicating that the correspondence is invalid.

4.6 Other implementation considerations on GPU.

This section involves other implementation details when doing registration on the GPU. Section 4.6.1 shows how the work is divided between CPU and GPU for the ICP algorithm. Section 4.6.2 is about point cloud transformation on the GPU and section 4.6.3 is about how the reductions in the transformation estimation is performed on the GPU.

4.6.1 CPU and GPU interaction

This section will show how the work is divided between the CPU and the GPU. This architecture is based on the one used by Qiu et al. (2009). However, the same architecture/pipeline is valid for the other GPU based registration algorithms. Figure 4.3 shows the interaction and division of work between the CPU and GPU.

4.6.2 Point cloud transformation

Transformations of point clouds is a trivial task for the GPU and is in itself a massively parallel task. Transformations of the point cloud is therefore performed on the GPU. This is in itself utilizing the parallelity of the transformation operation as well as it makes avoids transferring the point cloud back and forth from the GPU between each iteration. The implementation is basically applying a transformation matrix to each point.

4.6.3 Reduction trees

Many of the transformation estimations described in section 3.1 are based on summation of matrices for each point. This can also be performed on the GPU and as Harris (2018) shows, there are many ways to do this. The method used in this project is something that looks like ”Kernel 4” in the reduction presentation by Harris (2018). While the presentation show other faster alternatives, the speedup is probably less significant when operating with matrices and vectors as each level in the reduction tree does more work compared to integer summation. These methods could be considered, but where not used in this project mainly to have a simpler implementation with cleaner code, while still being fast enough and drastically faster than a similar CPU implementation. As claimed by Qiu et al. (2009), the transformation estimation is generally not the bottleneck in the ICP algorithm.

Each thread from the matrix computation step outputs it's own partial sum. Given n blocks and m threads per block, this gives an output size nm which is fed into the reduction tree. The summing tree will then first call the reduction kernel with n blocks and m threads per block. This will reduce the array size to n as a local reduction is performed for each block. The next step, m threads are still run per block, but the number of blocks have been reduced to $\frac{n}{m}$. This procedure is repeated until there is one element left in the array. By selecting $n \leq m$, this procedure can be completed with two calls to the reduction kernel, though other values are still possible. The final step is to transfer the first element of the array (containing the sum of all elements in the tree) back to the CPU where it can compute the final transformation.

The kernel itself works by doing a local reduction of elements within the same block. Each thread is assigned to an array element, and by using the sequential addressing method described by Harris (2018) it computes the sum of it's own element and another element based on the sequential addressing mode. Finally, the first thread computes the answer corresponding to it's block while the other threads in that block have completed their local sum.

Both the point-to-plane lls transformation and dual quaternion transformation estimators use a summing tree in the implementations. These are the two algorithms that have required a reduction tree on the GPU.

4.6.4 Convergence/termination criteria

The convergence criteria implemented in PCL are based on three conditions Open Perception Foundation (2017b). The first condition is the maximum iteration count. If the iteration count reaches this value, it is terminated. The second convergence criteria is the fitness epsilon. If the fitness of the registration (as described in section 5.1.3) changes less than a certain threshold (the fitness threshold), the algorithm will terminate. The final criteria is the transformation epsilon. Similarily as the fitness epsilon, if the transformation (for example translation) between two consecutive registrations drops below a certain threshold, the algorithm is said to converge. All of these convergence criteria are tested in section 5.1.4.

4.6.5 Evaluation tool

The evaluation tool is a python framework developed for this project. This program reads log-files from the registration program and robot controller program and formats the data in a more readable way such as visualizations and tables. The following list shows the most important features of this program.

- Show 3D pose visualizations from lists of pose-matrices.
- Generate timing tables.
- Compute evaluation metrics (as in section 5.1.3) from two timed lists of poses (for example robot pose and estimated pose from registration.)

Generally, all graphs shown in section 6 are generated using this tool.

CHAPTER 5

Evaluation

While chapter 4 explained the implementation done for this project, this section will go through the strategy for evaluation of the algorithms in section 5.1 and the results in section 6.

5.1 Experimental setup

This section will introduce the generated test sets, evaluation metrics and experimental setup.

5.1.1 Robot and camera setup

For this project, both free-hand scans and camera mounted on robot arm was performed. The robot arm provided for the project was the Franka robot arm (Franka (2017)). An Intel RealSense SR300 camera was mounted on the robot arm. Figure 5.1 shows how this camera is mounted to the robot arm. An Intel RealSense D435 camera was used for the free-hand scans.

5.1.2 Test sets

Several test sets were generated for the project. The test sets can be divided into two main categories depending on the scan method. The first category is the free-hand scans, and the other category is taken from a camera mounted on the robot arm as shown in figure 5.1. The following lists will contain the names of the test-sets as well as a few properties and considerations for the test set. The coordinate system used when describing the test sets are also shown in figure 5.1.

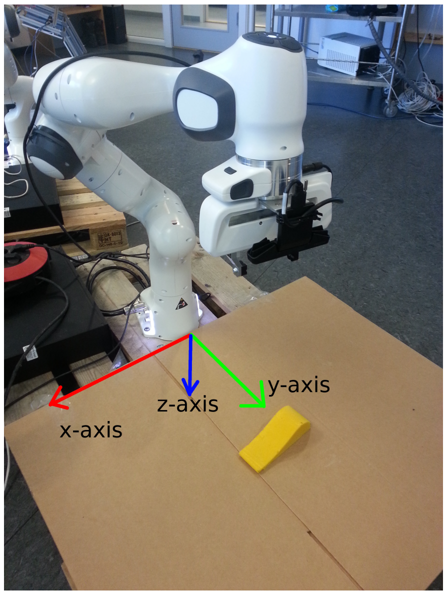


Figure 5.1: Figure showing the Franka robot arm and an Intel RealSense SR300 camera mounted on it. This was the setup used for robot arm based scans in this project. The figure also shows the coordinate system used when explaining the different test sets in this section.

Robot trajectory scans

The following list contains multiple test sets where the scan trajectory is the triangular scan as described in section 5.1.2 using the robot arm. The SR300 camera is used for these scans because it is the one that is mounted on the robot arm. The are taken from the robot-mounted Intel RealSense SR300 camera. Each of the test set is named after the object it scans. All object scans consist of 138 images with a framerate of 10 FPS (most of the tests does not use all images). The point clouds usually contain 300 000 points when read from the camera.

Monkey This is a stuffed toy, consisting of both large and smaller features as well as a mostly smooth surface. The fabric it is made of enables good 3D-synthesis for the SR300 camera. The object is close to mirror-symmetric along the y-z-plane. Some soft-body deformations are present, making it slightly asymmetric. Symmetry is not considered an issue for this object. The main challenge of this object is to align the smaller features correctly that can disappear when using downsampling.

Banana The surface of the objects enables good 3D-synthesis. The object is close to mirror-symmetric along the x-y plane, but because the plane is present, no symmetry is present. The main challenge of this set is the small size of the object compared to the field of view of the camera, but in general, this is a quite simple object to scan.

Pear The surface of the object enables good 3D-synthesis. The object is rotational symmetric along the y-axis. While this is not the primary axis of rotation for the robot movement, symmetry is expected to cause issues with the registration. Other than the small size of the object, this is a quite simple scan.

Apple The surface is slightly more glossy, but still gives quite good 3D-synthesis. The object is rotational symmetrical along the z-axis. Since a lot of movement is rotational relative to the z-axis, symmetry is expected to be an issue. Since the object has a very smooth surface, the slight asymmetry of the object is not expected to help much in the registration. The main challenge is the rotational symmetry making this a hard scan to register.

Cap The cap object is made of a fabric enabling good 3D-synthesis. The object is large compared to the other objects, has asymmetric features and has large and smooth surfaces. It is mirror-symmetric along the y-z plane. Taking all this into consideration, this object is not expected to give any problems.

Hat The object is made of a fabric enabling good 3D-synthesis. The object is relatively large compared to the other objects with relatively large features and a smooth surface. This object contains no symmetry. This object is not expected to give difficulties to register. Compared to the other objects, this is quite tall compared to the plane it is placed on, but it's not expected to be a challenge.

Fruit-scene This scene consists of an apple, a pear and a banana. The goal is to get a better registration of the pear and apple removing the rotational symmetry. The scene itself is partially symmetric in the x-y plane, and is not expected to give any issues. Given a single object, it can be hard to determine the correctness of the registration, so this test set can verify that the registration also is valid for a large area with muliple separated objects.

Box The object is made of cardboard making good 3D-synthesis. This object is mirror symmetric along multiple axes. The object has 4 symmetrical rotations around the z-axis as well as subsections that are translational symmetric (for example small segments of the surfaces). Using small frame-to-frame movements, this should not be an issue as the top surface is always visible. The object has clear surfaces and has rough edges. This object is not expected to give issues when registering. The main challenge is the symmetry when doing large steps between individual images.

Some plastic stuff This scene has a surface that is difficult for the 3D-reconstruction using the SR300 camera. The camera is only able to register a small portion of the object at different angles (depends on the viewing angle), and will produce more erroneous point than with previous objects (this is a problem with this kind of objects when scanned with the SR300 camera). The symmetry is similar to that of the pear.

Some casted object This is a different material than the previous objects. The yellow color (as seen in figure 5.1) makes it easy to see errors in the registration. This object is not considered a particular challenging scan.

Free-hand scans and food objects

The next test-set consists of food-objects captured with a hand-scanner. The scans has a lot of supporting structures in the environment to aid the registration process. This is because the features of the food objects themselves are quite small, and adding larger features helps the registration. These structures can easily be removed with some filtering. The tabletop is blue, and each object can be extracted based on color and position if this is wanted. The Intel RealSense D435 camera was used because of the wavy pattern generated by the Intel RealSense SR300 camera (see figure 6.4). These tests use different frame rates and durations, resulting in different number of images for each test set. Other reference objects are placed in the scene to aid the registration. While some registration algorithms are able to perform the registration anyway, it can enable more algorithms to register the objects.

Lettuce Free-hand scan with 19 RGB-D images using the D435 camera. The lettuce is placed on a table together with a box as a reference object. The surface of the lettuce gives good 3D reconstruction with with both cameras. The scene itself does not show any symmetry because of the reference objects.

Pork-filet Free-hand scan with 29 images using the D435 camera. This scene consists of a pork-filet with multiple reference objects. When using the SR300 camera, the meat gives a wavy effect in the depth stream shown in figure 6.4, which is why the D435 camera was used.

Pork-chop Free-hand scan with 13 images using the D435 camera. Similar to the previous scene, but the object itself is very flat, making it hard to separate from the table. Therefore, other reference objects were added around it to aid the registration. As in the previous scan, the meat generates a wavy-pattern (shown in figure 6.4) when using the SR300 camera, so the D435 camera was used for this object.

Salmon-top Free-hand scan with 21 images using the D435 camera. The object is a salmon filet with the skin side facing downwards. There are multiple reference objects in the scene to aid the registration because the object itself is flat. As with the previous scan, the wavy pattern shown in figure 6.4 was observed for the salmon as well when using the SR300 camera, which is why the D435 camera was used.

Robot scan trajectory test sets

In order to test some properties, some extra test sets where generated. The following test sets will focus on different scan trajectories. The goal of the trajectories is to show guidelines on scan trajectory as well as exploring some weaknesses of the ICP algorithm. Only a few of the objects mentioned above are used as the interesting part here is how well different trajectories work, and not properties of the objects themselves. The scanned object is a banana. See figure 6.5 for a visualization of the trajectories mentioned here. All trajectories run the camera at 10 FPS, and the generated point cloud contains approximately 300 000 points. The following list contains all tested scan trajectories. Note that all scan trajectories except the transnational scan has the object centered in the camera view.

Square scan A scan that looks at the object from object from four different sides. This gives a good view of the entire object, but is the slowest scan tested in this project. This results in about 162 images in total.

Long swipe scan A trajectory consisting of 3 waypoints looking down at the object. The trajectory goes over the top of the object. This scan has a total of 65 images. This is included because certain objects don't need to get a full view of all sides of the object.

Short swipe scan A short trajectory looking down at the object. As this is a short scan, it ends up with a total of 51 images. This is included because certain objects don't need to get a full view of all sides of the object.

Translation scan While most scans are rotational, this one is translational. This means that the movement relative to the object is mostly translational and not rotational. As shown in section 6.9 this introduces a few difficulties for the registration.

Triangle scan Trying to get as much as the objects as possible through 5 waypoints where 3 of the waypoints are positioned in a triangluar pattern such that the camera gets a view of all sides of the objects. This results in a total of 139 images.

Zoom movement scan This scan goes through a series of waypoints where the distance to the object is changed (meaning that it's actually not doing a zoom). No other movement is present. This trajectory is added to the project to evaluate if the registration algorithms are able to register object where the resolution of the object change.

Zoom scan This is similar to the zoom scan (meaning that this is not really a zoom either) in that it changes the camera distance to the object. At the same time, the camera is moved around the object. This can be used to test whether the registration algorithm is able to handle change of resolution of the object itself as well as the view of the object itself is changed.

5.1.3 Evaluation metrics

This section will cover the different evaluation metrics used for evaluation of the algorithms. The two most interesting metrics to measure is the registration quality and the performance of the algorithms. The registration quality is a measure of how good the registrations are, which implies the quality and correctness of the object. The performance is the time the algorithm uses for normal payloads.

Registration quality

This section will go through multiple metrics that are used to determine the quality of the registrations.

The first metric is the **Fitness** F. This is one of the internal metrics used by the ICP algorithm used to determine convergence. This metric is based on the 2.10.1. The metric is defined as the following:

$$F = \frac{1}{N_t} \sum_{i}^{N_t} d(p_{t,i}, \min_{j \in P_m} ||d(p_{t,i}, p_{m,j}))||^2 \tag{5.1}$$

where N_t is the number of points in the registration cloud, P_m is the model cloud set, $p_{t,i}$ is point with index i in registration cloud P_t, and $p_{m,j}$ is point with index j in model cloud P_m and $d(a,b)$ is the euclidean distance between point a and b. In other words, F is the average squared distance from all points in point cloud P_t to it's closest point in point cloud P_m. This metric is good to show that the algorithms converge, but it is bad to compare different algorithms to each other as some of the ICP variations tested does not optimize this metric. It is also bad for comparing between different test-sets as they have different properties regarding correspondence rejection, number of points, etc. This motivates for the use of a metric that uses the object pose based on some reference.

The next measure will be referred to as **precision** P. In short, this measure is the distance the estimated path has from some estimated reference path. This metric will be divided into the four following sub-metrics:

- P_t^A The translational absolute error.
- P_t^R The translational relative error.
- P_r^A The rotational absolute error.
- P_r^R The rotational relative error.

All these four metrics are based on the Relative Pose Error (RPE) as defined by Sturm et al. (2012). Before computing the final error metrics, the Relative Pose Error E_i must be computed for all the captured frames using the following equation:

$$E_i = (Q_i^{-1} Q_{i+\Delta})^{-1} (P_i^{-1} P_{i+\Delta}) \tag{5.2}$$

where Q_i is some ground truth trajectory at step i and $Q_{i+\Delta}$ is the same ground truth trajectory at step $i+\Delta$ for some fixed time interval Δ. P_i is the estimated trajectory at step i and $P_{i+\Delta}$ is the estimated trajectory at step $i+\Delta$. Using the $T = A^{-1}B$ is a technique to find the transformation $TA = B$, which is the transformation that maps A to B. The RPE error is doing this twice, meaning that E_i is the transformation that maps the change of Δ in one trajectory to another with equal step size.

With this relative pose error, a total relative pose error can be computed as well as a total pose error. Sturm et al. discusses the differences between using Absolute Trajectory Error (ATE) and evaluating the RPE over all possible Δ, and conclude that there is practically no differences between them (Sturm et al. (2012)). In this report, the metrics will mostly be used to compare the implemented algorithms and not necessarily to give an absolute measure on the registration quality. Based on the RPE, Sturm et al. suggests the following metrics for pose estimation error.

$$P_t^A = \frac{1}{n} \sum_{\Delta=1}^{n} \sqrt{\frac{1}{m} \sum_{i=1}^{m} ||\text{trans}(E_i)||^2} \tag{5.3}$$

$$P_t^R = \sqrt{\frac{1}{m} \sum_{i=1}^{m} ||\text{trans}(E_i)||^2} \tag{5.4}$$

$$P_r^A = \frac{1}{n} \sum_{\Delta=1}^{n} \sqrt{\frac{1}{m} \sum_{i=1}^{m} ||\text{rot}(E_i)||^2} \tag{5.5}$$

$$P_r^R = \sqrt{\frac{1}{m} \sum_{i=1}^{m} ||\text{rot}(E_i)||^2} \tag{5.6}$$

m and n are the number of estimated relative pose errors (which is the same as number of captured frames$-\Delta$). The two are used simply to separate the two sums. Δ is some fixed time interval, and E_i is the relative pose error as calculated in equation 5.2. $||\text{trans}(E_i)||$ is the euclidean length of the translational component of the E_i matrix. In other words, it is the euclidean distance of the translation. $||\text{rot}(E_i)$ is the rotational length of the rotational component of the E_i matrix. This technique is shown in section B.1.

The RPE based evaluation metrics are sensitive to extreme outliers. These outliers are significantly off the actual path and will give large error estimations if the algorithm that can give extreme outliers. These outliers can easily be rejected during the registration, and should therefore be accounted for by some error estimation. Initially, an error metrics that took this into consideration was proposed, but as shown in section 6, most of the registrations was valid, and those metrics didn't show much extra. The suggested metrics was based on only selecting the metrics that had a translational or rotational error below a certain threshold (either by counting them or computing the RPE based metrics only on those RPE-measures). It was concluded that in the case of extreme outliers, it was a more effective measure to simply show the trajectory and show the RPE based measures from equations 5.3, 5.4, 5.3 and 5.6 instead.

Performance

Since the goal of the project is to perform point cloud registration in real-time, it is important to have a measure of performance. Given real-time considerations, the interesting factor is the time it takes to perform a registration. Therefore, "wall" time is used to measure the performance of the algorithms, which is simply to calculate the difference of the system clock before the registration is started and after it is completed. The registration framework described in section 4.3, logs the time at different stages in the program, which in turn is used to compute the wall time.

One issue with wall time is that it is dependent on unknown factors present in the operating system and computer. The operating system might be configured to give stable behaviour in this manner, but this is not done for this project as it probably is a tedious process expected to have minimal effect on the end result. Instead, the registration program is run alone on the computer (together with a bare minimum of background processes to keep the computer running) to reduce the impact of the unknown factors.

5.1.4 Experimental setup

This section will go through the parameters, filters and algorithms that where used for different experiments. It will contain a list of the experiments that was run.

Default parameters

All meaning of all arguments are explained in appendix C. The following list contain parameters that was used on all registrations.

Normals radius = 0.005 Though normals are not computed for all experiments, the normal radius is set to this value when it is computed. The normals are always computed when using the point to plane lls transformation estimator or the normal shooting correspondence estimator. It is not computed for all other tests.

	x_{min}	x_{max}	y_min	y_max	z_{min}	z_{max}
Apple	-0.15	0.15	-0.10	0.10	0.001	10.0
Banana	-0.10	0.10	-0.05	0.05	0.001	10.0
Box	-0.13	0.13	-0.12	0.12	0.001	10.0
Cap	-0.20	0.20	-0.20	0.20	0.001	10.0
Cast	-0.10	0.10	-0.07	0.07	0.001	10.0
Fruits	-0.10	0.10	-0.10	0.10	0.001	10.0
Hat	-0.13	0.13	-0.10	0.10	0.001	10.0
Monkey	-0.12	0.12	-0.08	0.08	0.001	10.0
Pear	-0.10	0.10	-0.10	0.10	0.001	10.0
Plastic	-0.10	0.10	-0.10	0.10	0.001	10.0
Pork chop	-0.20	0.20	-0.20	0.20	0.001	10.0
Lettuce	-0.20	0.20	-0.20	0.20	0.001	10.0
Pork filet	-0.20	0.20	-0.20	0.20	0.001	10.0
Salmon	-0.20	0.20	-0.20	0.20	0.001	10.0

Table 5.1: Table is showing the correspondence rejection boxes used for the different test sets. The boxes are represented using boundaries for each dimension, which is the way it is represented in the registration framework. All values are in **cm**. As all values are specified in camera coordinates, the boxes are long and thin. Note that the x and y dimensions of the hand-held scans are relatively large, as the objects are not guaranteed to be centered in the camera frame.

Color This argument is always enabled. It has no effect on the registration, but using color gives better visualizations.

Use previous transform This argument makes the registration algorithm use the transformation of the previous image as initial alignment.

Post filter A voxel grid post filter was used on all datasets. The leaf size was set to 0.001 m (1 mm). As noted in section 4.3, the post filter is applied on the accumulated cloud after registration. The reason for this is to keep the frame rate more stable and keep the accumulated point cloud to grow too big.

Default correspondence box

Some of the point tests used the correspondence box rejector described in section 4.5. For a given test set, the same correspondence rejection box was used. Table 5.1 shows the boundaries of the correspondence rejection boxes for each of the test sets that used it.

Best registration experiment

The first test presented is simply the currently best found parameters for registration of each of the scenes described in section 5.1.2. The strategy for finding these parameters was simply trial and error, where new arguments was tried until the result was satisfactory. Later chapters will explore some of these parameters in more depth. This experiment is designed to give some good initial default parameters. The following list contains all parameters that was used for all test sets for the best registration:

Transformation estimation = ”point to plane lls” This was found to give the best and fastest convergence for the best registrations found.

Correspondnece estimation = ”nearest point”

Fitness epsilon = $1 \cdot 10^{-8}$

Transformation epsilon = $1 \cdot 10^{-8}$

Max iterations = 50

Algorithm = ”ICP kD-tree GPU” Given a large enough stack, the GPU implementation performed as good as the PCL implementations.

Stack size = 20 This was deep enough for most point clouds in this project.

Defaults Default arguments as specified in section 5.1.4.

The only remaining arguments is the correspondence rejection box. The boundaries are defined in table 5.1. The correspondence rejection box was only used for the banana, cap, cast, hat, monkey, plastic, pork-chop, pork-filet, salmon and lettuce datasets. The tests used a frame rate of 2.5 FPS, resulting in a total of 34 images. Note that even though the frame rate is 2.5 FPS, it might use more time for the actual registration.

The results from these registrations are shown in figure 6.1 and 6.2. Additionally, the evaluation tool 4.6.5 was used to plot the estimated trajectory from the registrations and the error metrics from section 5.1.3. This is shown in figure 6.3 and table 6.1.

Iteration count experiments

These experiments were created to show how the number of iterations effect the registration performance and quality. The idea is to set the fitness epsilon and transformation epsilon sufficiently low, such that the iteration count is the limiting termination criteria. The following list were the default arguments used for this experiment:

Fitness epsilon = $1 \cdot 10^{-10}$

Transformation epsilon = $1 \cdot 10^{-10}$

Algorithm = "ICP kD-tree GPU" Given a large enough stack, the GPU implementation performed as good as the PCL implementations.

Stack size = 20 This was deep enough for most point clouds in this project.

Defaults Default arguments as specified in section 5.1.4.

Different combinations of transformation estimation, correspondence estimation and algorithm are tried while the number of iterations are varied. This test set was run on the fruits dataset, using a framerate of 1.25 FPS giving a total of 16 images to register. Note that even if the framerate is 1.25 FPS, the registration might use more time per image than this. The results from this is shown in figures 6.8 and 6.9. Additionally, table 6.2 is provided to show how many of the registrations that were successful for the different variations tried.

Fitness epsilon experiments

This experiment was identical to the iteration count experiment in section 5.1.4 except for the fitness epsilon and iteration count. The fitness epsilon was varied while the maximum iteration count was kept constant at 500. The results from this experiment is shown in figures 6.10 and 6.11. Additionally, table 6.3 is provided to show how many of the registrations that were successful for the different variations tried. Refer to section 5.1.4 for more details and default parameters.

Transformation epsilon experiments

This experiment was identical to the iteration count experiment in section 5.1.4 except for the transformation epsilon and iteration count. The transformation epsilon was varied while the maximum iteration count was kept constant at 500. The results from this experiment is shown in figures 6.12 and 6.13. Additionally, table 6.4 is provided to show how many of the registrations that were successful for the different variations tried.. Refer to section 5.1.4 for more details and default parameters.

Experimenting with frame rate

Another interesting question regarding real-time registration is requirements on frame-rate. This test is designed to vary the frame rate and evaluate the registration quality and performance under those changes. The following list contains the constant registration parameters that was run for all the image tests:

Max iteration count = 50

Fitness epsilon=$1 \cdot 10^{-8}$

Transformation epsilon=$1 \cdot 10^{-8}$

Stack size = 20 This was deep enough for most point clouds used in this project.

Defaults Default arguments as specified in section 5.1.4.

Different combinations of transformation estimation, correspondence estimation and algorithms was tried while the frame rate was varied. The results from this is shown in figures 6.14, 6.15 and 6.16. Additionally, table 6.5 is provided to show how many of the registrations that were successful for the different variations tried. The dataset used for this experiment is the ”fruits” dataset. Consult appendix C for more details on the arguments used.

Scan method experiments

The scan methods introduced in section 5.1.2 are evaluated by testing a few registration algorithms. The goal is to evaluate how good the registration algorithms are to track different scan methods. The following registration arguments were used for all experiments regarding the scan method:

Max iteration count = 50

Fitness epsilon=$1 \cdot 10^{-8}$

Transformation epsilon=$1 \cdot 10^{-8}$

Correspondence estimation = ”nearest point” This was used as there was no significant differences between the different correspondence estimators (further discussed in section 7.8)

Stack size = 20 This was deep enough for most point clouds used in this project.

Defaults Default arguments as specified in section 5.1.4.

The transformation estimation, algorithm and scan method was varied for the different scan methods. The results from these experiments are shown in table 6.6 and figure 6.5, both generated using the evaluation tool 4.6.5. Each test set was designed to approximately 16 images, which gave slightly different results in both frame rate and number of images. This was the results as a steady frame rate was desired. Though the frame rate is specified, the actual registration might have used more or less time. The following list contains the actual number of images and frame rates used in the registrations:

Square 16 images was used, giving a frame rate of 1 FPS.

Long swipe 16 images was used, giving a frame rate of 2.5 FPS.

Short swipe 17 images was used, giving a frame rate of 3.33 FPS.

Translation 16 images was used, giving a frame rate of 2 FPS.

Triangle 15 images was used, giving a frame rate of 1.11 FPS.

Zoom move 19 images was used, giving a frame rate of 3.33 FPS.

Zoom 17 images was used, giving a frame rate of 2.5 FPS.

Stack size experiments

The major question regarding the proposed kD-tree search method in section 4.4.3 and 4.4.4 is how the stack size affects the registration performance and quality. The following registration parameters where the defaults used for the experiments used to evaluate the effect of stack size:

Max iteration count = 50

Fitness epsilon=$1 \cdot 10^{-8}$

Transformation epsilon=$1 \cdot 10^{-8}$

Correspondence estimation = ”nearest point” This was used as there was no significant differences between the different correspondence estimators (further discussed in section 7.8)

Algorithm = ICP kD-tree GPU This is the GPU implementation of the kD-tree search as described in section 4.4.4.

Defaults Default arguments as specified in section 5.1.4.

The dual quaternion and point to plane lls transformation estimator was evaluated while varying the stack size of the kD-tree serach. The results from this is shown in figure 6.18, 6.19. The frame rate was kept constant at 1.25 FPS giving a total of 16 images. Though the frame rate is 1.25 FPS, the registration might have a faster or lower frame rate depending on the registration time. The test set is the ”banana” test set.

Additionally, the fitness score was evaluated for different stack sizes and different iteration counts. The test is identical to that described above, the only difference is described in the following list:

Fitness epsilon=$1 \cdot 10^{-12}$

Transformation epsilon=$1 \cdot 10^{-12}$

Experiments with the voxelized ICP algorithm

This test was a quite extensive test evaluating both the basic voxelized ICP algorithm and the suggested improvements. This test experiment also included a comparison using a voxel grid filter to compare the voxelized algorithms performance and correctness to other similar approaches. As with the previous tests, this test contains a few default parameters that is presented in the following list:

Max iteration count = 50

Fitness epsilon=$1 \cdot 10^{-8}$

Transformation epsilon=$1 \cdot 10^{-8}$

Defaults Default arguments as specified in section 5.1.4.

The other parameters such as transformation estimation, voxel size, number of voxel search levels, voxel level downscale and pre-search range was varied and evaluated (read section C for the meaning of these parameters). The frame rate was kept at 2.5 FPS (note that this is referring to the capture frame rate and not the registration frame rate). The test set used was the ”fruits” dataset. The different approaches varying voxel size, voxel search levels and voxel level downscale is shown in figure 6.21a and figure 6.23a.

A similar method to the one above was performed for the pre-search strategy. The voxel search levels and voxel level downscale parameters are not used, and the pre-range search is used instead. The results from this is shown in figure 6.21b and figure 6.23b.

Similarly, the same parameters was used for the voxel grid search filter. The voxelized algorithm was replaced with the kD-tree GPU ICP algorithm. The voxel size (also called leaf size) was varied, and the results are shown in figures 6.22 and 6.24.

CHAPTER 6

Results

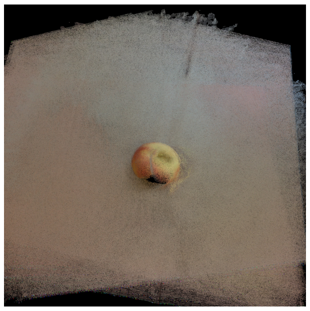
This chapter will show the results from the experiments described in chapter 5.1.

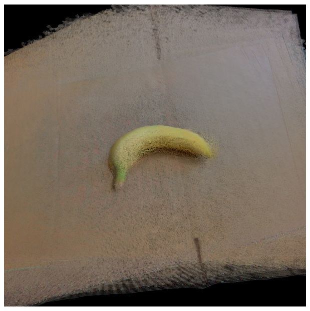
6.1 Registration results

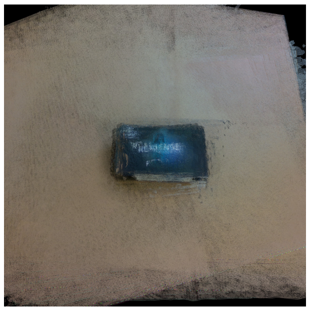
Section 5.1.4 described how the default parameters were found to give good registration results. A point cloud visualization of these parameters is shown in figure 6.1 for each of the robot-based scans. Similarily, figure 6.2 shows a point cloud visualization when using the best found parameters for the free-hand scans. These figures show that a good registration is found, that has converged to the correct minima for all images. No other registration is expected to be significantly better than these, and they are considered good enough for a robotics application. Based on this, table 6.1 shows the RPE based error measures for the registrations shown in figure 6.1. Because no other registrations are expected to give better results than those shown in figure 6.1, table 6.1 is used as a reference on what to expect from a good registration given the setup explained here. The corresponding estimated trajectories from the registrations are shown in figure 6.3. In this figure, the estimated trajectory from the registration is plotted together with the estimated reference trajectory from the robot. By comparing table 6.1 and figure 6.3, it's possible to see the relation between the estimated trajectory and the corresponding error measures.

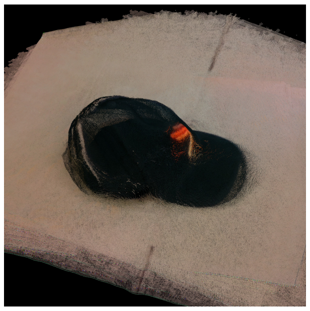
6.2 Camera differences

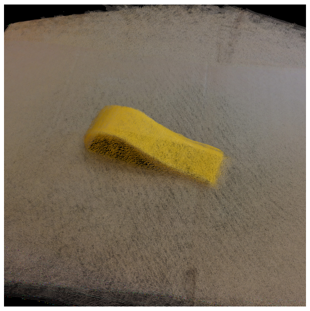
This section is based on an observation when scanning some food objects in which a distortion pattern was observed when using the SR300 camera. Figure 6.4a and 6.4c shows a comparison

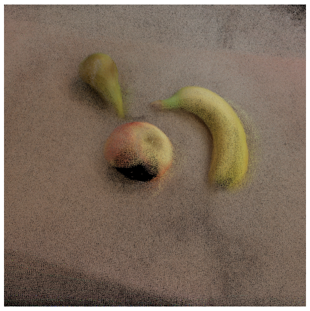

(a) Apple

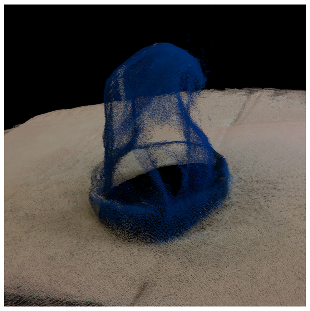

(b) Banana

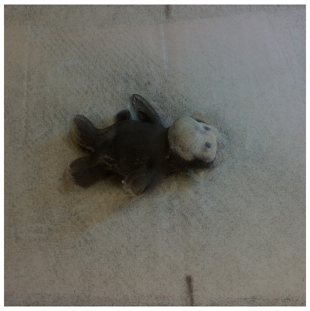

(c) Box

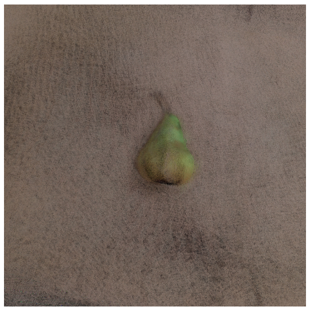

(d) Cap


(e) Cast


(f) Fruits


(g) Hat


(h) Monkey


(i) Pear

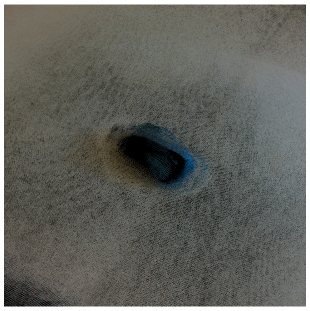
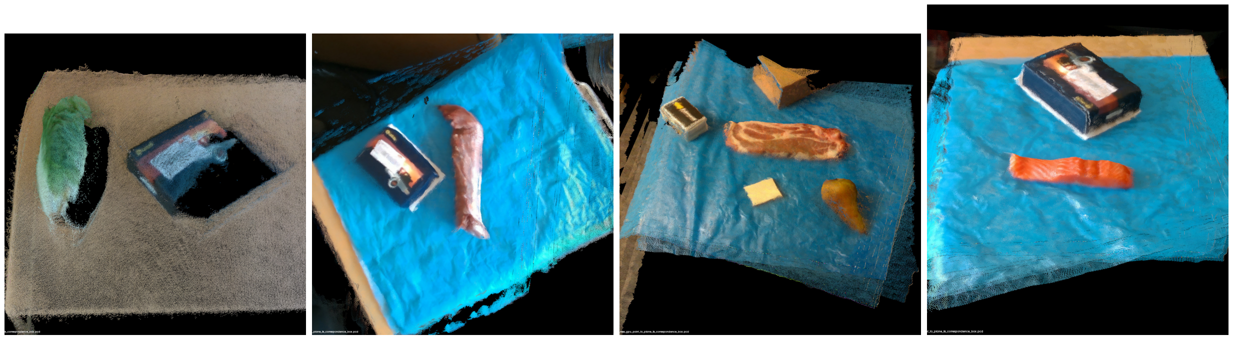

(j) Plastic

Figure 6.1: Figure showing the accumulated point clouds of the objects described in section 5.1.2. This figure is considered an near-optimal registration of the objects shown in the figure, meaning that it's not expected to get any better registrations in this project, and that the registration quality of these images are good enough for a robotics application. Table 6.1 shows the RPE based registration errors from these images. The figures are generated using the PCL viewer tool in the PCL package.

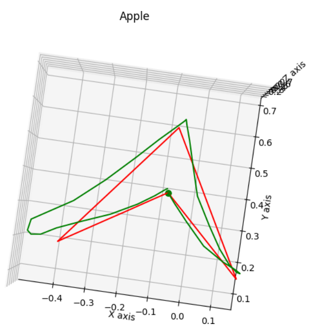


(a) Lettuce (b) Pork filet (c) Pork chop (d) Salmon

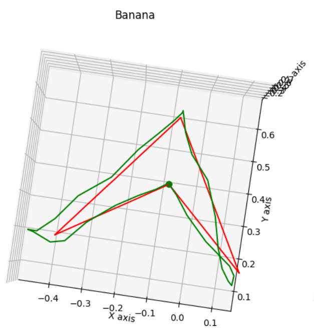
Figure 6.2: Figure showing the accumulated point clouds of the objects described in section 5.1.2. This figure is considered a near-optimal registration, meaning that it's not expected to get any better registrations in this project, and that the registration quality of these images are good enough for a robotics application. Note that there are placed a lot of boxes in the scene. This was done due to an earlier version of the registration framwork that didn't support point correspondence rejectors. It's unknown if these reference objects actually are required given the latest version of the registration framework, but the registration would probably be fine even without the extra objects added to the scene, though the pork-chop and salmon would be harder than the pork filet and lettuce. The figures are generated using the PCL viewer tool provided with the PCL package.

	P_t^R	P_t^A	P_r^R	P_r^A
apple	0.032	0.091	0.131	0.397
banana	0.034	0.091	0.130	0.386
box	0.032	0.090	0.130	0.394
cap	0.031	0.090	0.127	0.387
cast	0.034	0.092	0.128	0.387
fruits	0.032	0.091	0.130	0.393
hat	0.030	0.088	0.124	0.382
monkey	0.031	0.089	0.126	0.386
pear	0.031	0.353	0.133	0.482
plastic	0.034	0.093	0.129	0.385

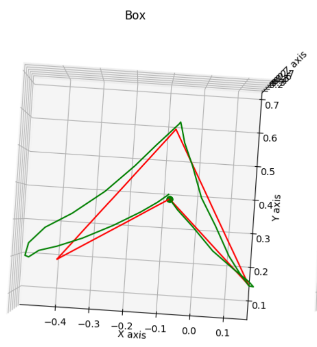
Table 6.1: Table showing the error metrics (as described in section 5.1.3) for the registrations explained in section 5.1.4, and shown in figure 6.1. These registration are considered to be near-optimal, meaning that no other registration is expected to noticeable better. This table is therefore used as a reference on what the evaluation metrics show for good registrations. As explained in section 5.1.3, P_t^A is the RMSE of the RPE error for each frame, P_t^A is the RMSE of the absolute RPE error. Both is the translational components. P_r^R and P_r^A are the same metrics applied to the rotational error instead of the translational. This table shows that each frame has about 3 cm of drift from the reference path (could possibly be camera calibration issues or timing issues). Note that the fitness measure is not included.



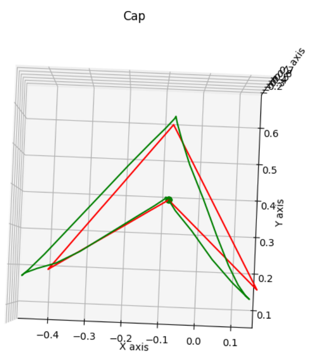
(a) Apple



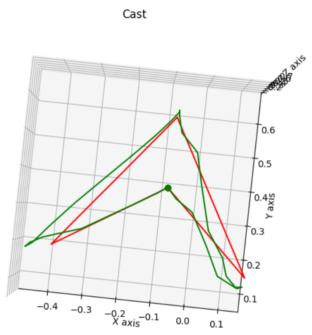
(b) Banana



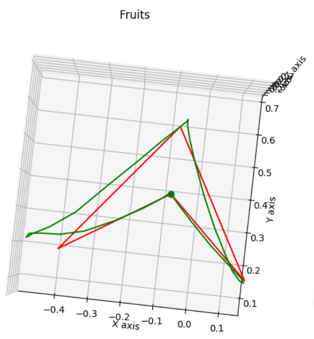
(c) Box



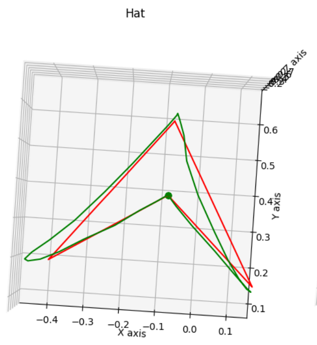
(d) Cap



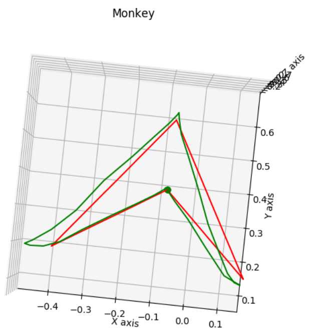
(e) Cast



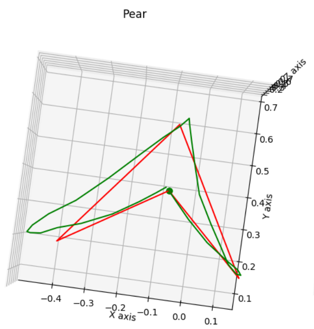
(f) Fruits



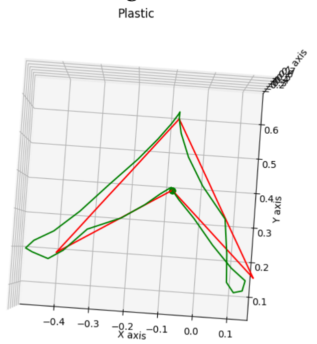
(g) Hat



(h) Monkey



(i) Pear



(j) Plastic

Figure 6.3: Figure showing the estimated trajectories of the scans shown in section 6.1 and figure 6.1. The estimated paths (green line) are plotted together with the reference path (red line) obtained from the robot pose estimation. The robot trajectory for these scans are the triangle scan trajectory from figure 6.5b. These figures are shown from a top view (looking down on the objects) which is the most interesting view for the reference path plot. The 3 cm drift per frame observed in 6.1 is reasonable considering this figure.

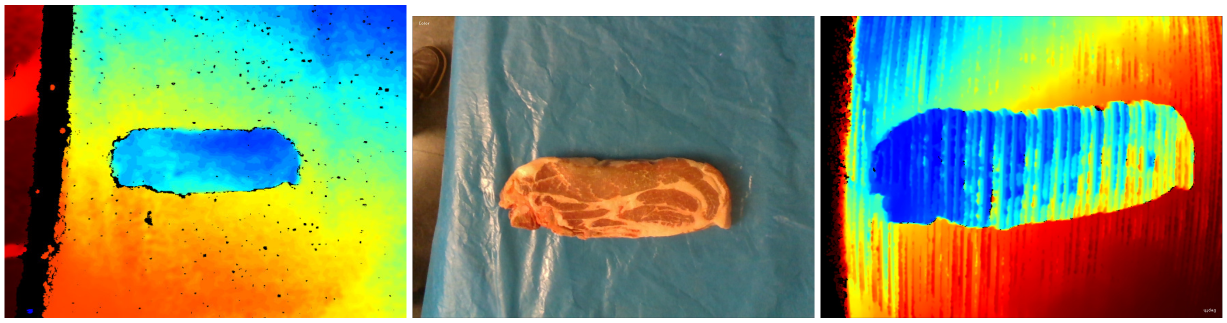

(a) D435 Depth (b) D435 Color (c) SR300 Depth

Figure 6.4: Figure showing some distortion artifact observed using the SR300 camera. (a) shows a depth image of a pork chop using the Intel RealSense D435 camera, (b) shows the same object with the color camera of the D435 camera and (c) shows a depth image of the same object using the Intel RealSense SR300 camera. The images are screenshots from the rs-capture tool provided with the Intel RealSense software (Intel (2018a)).

between the depth images from the SR300 camera and the D435 camera, showing the distortion pattern observed with the SR300 camera. The reason this pattern is observed is briefly discussed in section 7.6.

Other than the distortion patter, there are a few other differences between the cameras. Some of them will be mentioned instead of visualized as the differences are less significant. All of the following observations are observed for this project.

The D435 get certain ”holes” in the depth stream indicating unknown depths, which is not the case for the SR300 camera. This is also seen in figure 6.4. This can be avoided by using hole-filling techniques in software, but in general, these ”holes” are not an issue for a registration of reconstruction. Another difference is that the D435 camera works better for further distances. As the D435 camera is a stereo camera, it isn’t as dependent on the projected pattern as the SR300 camera. However, when scanning smaller objects close to the camera as done in this project, the SR300 camera generally gives better images and works better at close distances than the D435 camera. Another observation is that the SR300 is somewhat more limited on the surfaces it is able to scan. The SR300 is unable to reconstruct the depth of objects with certain optical properties. This is typically absorption and reflections that are present in certain types of plastics. The plastic example in figure 6.1i is an example of one such object.

6.3 Camera and robot paths

This section will show the estimated camera trajectories for the robot scans and free-hand scans.

6.3.1 Robot trajectories

Section 5.1.2 describes all the different robot-scan trajectories used for this project. The estimated trajectories based on the internal pose estimation of the robot is shown in figure 6.5. The figure clearly shows that the robot moves in a linear accelerating motion between waypoints. The markings on the path is setup at regular intervals, and indicate the orientation of the camera. They represent the coordinate axes of the camera (red is x, green is y, and blue is z). This shows that for most scans, the blue line (viewing direction of the camera) is pointing towards a center point, which is the approximately at the center of the object. The figures show that the robot follows the expected trajectory based on the waypoints and movement methodology described in section 4.1.

To make the trajectory show the camera trajectory instead of the end-effector trajectory, a transformation matrix was applied to all the end-effector poses. This transformation matrix was found using the calibration guide by VISP (2018). The resulting transformation matrix is the following:

$$T = \begin{bmatrix} 0.04764713966 & -0.9410012188 & -0.3350320227 & 0.02151870377 \\ 0.9980178969 & 0.05865345273 & -0.02280460337 & -0.01713208633 \\ 0.04110994447 & -0.3332813805 & 0.9419307267 & -0.048754478 \\ 0001 \end{bmatrix}$$

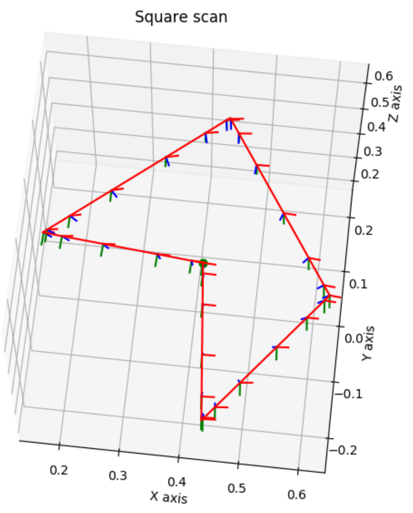
This matrix was applied to all poses obtained from the Franka robot before plotting or comparison with the reference trajectories obtained from the robot.

6.3.2 Estimated free-hand trajectories

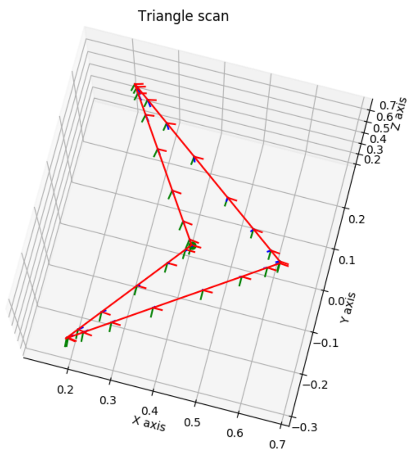
Section 5.1.2 describes how the free-hand scans are performed. The estimated camera trajectory estimated from doing the registrations are shown in figure 6.6. As with the robot scans, the markings are placed at regular intervals to indicate the orientation of the camera (see section 6.3.1). The trajectories look similar to the actual trajectories done, though no external tracking was performed to verify this claim, but the registrations shown in figure 6.2 indicate that the registrations was quite good, meaning that these trajectories are relatively precise.

6.4 ICP variations

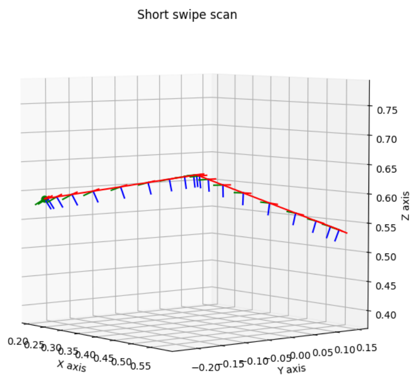
Section 4.6.4 describes an experiment to find good convergence criteria. This section will focus on the results from these experiments, and will show both performance and error-measures.



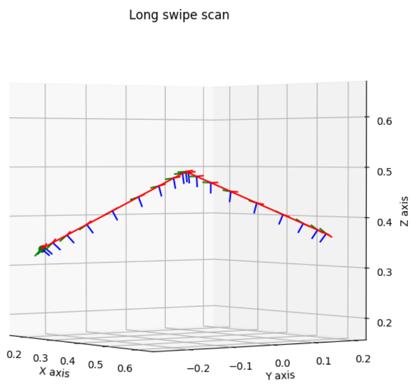
(a) Square scan



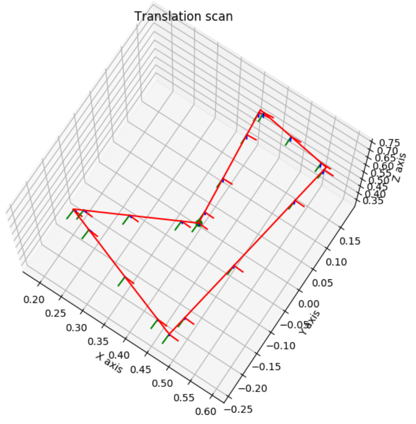
(b) Triangle scan



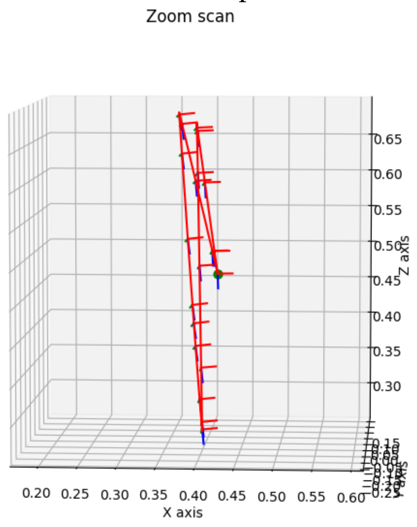
(c) Short Swipe scan



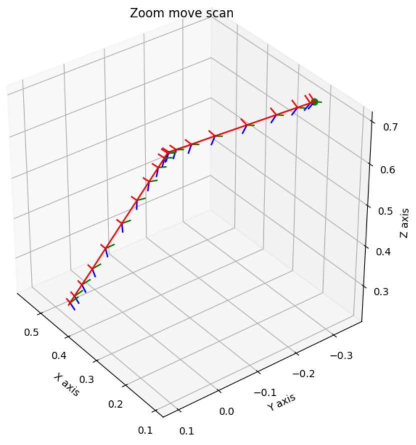
(d) Long swipe scan



(e) Translation scan

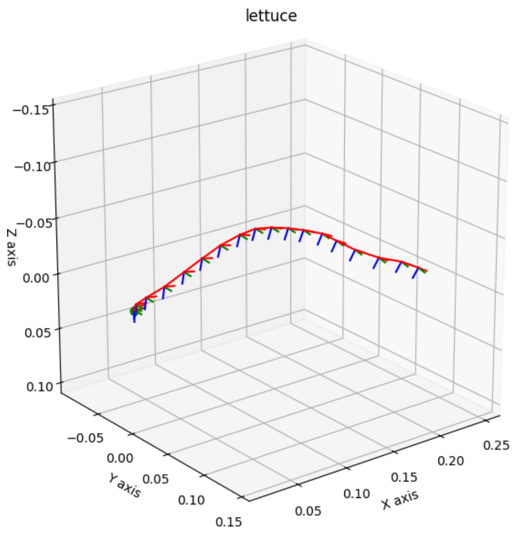


(f) Zoom scan

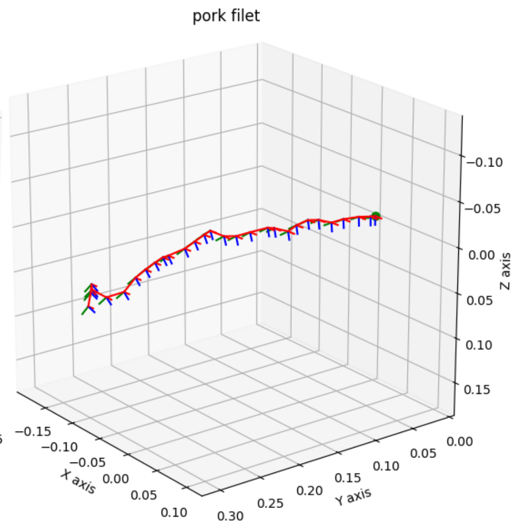


(g) Zoom movement scan

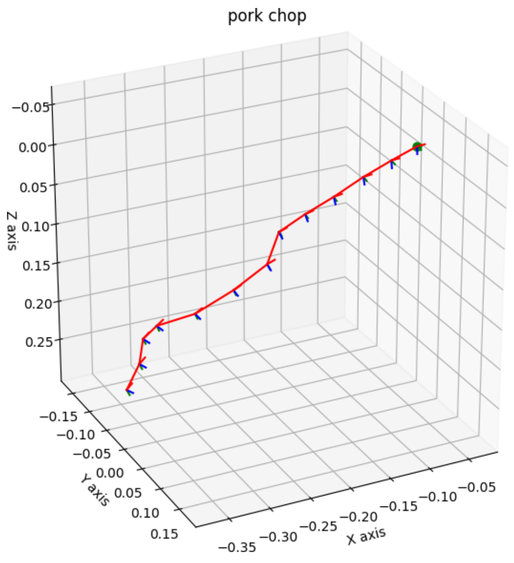
Figure 6.5: Figure showing the paths of the camera based on the estimated pose of the robot. The viewpoint of figure is positioned along the axis of least variance (meaning that as little as possible of the 3D information of the figure is removed in the projection). The markings on the trajectory indicate the coordinate axes of the camera at a given time (red is x, green is y and z is blue).



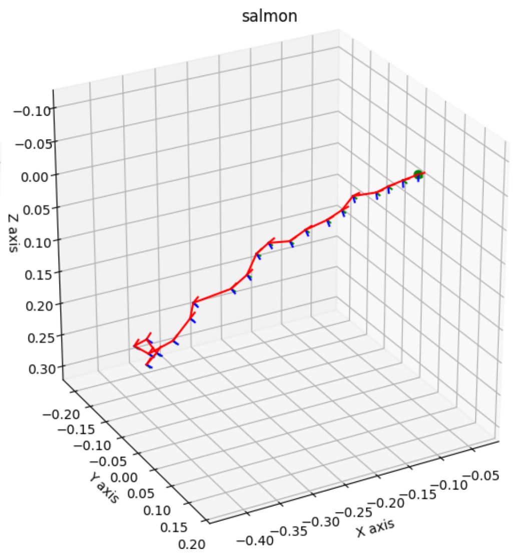
(a) Lettuce



(b) Pork filet



(c) Pork chop



(d) Salmon

Figure 6.6: Figure showing the estimated trajectories of the free-hand scans from section 5.1.2 whose accumulated point clouds are shown in figure 6.2. The markings on the estimated trajectory correspond to where the individual images are taken (which is not the case in figure 6.5). The trajectories show a realistic path based on how the real scans were performed which indicate a good registration result.

	Number of iterations					
	1	8	16	32	64	128
icp voxel gpu, nearest point, dual quaternion	4	1	1	1	1	1
icp gpu, nearest point, dual quaternion	1	1	1	1	1	1
icp gpu, nearest point, point to plane lls	15	3	1	1	1	1
icp voxel gpu, nearest point, point to plane lls	15	0	0	4	0	0
icp voxel, nearest point, point to plane lls	0	0	0	0	0	0

Table 6.2: Table shows the number of valid registrations performed for different variations of the ICP algorithms shown in figure 6.8 and 6.9. As the test set contains 15 images, it's desirable to have 15 in all cells in the table. This means that the registration parameters were suboptimal for the algorithms that show fewer than 15 successful registrations. Read this together with figures 6.8 and 6.9. Note that the algorithms that performed all 15 registrations correctly are left out of this table.

6.4.1 Trajectory estimation

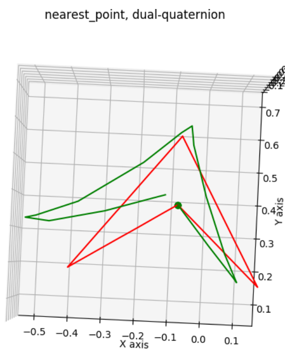
First of all, figure 6.7 shows the estimated path from some of the different ICP variations evaluated. Note that compared to figure 6.3, figure 6.7 use half the number of images and does not include the last image. Therefore, the reduced precision and lack of loop closure is expected.

6.4.2 Varying number of iterations

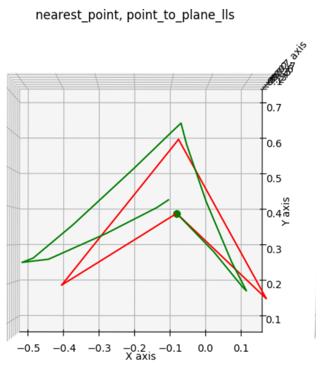
This section will focus on how some of the evaluation metrics change when varying the maximum number of iterations in the ICP algorithm.

Figures 6.8a and 6.8b show the running times of different versions of the ICP algorithm when varying the maximum number of iterations. This figure does not show the entire truth because the point to plane lls transformation estimator reached convergence based on the fitness or transformation epsilon (section 7.8) before reaching the iteration count criteria. These variations simply converged a lot faster than expected. Also note that a few algorithms have been left out because the registrations failed. The ICP voxel GPU implementation also has a drop in running time when increasing number of iterations. This is also because the registration fails in the middle of the search, giving better results than expected.

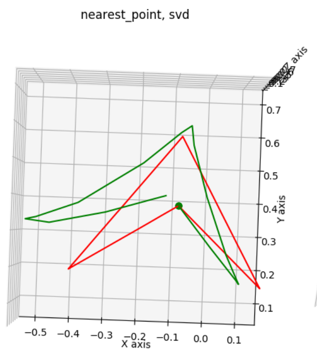
The translational RMSE of the RPE error corresponding to the running times shown in figure 6.8 is shown in figure 6.9. As multiple of the algorithms might be unable to perform the registrations, this figure should be read together with table 6.2, which shows the number of valid registrations for each configuration of the algorithm. The table has only included the algorithms that got at least one invalid registration.



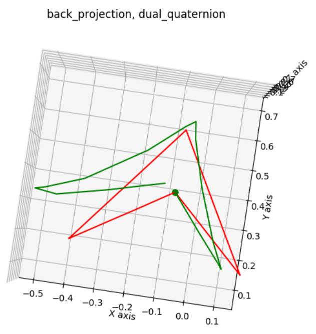
(a) Nearest point dual quaternion



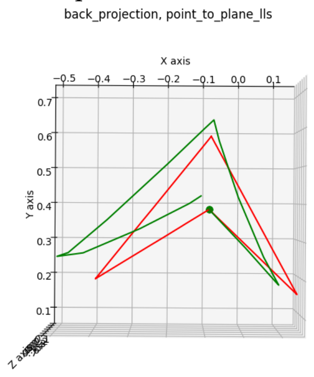
(b) Nearest point point to plane lls



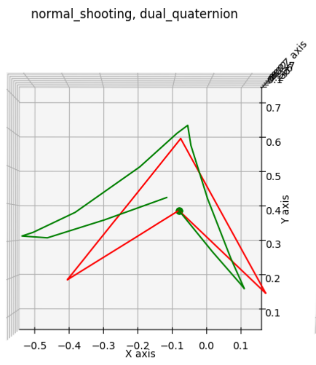
(c) Nearest point svd



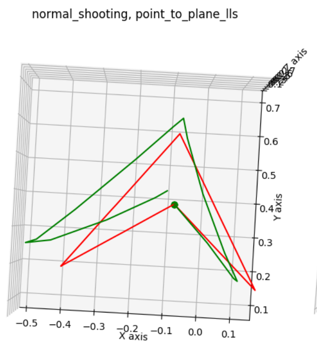
(d) back projection dual quaternion



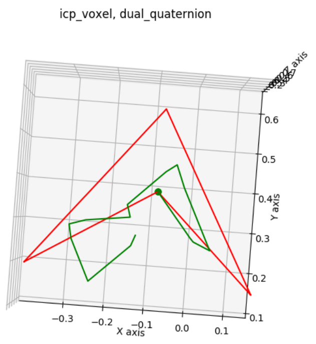
(e) back projection point to plane lls



(f) normal shooting dual quaternion

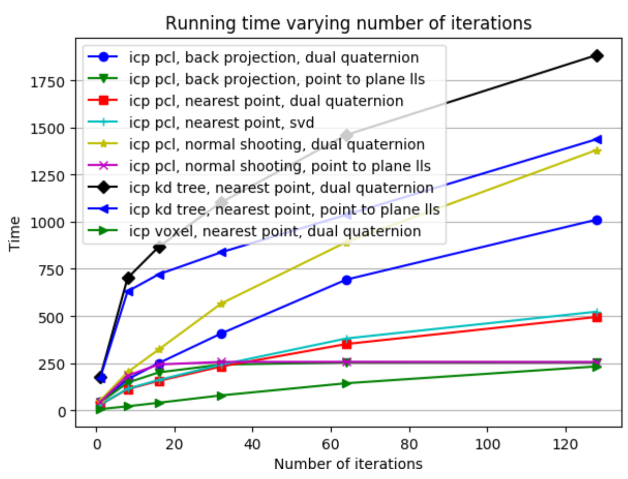


(g) normal shooting point to plane lls

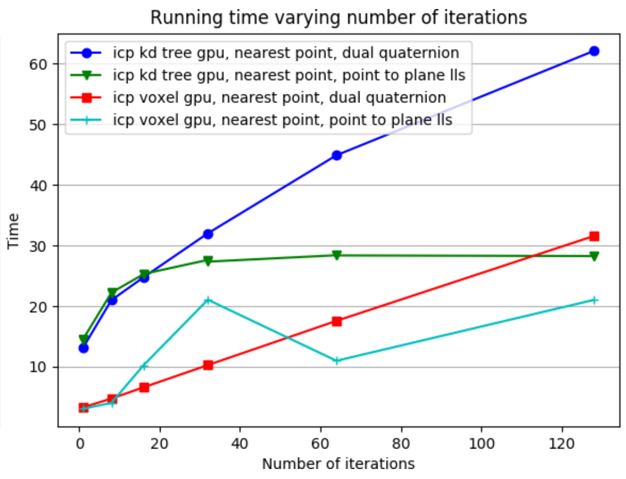


(h) voxelizied dual quaternion

Figure 6.7: Figure showing the estimated trajectory of the different icp-variations evaluated. Note that the voxelized search with the point to plane lls transformation estimator is not shown here because it was unable to find correspondences. The figure shows a slight drift from the optimal registrations in figure 6.3, but also note that this registration used half of the framerate, which also removed the final image (thereby the lack of loop closure on a few of the scans). This figure show that there is little difference between the algorithms. The dual quaternion and SVD transformation estimators shown in (a), (c), (d), (f) and (h) have a slightly larger drift to the left side of the image compared to the tighter fit of the point to plane lls transformation estimator shown in (b), (e) and (g). The voxelized ICP variation sticks out as a lot worse than the other variations in (h).

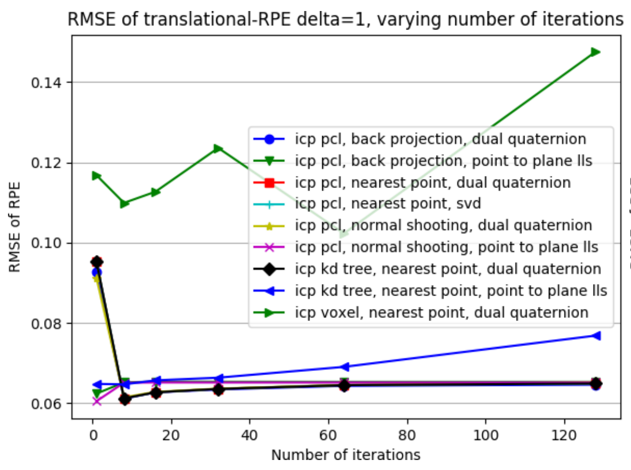


(a) Slowest ICP variations

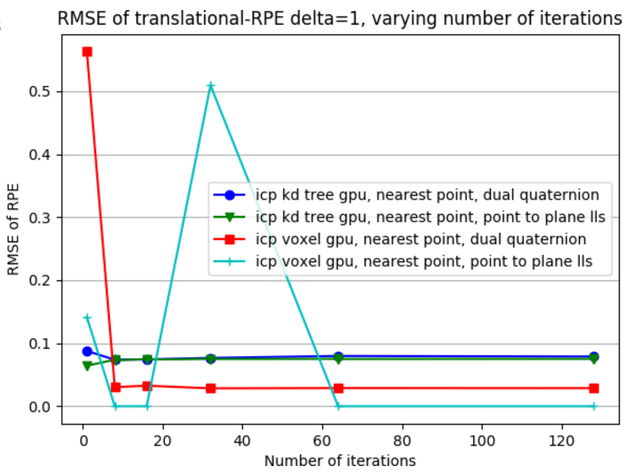


(b) Slowest ICP variations

Figure 6.8: Figure showing the total running time of different variations of the ICP algorithms run on the fruits dataset with 16 images when varying iteration count. The slowest algorithms are shown in (a) and the fastest in (b), and time is measured in seconds. Note that a few algorithms have been left out because the registration failed (see table 6.2). Also note that ”icp voxel gpu, point to plane lls” algorithm also failed the registration which is the cause for the strange running time drop in (b). The point to plane lls algorithms terminated unexpectedly fast, reaching some other convergence criteria before the iteration count criteria, which explains the flattening in the figure.

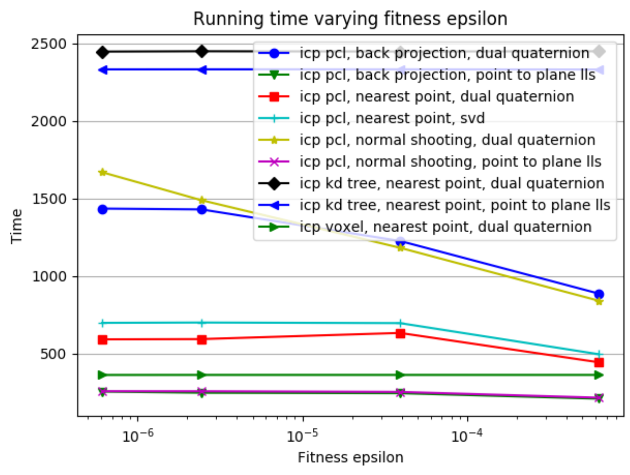


(a) Slowest ICP variations

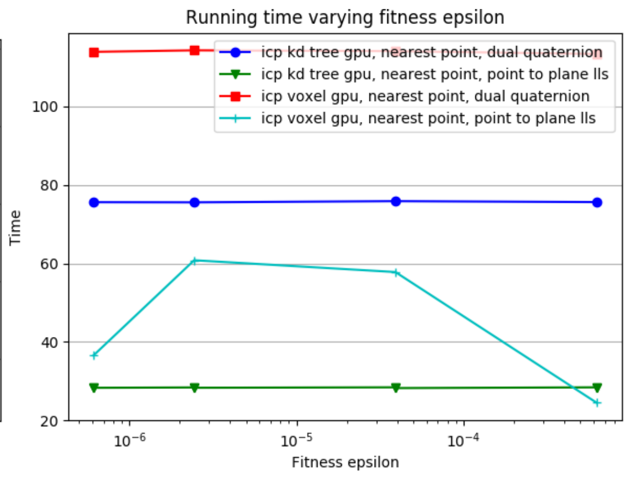


(b) Fastest ICP variations

Figure 6.9: Figure showing the RMSE of the translational RPE error with $\Delta = 1$ of different variations of the ICP algorithms varying iteration count. The slowest algorithms are shown in (a) and the fastest in (b). Note that a few algorithms have been left out because the registration failed (see table 6.2 for a list of the algorithms that failed. It is adviced to read that table together with this data). Also note that a few of the algorithms are hidden behind the black line because the function quite similar, and therefore gives similar error values. The zero values in (b) means that no valid registration was found.



(a) Slowest ICP variations



(b) Slowest ICP variations

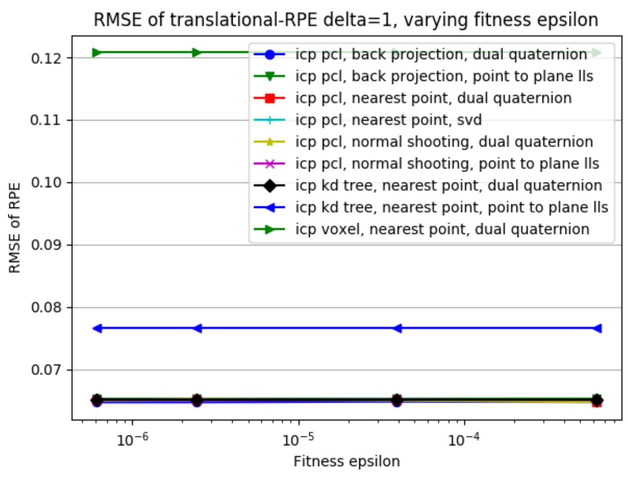
Figure 6.10: Figure showing the total running time of different variations of the ICP algorithms run on the fruits dataset with 16 images when varying fitness epsilon. The slowest algorithms are shown in (a) and the fastest in (b), and time is measured in seconds. Note that a few algorithms have been left out because the registration failed (see table 6.3). The figure shows that the fitness epsilon has little effect on running time. This was because the fitness epsilon was set too high, and the algorithm terminated immediately.

6.4.3 Varying fitness epsilon

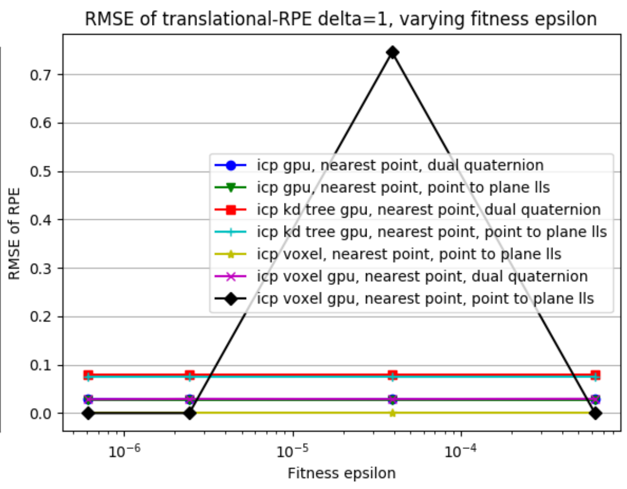
As mentioned in section , the fitness epsilon is one convergence criteria. This section will show some evaluation metrics of different variations of the ICP algorithm when varying the fitness epsilon termination criteria.

Figure 6.10 shows the running times of different variations of the ICP algorithm when varying the fitness epsilon convergence criteria. A few variations have been left out of the figure because they gave invalid results (these are shown in table 6.3). The figure shows that performance is somewhat better for a larger fitness epsilon, but for many of the algorithms, the graph looks flat. This indicates that the fitness epsilon values slightly missed the important area where the change is more significant. All the selected values were quite high, and given the unexpected fast convergence of the point to plane lls transformation estimator, it quickly skipped fitness epsilon values tested. The ICP algorithm is designed to minimize this metric, and it' is therefore expected to drop quite fast.

The corresponding error estimations of the running times shown in figure 6.10 is shown in figure 6.11. Table 6.3 shows how many valid registrations done for some of the datasets. The ones that are not shown in the table had all 15 registrations valid. This should be read together with figure 6.11. As mentioned in the previous paragraph, the error measures presented in figure



(a) Slowest ICP variations



(b) Fastest ICP variations

Figure 6.11: Figure showing the RMSE of the translational RPE error with $\Delta = 1$ of different variations of the ICP algorithms varying fitness epsilon. The slowest algorithms are shown in (a) and the fastest in (b). Note that a few algorithms have been left out because the registration failed (see table 6.3 for a list of the algorithms that failed. It is adviced to read that table together with this data). Also note that most of the algorithms are hidden behind bottom the black line. The zero values in (b) means that no valid registration was found. The figure shows that selected fitness epsilon gave no variation in the error, because the values were to high, and the algorithm terminated immediately.

6.11 are also quite flat, meaning that the tested fitness epsilon values was reached too quickly, not showing any differences in the registration.

6.4.4 Varying the transformation epsilon

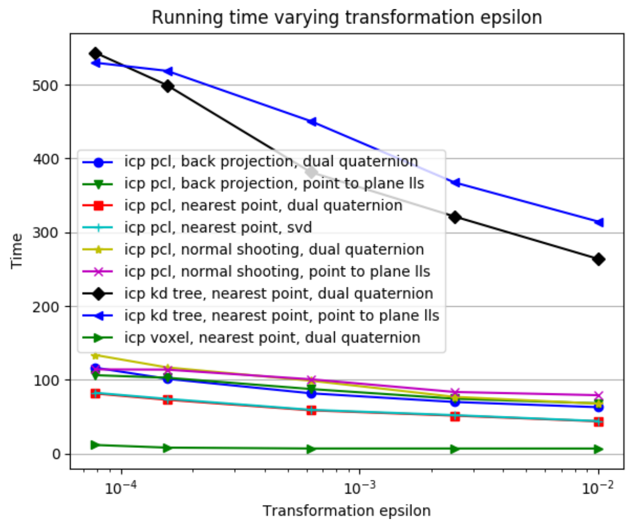
The final convergence criteria available in PCL is the transformation epsilon. This section will show a few evaluation metrics when varying the transformation epsilon.

The running time of the same variations of the ICP algorithm as used previously is shown in figure 6.12. This figure clearly shows that the performance is increased as the transformation epsilon is increased.

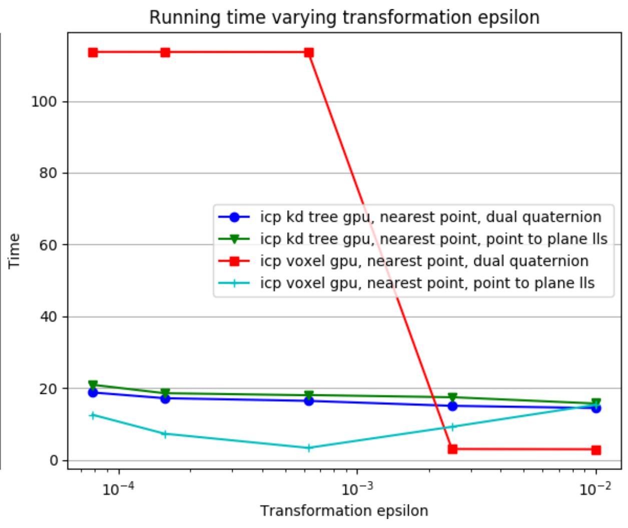
The corresponding error metrics to the running times shown in figure 6.12 is shown in figure 6.13. The figure show that the error is slightly increased as the transformation epsilon is increased. However, for the GPU implementations, the error is not changed much when increasing the transformation epsilon.

	Fitness epsilon			
	6.10352e-07	2.44141e-06	3.90625e-05	0.000625
icp voxel, nearest point, point to plane lls	0	0	0	0
icp gpu, nearest point, point to plane lls	1	1	1	1
icp voxel gpu, nearest point, point to plane lls	0	0	2	0
icp gpu, nearest point, dual quaternion	1	1	1	1
icp voxel gpu, nearest point, dual quaternion	1	1	1	1

Table 6.3: Table shows the number of valid registrations performed for different variations of the ICP algorithms shown in figure 6.10 and 6.11. As the test set contains 15 images, it's desirable to have 15 in all cells in the table. This means that the registration parameters were suboptimal for the algorithms that show fewer than 15 successful registrations, and they failed after zero or a few images. Read this together with figures 6.10 and 6.11. Note that the algorithms that performed all 15 registrations correctly are left out of this table.

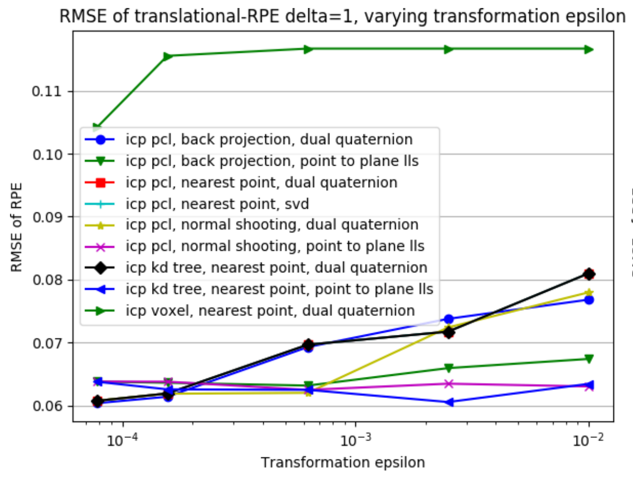


(a) Slowest ICP variations

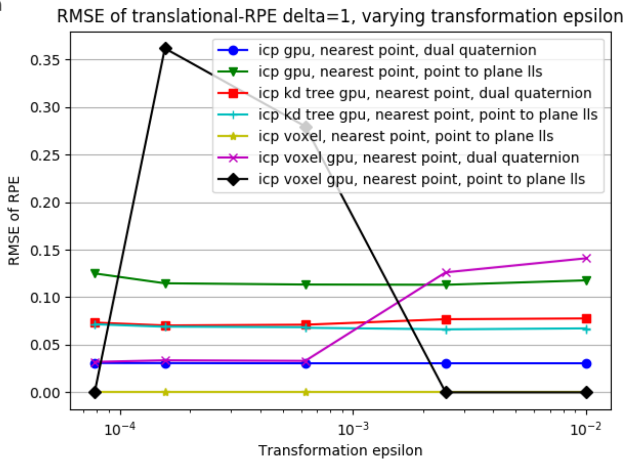


(b) Slowest ICP variations

Figure 6.12: Figure showing the total running time of different variations of the ICP algorithms run on the fruits dataset with 16 images when varying the transformation epsilon. The slowest algorithms are shown in (a) and the fastest in (b), and time is measured in seconds. Note that a few algorithms have been left out because the registration failed (see table 6.4). The "ICP voxel" timings in (b) is strange because it was unable to register all images. Figure shows that slacking the transformation epsilon does improve performance. The zero value indicates that the registration was invalid.



(a) Slowest ICP variations

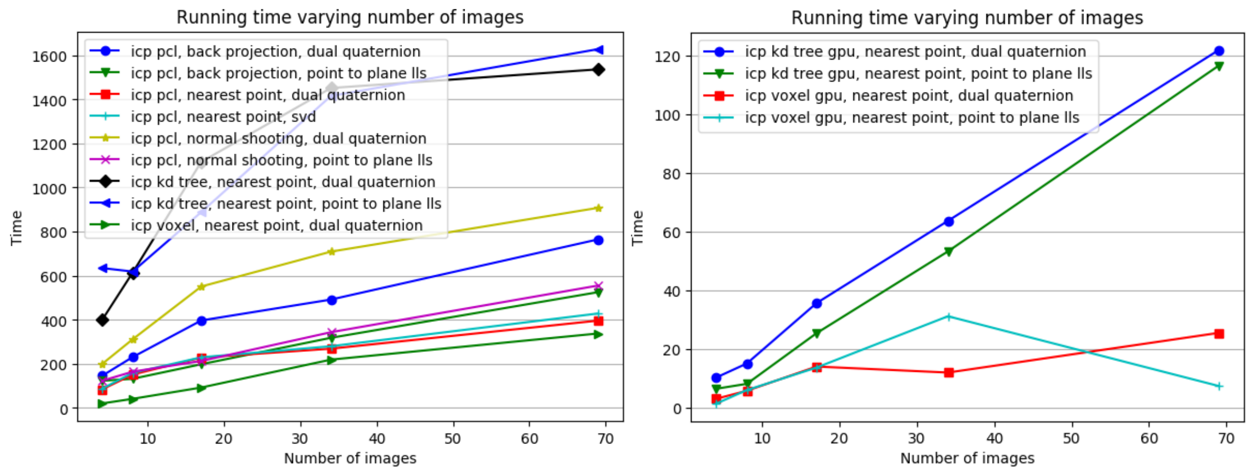


(b) Fastest ICP variations

Figure 6.13: Figure is showing the RMSE of the translational RPE error with $\Delta = 1$ of different variations of the ICP algorithm varying transformation epsilon. The test is run on the ”fruits” test set with a total of 16 images. The slowest algorithms are shown in (a) and the fastest in (b). All algorithms have been included, but some of them contains invalid registrations which can be seen in table 6.4. The invalid registrations are shown as zeros in this figure. The figure shows that slacking the transformation epsilon somewhat gives worse registrations. The GPU versions in (b) show less effect on this value.

	Transformation epsilon				
	7.8125e-05	0.00015625	0.000625	0.0025	0.01
icp voxel gpu, nearest point, dual quaternion	1	1	1	15	15
icp voxel, nearest point, point to plane lls	0	0	0	0	0
icp gpu, nearest point, dual quaternion	1	1	1	1	1
icp voxel gpu, nearest point, point to plane lls	0	13	15	0	0

Table 6.4: Table shows the number of valid registrations performed for different variations of the ICP algorithms shown in figure 6.12 and 6.13. As the test set contains 15 images, it’s desirable to have 15 in all cells in the table. This means that the registration parameters were suboptimal for the algorithms that show fewer than 15 successful registrations, and they failed after zero or a few images. Read this together with figures 6.12 and 6.13. Note that the algorithms that performed all 15 registrations correctly are left out of this table.



(a) Slowest ICP variations (b) Fastest ICP variations

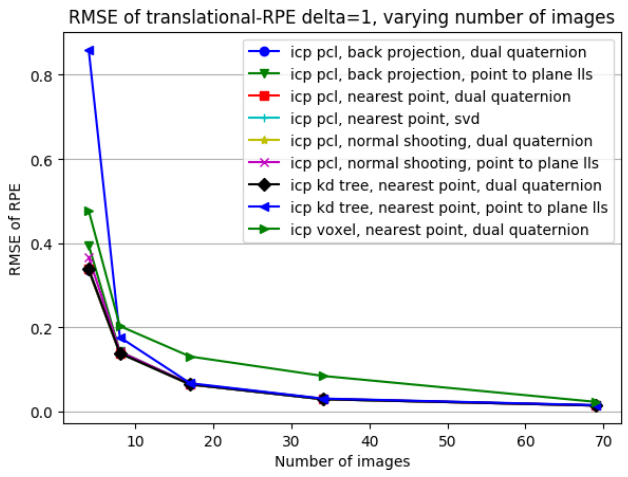
Figure 6.14: Figure showing the total running time of different variations of the ICP algorithms run on the fruits dataset when varying number of images. The slowest algorithms are shown in (a) and the fastest in (b), and time is measured in seconds. The figure show that performance is increased by using more images, but due to a worse initial alignment, the correspondence estimation step takes slightly longer time per image compared when using fewer images.

6.4.5 Varying number of images

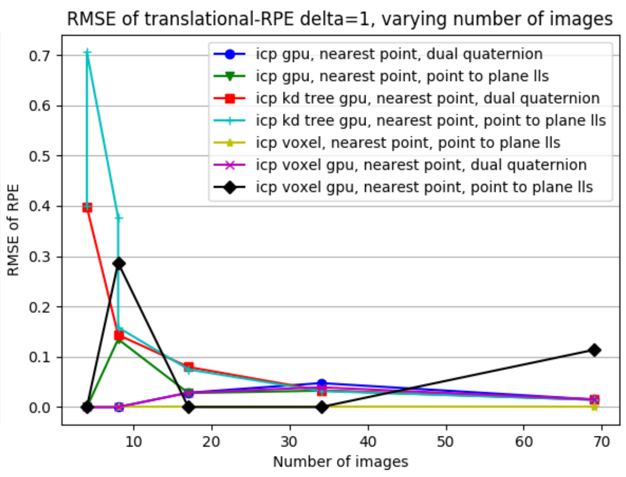
This section will show evaluation metrics when varying the number of images used in the registration based on the method described in section 5.1.4.

The running time of the different ICP variations when varying the number of images in the registration are shown in figure 6.15. It's clear that the number of images greatly affects the running time, but some of the variations flattens flattens slightly. Note that it's really the frame rate that changes, meaning that more images gives a smaller step for the robot arm between consecutive images. Because of this, the initial alignment between consecutive images better which reduces the search time slightly.

Again, the performance is presented together with the error metrics. The error measures related to the figure 6.14, the RMSE of the translational RPE with $\Delta = 1$ is shown in figure 6.15. The problem with this measure is that it shows the RMSE of the RPE error between each frame, meaning that using more frames is expected to give less drift per image. Therefore the absolute error (as described in section 5.1.3) is shown in figure 6.16. As the metric averages over all Δ it takes all steps in to consideration (but is still rewarding the small steps slightly more than large steps). Because of this, the absolute error flattens out when using enough images. Based on figure 6.16 even using 8 images gives quite good results, and going going above 17 generally seems to be just as good as more images for most variations. The only difference is the voxelized



(a) Slowest ICP variations



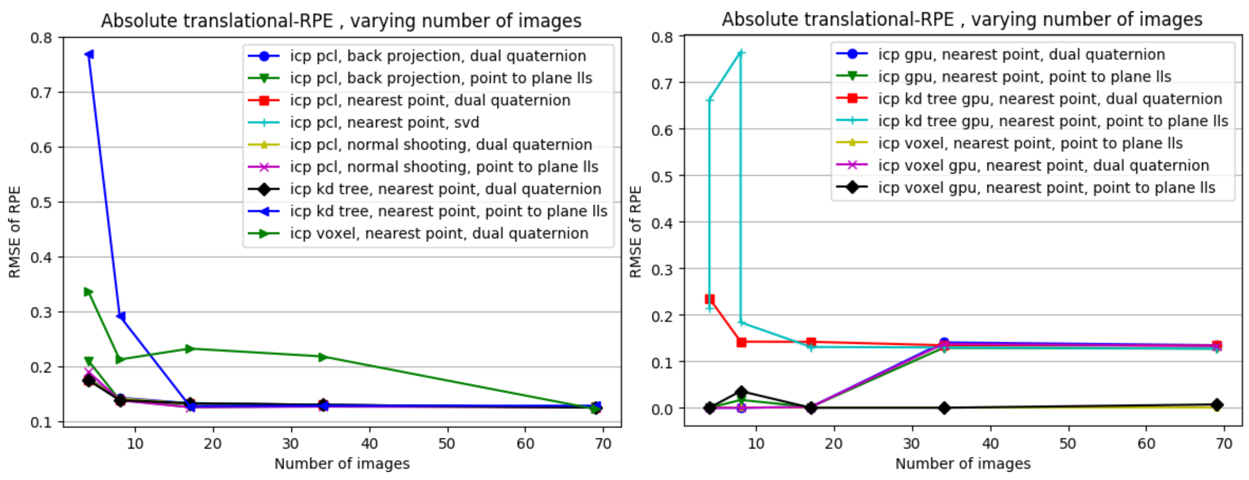
(b) Fastest ICP variations

Figure 6.15: Figure showing the RMSE of the translational RPE error with $\Delta = 1$ of different variations of the ICP algorithm when varying number of images. The test was run on the "fruits" test set. The slowest variations are shown in (a) while the fastest are shown in (b). As some of the registrations were unsuccessful, table 6.5 shows the number of valid registrations for the different number of images tried in this figure. The table is used together with this figure to evaluate the error of the registrations. The figure show that the error per image goes down as more images are added. This is expected, and a bit hard to interpret, so see figure 6.16 for the absolute errors.

methods which generally is unable to get a valid registration using less than 34 images.

6.5 Scan method results

This section shows the results from the different scan methods as described in section 5.1.4. The table is used to compare the different ICP variations on the test sets. It's hard to compare the actual methods to each other as they have a different number of images and distance between them. Therefore, figure 6.17 is included to give a visual comparison of the scan methods and the point to plane lls transformation estimator. In this figure, it's only the triangle and translation scans that are completely off. The zoom scan also have done a single mistake at the end. However, the triangle scan has shown better results in figure 6.3, so this was probably an unlucky single event for the triangle scan method. The same could be said for the translation scan, but during the project, there has been little luck getting good registrations with the translational scan method. One example of one such trial is shown in section 6.9, but is generally a hard task without a good initial alignment. The long-swipe, short-swipe and zoom move scan has relatively good registrations apart from the wrong alignment in the scan start.



(a) Slowest ICP variations (b) Fastest ICP variations

Figure 6.16: Figure showing the absolute translational RPE error as described in section 5.1.3 of different variations of the ICP algorithm when varying number of images. The test is run on the ”fruits” dataset. The slowest variations are shown in (a) while the fastest are shown in (b). As some of the registrations were unsuccessful, table 6.5 shows the number of valid registrations for the different number of images tried in this figure. The table is used together with this figure to evaluate the error of the registrations. This figure shows that the total absolute error is more or less unaffected by the number of images used. This indicates that the algorithm is able to perform all registrations correctly, even with low frame rates.

	Number of images				
	4.0	8.0	17.0	34.0	69.0
icp gpu, nearest point, point to plane lls	0	1	1	32	66
icp gpu, nearest point, dual quaternion	0	0	1	32	66
icp voxel gpu, nearest point, dual quaternion	0	0	1	32	66
icp voxel, nearest point, point to plane lls	0	0	0	0	0
icp voxel gpu, nearest point, point to plane lls	0	1	0	0	1

Table 6.5: Table shows the number of valid registrations performed for different variations of the ICP algorithms shown in figures 6.14, 6.15 and 6.16. As the test set contains 15 images, so it’s desired to have 15 in all cells in the table. This means that the registration parameters were suboptimal for the algorithms that show fewer than 15 successful registrations, and they failed after zero or a few images. Read this together with figures 6.14, 6.15 and 6.16. Note that the algorithms that performed all 15 registrations correctly are left out of this table. This table corresponds to figures 6.15 and 6.16.

	Scan method						
	cross	l.swipe	s.swipe	trans	triang.	zoom	z. m.
icp pcl, dual quaternion	0.076	0.024	0.013	0.110	0.076	0.071	0.044
icp pcl, point to plane lls	0.049	0.024	0.012	0.081	0.073	0.067	0.037
icp kd tree gpu, dual quaternion	0.065	0.035	0.014	0.090	0.090	0.066	0.040
icp kd tree gpu, point to plane lls	0.055	0.037	0.013	0.142	0.109	0.151	0.038
icp voxel, dual quaternion	0.139	0.026	0.017	0.110	0.366	0.217	0.095
icp voxel, point to plane lls	0.000	0.000	0.000	0.000	0.000	0.000	0.058
icp voxel gpu, dual quaternion	0.215	0.136	0.205	0.157	0.302	0.000	0.000
icp voxel gpu, point to plane lls	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Table 6.6: Table showing the translational RPE with $\Delta = 1$ for the different scan methodologies described in section 5.1.4 and different variations of the ICP algorithm. Note that some values are zero, which means that the registration failed. The table again show that the point-to-plane lls transformation estimator is the most accurate for most test sets. The dual quaternion method comes in between and the voxelized methods are worst. One additional thing to note is that the GPU implementations perform slightly worse than the CPU implementations. The table also shows that the tranlational, triangle and zoom scans are harder than the others. Figure 6.1 show that the triangular scan is usable, so again, this could be caused by an unfortunate setup that favored certain scan methods. See figure 6.17 for corresponding trajectories.

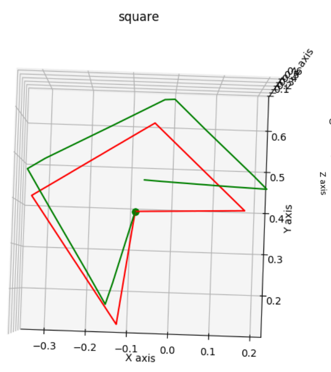
Considering all this data, there is little indication that any scan method is better than any other. If a complete view of the object is required, the square, triangle or translation scan is required. The triangle scan is slightly faster than square scan, and the translational scan is not recommended if it can be avoided. The long-swipe and short swipe scans are good for a quick scan if not an entire view of the object is required. The zoom scans (they are moving the camera closer to the object, and is not actually zooming) show that the camera can be moved closer and further away from the object if more or less detail is desired.

6.6 kD-tree search stack size

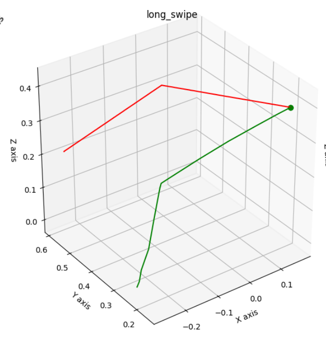
One important question regarding the kD-tree search is the size of the fixed size stack. This section will show the results generated from the method described in section 5.1.4, which includes performance and error measures comparisons.

6.6.1 Performance

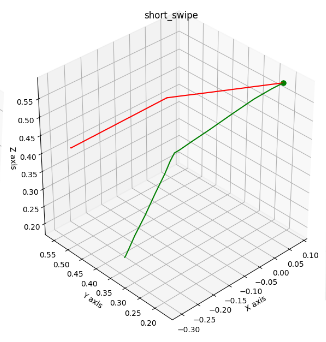
The running time of the kD-tree search is shown in figure 6.18 where the stack size is varied. The figure is based on the GPU implementation, but similar results are expected for the CPU version. The corresponding error measures are shown in figure 6.19. These figures show what is expected, which is that the registration takes more time when the stack size is increased, and



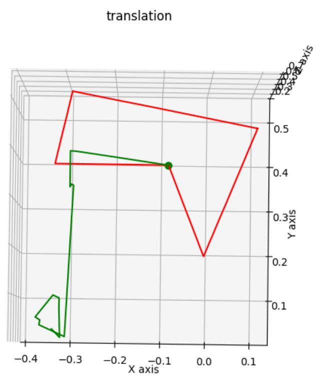
(a) Square



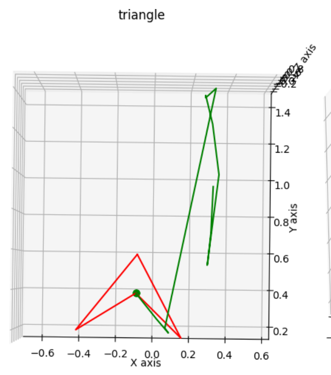
(b) Long swipe



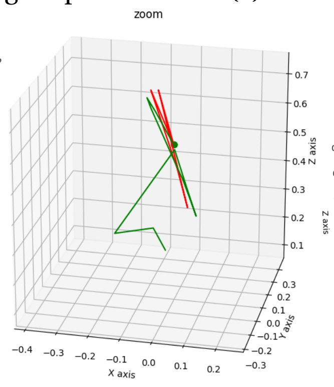
(c) Short swipe



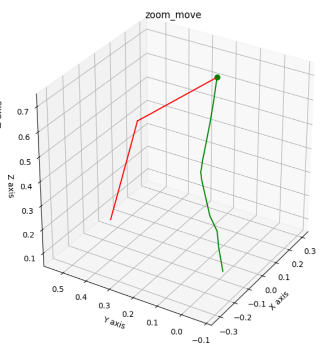
(d) Translation



(e) Triangle



(f) Zoom



(g) Zoom move

Figure 6.17: Figure is showing the estimated trajectory (green) for the point to plane lls transformation estimator and the reference robot trajectory (red) together for different scan methods. Most figures shows correct results, while (d), (e), (f) shows some errors. The (e) and (f) set was probably due to an ”unlucky” registration. This can happen because the parameters that work for one type of algorithm might not work for another. The translation scan is generally hard as the transformation estimator tries to align the underlying plane with maximum overlap instead of maximizing the overlap of the objects themselves.

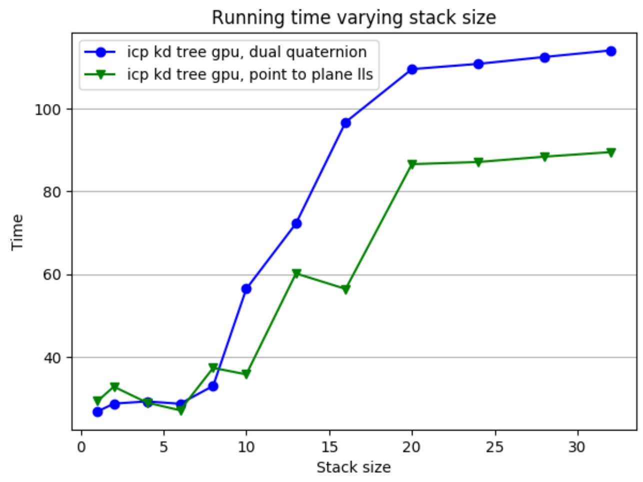


Figure 6.18: Figure showing the running times of the GPU implementation of the ICP algorithm using the kD-tree search when varying stack size. The test set is the ”banana” test set with a total of 16 images giving a frame rate of 1.25 FPS. As expected, increasing the stack size gives a slower algorithm. However, when using a stack larger than the depth of the tree (about stack size 20), it makes little difference on performance.

that the error is reduced as the stack size is increased.

To show the convergence of the algorithms, figure 6.20 was included. However, looking at the figure, the fitness value seems to increase as more iterations are performed. While some of the large values are due to erroneous registrations, which can be compared with figure 6.19. The reason for these results are discussed further in section 7.8.1.

6.7 Voxel search variations

As described in section 4.4.5, a few suggested improvements to the voxelized search correspondence estimator which is the basis for the method described in section 4.4.5. This section will also do a comparison of the voxelized search method and using a voxel grid filter and perform a standard search.

Figure 6.21 shows the running time of different voxel sizes and variations of the voxel search algorithm. The running times of a standard kD-tree search on the GPU usng a voxel-grid filter is shown in figure 6.22.

As with the previous sections, the performance is presented with the some error evaluation metrics in figure 6.23, and figure 6.24 for the voxel grid filter alternative. The voxelized algorithm

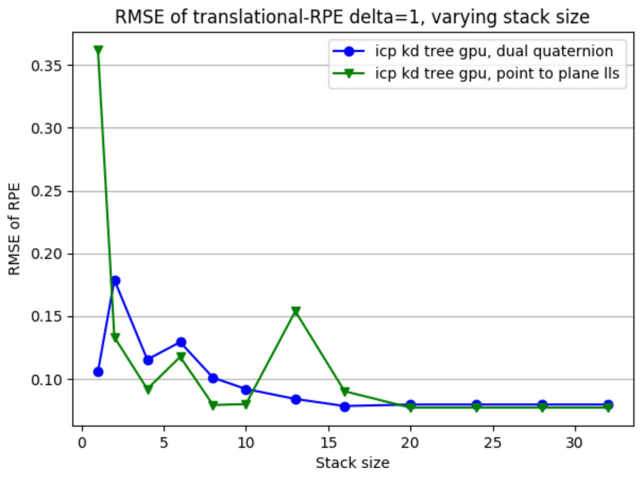
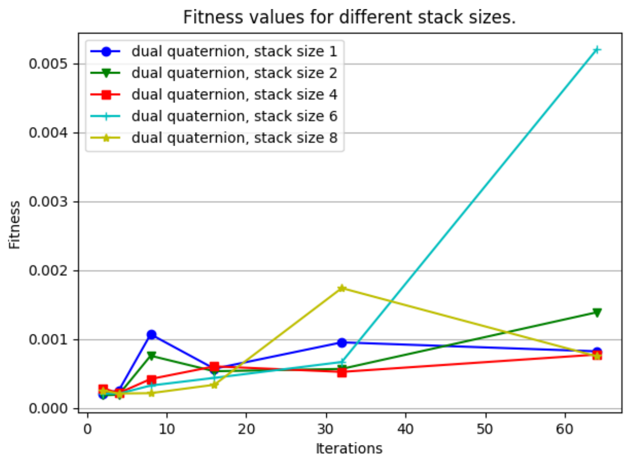


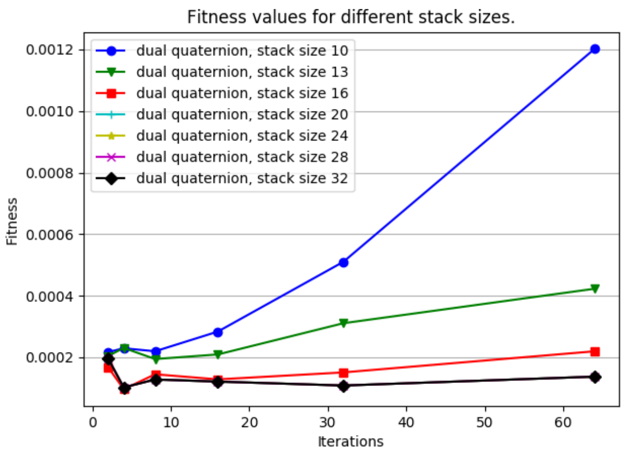
Figure 6.19: Figure showing the RMSE of the RPE with $\Delta = 1$ for the GPU implementation of the ICP algorithm when varying the stack size. The test set is the ”banana” dataset with a total of 16 images. The figure shows that increasing stack size improves the registration quality, though it can be acceptable in some cases to use a small stack size (stack size of 10, shows to be close the the best registrations obtained). As the stack size is increased above 20, no precision is gained because the stack size is as large as the depth of the kD-tree.

	Fitness epsilon							
	0.002	0.003	0.005	0.007	0.01	0.03	0.05	0.1
icp voxel, pre-search range 2.0	7	20	32	-	32	32	-	-
icp voxel gpu	23	32	32	9	32	32	32	32
icp voxel	32	32	32	32	32	32	32	32
icp voxel gpu, pre-search range 2.0	0	1	32	-	32	32	-	-
icp voxel, pre-search range 8.0	0	1	6	-	32	32	-	-
icp voxel gpu, pre-search range 8.0	0	2	32	-	32	32	-	-
icp voxel, pre-search range 4.0	2	7	26	-	32	32	-	-
icp voxel gpu, pre-search range 4.0	0	1	32	-	32	32	-	-

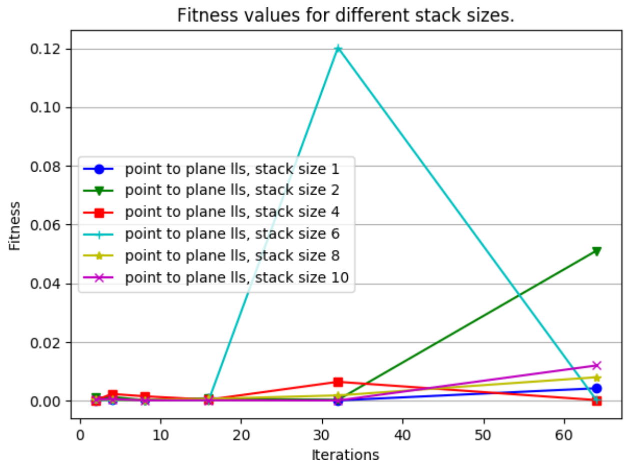
Table 6.7: Table showing the number of registered images for the running times and error values shown in figure 6.21 and 6.23. All sets contain 32 images, and any lower values indicate that the registration failed. Registrations also failed where no counts are specified. As it’s desired to complete all 32, any lower values in the cell indicate that the registration was more or less unsuccessful.



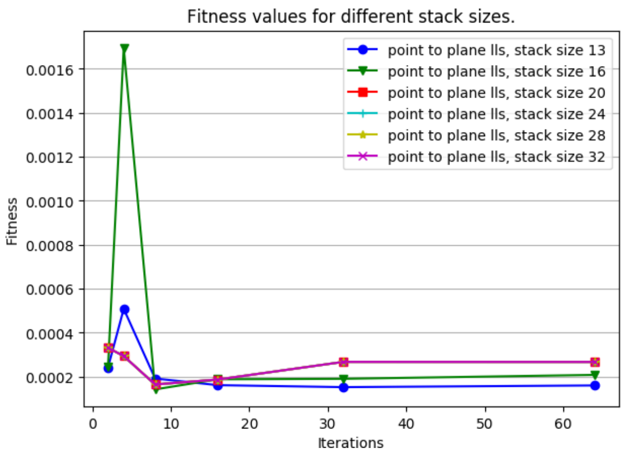
(a) Dual quaternion



(b) Dual quaternion

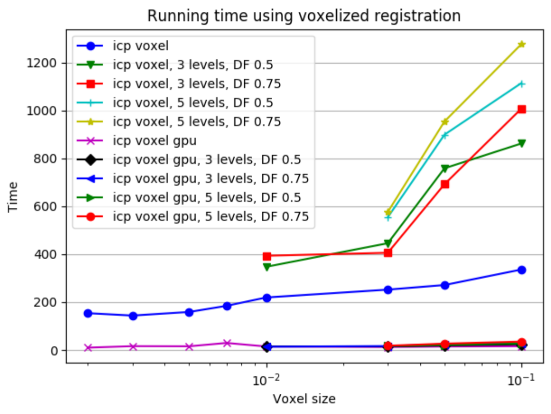


(c) Point to plane lls

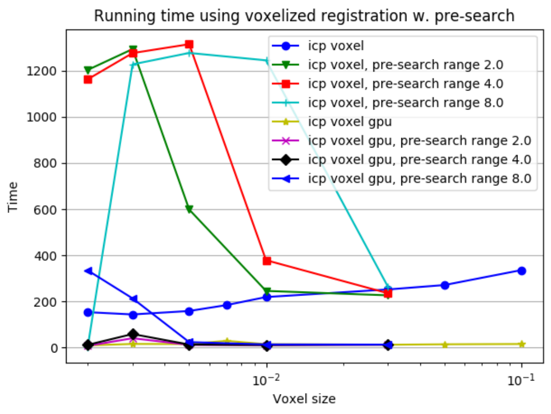


(d) Point to plane lls

Figure 6.20: Figure showing the fitness values when varying the stack size in the GPU implementation of the ICP algorithm with both the dual quaternion and point to plane lls transformation estimators. The fitness values are defined in section 5.1.3. The figure show that using more iterations give a worse fitness. This is not a good sign, but the fitness measures the closest point error, which might not be the correct thing to do. Therefore, the desired property is to see that it stabilizes at some point, which is the case when using larger stack sizes in (b) and (d). (a) and (c) show that the convergence is less stable when not using a precise nearest neighbour search.



(a) Multilevel



(b) With pre-search

Figure 6.21: Figure showing the running times of different variations of the voxelized ICP algorithm varying voxel size, multi-resolution levels and pre-search radius. The test is run on the ”fruits” dataset using a total of 32 images. (a) shows multiple variations of using multi-resolution search while (b) shows multiple variations of the pre-search strategy, both explained in section 4.4.5. In (a), ”DF” is a shortening of dynamic factor, which is the number that is multiplied with to get the voxel size of the next level (as described in section 4.4.5).

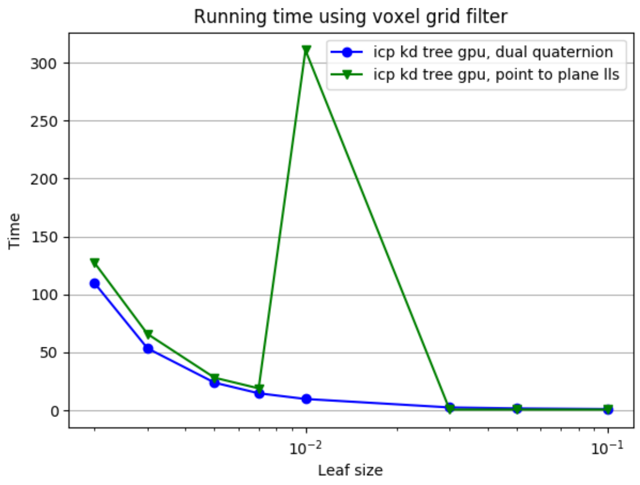
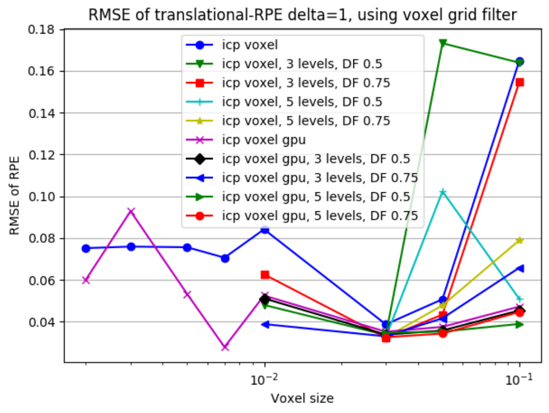
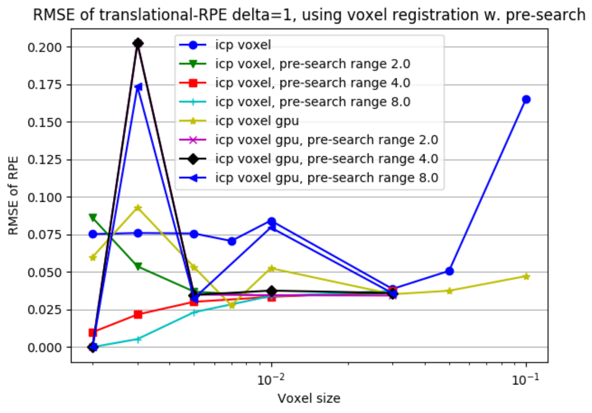


Figure 6.22: Figure showing the total running time of the kD-tree ICP GPU algorithms run on the fruits dataset with 32 images when voxel-grid filter leaf size (voxel size). Because the normals radius was constantly kept at 5 mm, the point-to-plane lls transformation estimator failed when the voxel size was any larger than this. The figure shows that performance can significantly be increased by lowering the resolution of the input cloud. Smaller voxel sizes are performing better on the GPU because the GPU implementation also search for the closest point in neighbouring voxels (thereby increasing search range slightly).



(a) Multilevel



(b) With pre-search

Figure 6.23: Figure showing the RMSE of RPE with $\Delta = 1$ for different variations of the voxelized ICP algorithm. In (a), the ”DF” means dynamic factor, which is the number the voxel size is multiplied by for each level. Note that the figure contains zero values. This indicates that the registration was invalid. Refer to table 6.7 for which registrations was valid. The figure shows that there is an optimal voxel size for the different variations. Note that the multi-resolution variations are plotted at their starting resolution, and that the pre-search values are plotted from the base resolution. The pre-search range denotes the pre-search radius in number of voxels. The figure shows that there is a sweet spot for the voxel size to get good registration results.

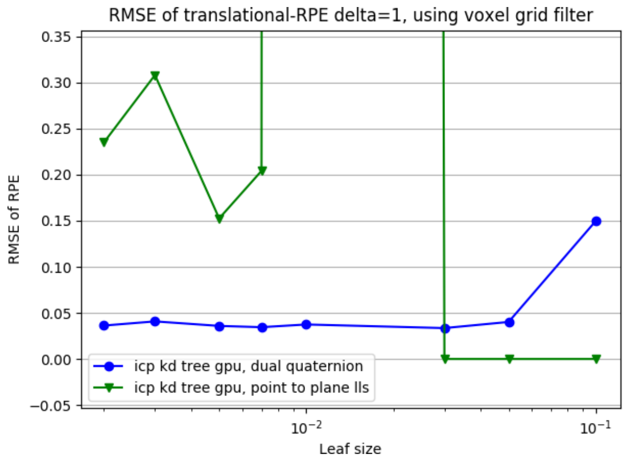
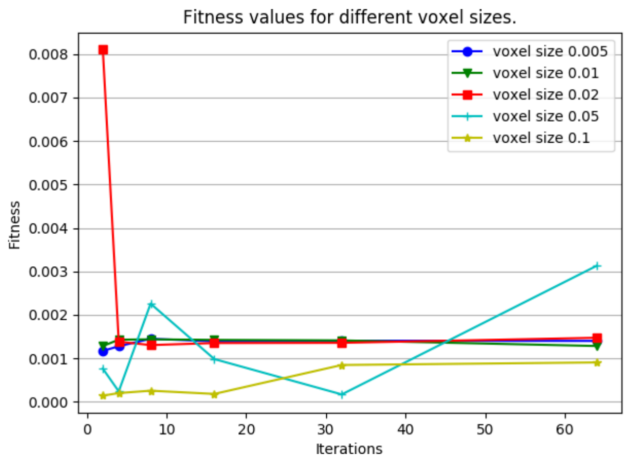
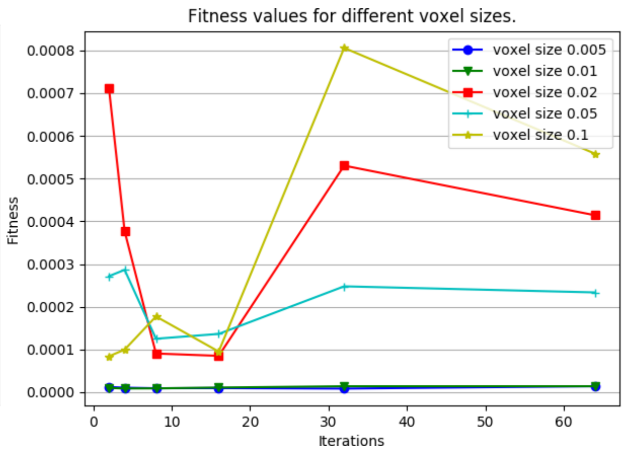


Figure 6.24: Figure showing the RMSE of RPE with $\Delta = 1$ for two standard GPU implementations of the ICP algorithm when varying the voxel size. Note that the figure contains zero-values, indicating that the registration was invalid. The figure show that the registration error is not affected much by the leaf size. Also note that the point to plane lls transformation estimation was for some reason unable to perform the registrations here.



(a) Voxelized ICP on CPU



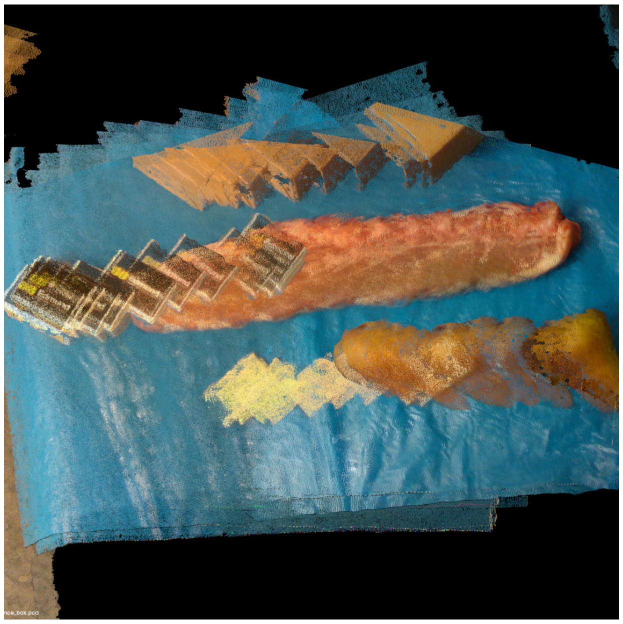
(b) Voxelized ICP on GPU

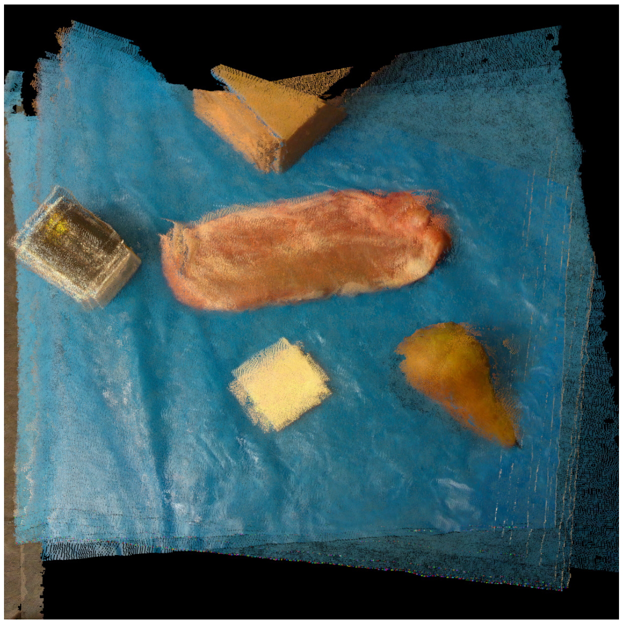
Figure 6.25: Figure showing the fitness values when varying the stack size in the voxelized ICP algorithm. The fitness values are defined in section 5.1.3. The figure show that using more iterations give a quite stable fitness. This is not a good sign, but the fitness measures the closest point error, which might not be the correct thing to do. Therefore, the desired property is to see that it stabilizes at some point, which is the case for some of the voxel sizes in (a). (b) show less stable behaviour, which is probably caused by the race condition explained in section 4.4.6.

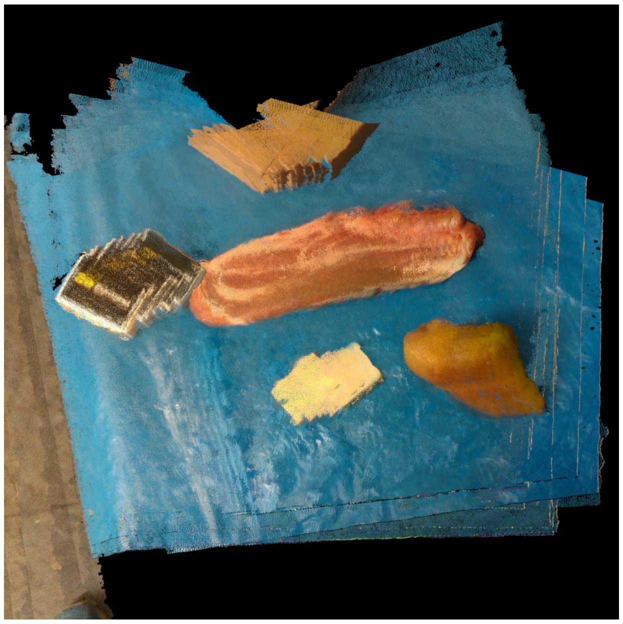
Finally, the fitness of the voxelized algorithm is presented in figure 6.25. As with the fitness of the stack size in section 6.6, the algorithms converge to some error fitness value for a good voxel size.

6.8 Correspondence rejection

This section is included to give a qualitative evaluation of the box correspondence rejector described in section 4.5. In general, this section is included to show the importance of boundary rejection strategies in general. Figure 6.26 shows some examples of registrations done with and without the box correspondence rejector. The reason it's done like this is that the previous used error metrics will not necessarily show the quality of these images. There is only minor differences between figure 6.26c and 6.26d that would not show on the RPE based methods. The fitness score also a bad idea as it probably is better for figure 6.26a the others because it is the only method that directly optimizes the fitness score. Figure 6.26 shows that optimizing the global fitness doesn't necessarily give the best registration. The boundary rejection strategy enables the algorithm to only optimize the important part of the point cloud instead of the global, which might converge to a local minimum.


(a) No correspondence rejection
Dual quaternion


(b) With correspondence rejection
Dual quaternion


(c) No correspondence rejection
Point to plane lls

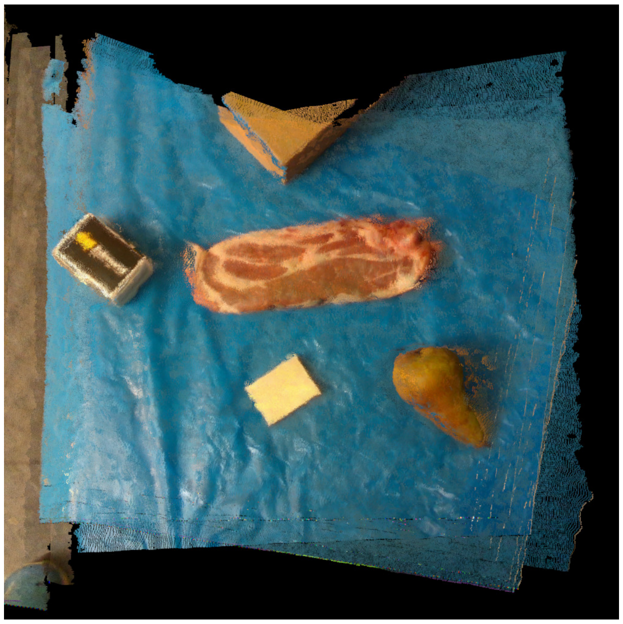

(d) With correspondence rejection
Point to plane lls

Figure 6.26: Figure showing four registrations of the pork chop dataset using the dual quaternion and point to plane lls transformation estimators with and without a correspondence box. It's clear that using the correspondence box in (b) and (d) greatly improves the registration quality. It also shows that the point-to-plane lls transformation estimator in (c) and (d) gives better registration quality than the dual quaternion approach in (a) and (b) for similar conditions and arguments. (b) has converged to the correct minima, but the dual quaternion approach simply requires a lot more iterations than the point to plane lls transformation estimator.

6.9 Registration of translation scans

Based on the results in section 6.5, the translational scan seems like a bad idea. This section will show the results of that section, explaining how a translational scan might be performed.

Figure 6.27 shows how application of filters can aid the registration. Though not perfect, figure 6.27b shows that the filtering makes a better registration than the unfiltered alternative.

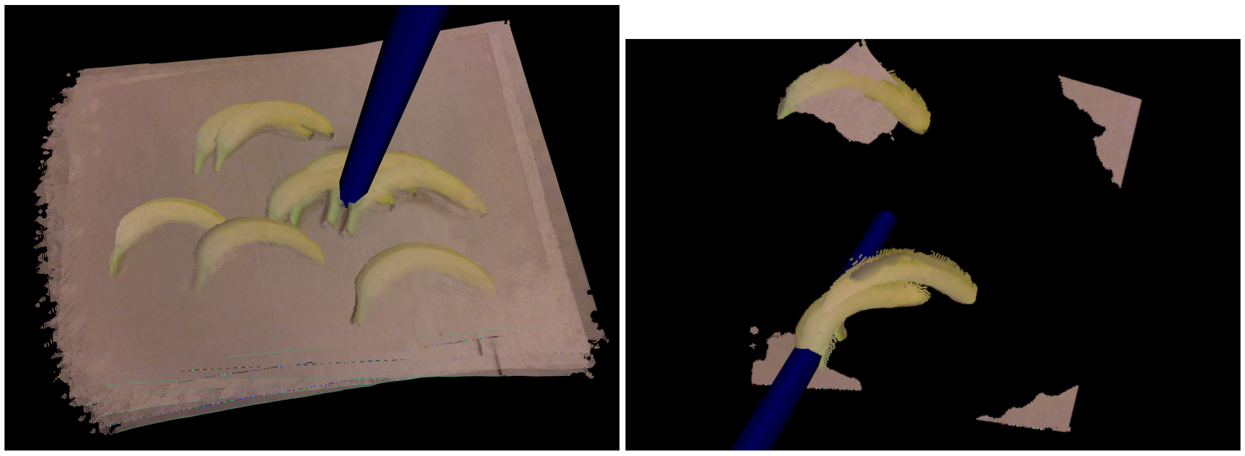
The reason the unfiltered variant looks is unable to register the banana correctly is because of the way the optimization stage in the ICP algorithm works. The problem is that if the algorithm were to align the banana, a large portion of the underlying plane would be shifted outside of the underlying plane of the reference point cloud. This in turn would give a significant distance to the closest point, and the optimizer would shift the model cloud such that it's better aligned with the reference cloud. Because the banana is so small as it is, the correspondences of the banana is simply not large enough compared to the much larger plane.

Another thing that makes this particular dataset hard is that the underlying plane is not perfectly flat. This is caused by some depth error produced by the camera itself, but the surface itself is not very flat. Using a plane model segmentation filter would require to set a large distance threshold, that would remove large portions of the banana itself as well. If it was possible to get the underlying plane to be more flat, the translational scan would be easier to register because of this.

Other options could be to filter the point cloud based on color. By first using a color segmentation on the banana, it could be possible to extract it from the plane. However, the plane in this project has a quite similar color to the objects themselves. However, in a robotics application, one option could for example be to color the plane red (given that it only work with bananas), that would make color based filtering easier. As discussed in section 3.1.1, another point selection strategy could also help with this type of registration.

6.10 Real time registration video

As additional results, a video is provided with this report to show the GPU implementations can perform registrations in real-time. The video shows that the framework is able to do real-time registrations as the robot moves. It also shows how a voxel grid filter is used to increase the performance significantly. Most of the clips use the ICP GPU implementation using a kD-tree search, and one clip shows the voxelized ICP implementation on GPU.



(a) Unfiltered (b) Filtered

Figure 6.27: Figure is showing the same input data and registration parameters with different filtering. (a) is unfiltered, while (b) has applied a box filter, multiple plane model segmentation filters and an euclidean cluster filter. The figure shows that filtering somewhat can enable hard scans, but due to certain properties of the scan, the banana is not perfectly aligned.

6.11 Other results

This section contains other small results that are doesn't directly fit into the other categories. The following list is a collecion of a few timings that is not included in the other results sections:

CPU Normal computation time This takes about 25.75 seconds on 35 images, which is 0.736 seconds per image. The timings for normal computation is not included in a figure because it shows an almost constant time per image for a given point cloud. This observation is done by analyzing the program logs.

GPU Normal computation time This takes about 1.752 seconds on 35 images, which is 0.050 seconds per image. The timings for normal computation is not included in a figure because it shows an almost constant time per image for a given point cloud. This observation is done by analyzing the program logs.

Voxel filter computation time Applying a double voxel grid filter (two times in a row with larger voxel size the second round) 0.624 seconds on 35 images. This is 0.018 seconds per image. As with the normal computations, this is more or less the same for all voxel sizes for a given point cloud and is therefore not included in a figure. This observation is done by analyzing the program logs.

Most of these times are negligible compared to the registration logs. They should be considered in a real-time application and is used in the discussions in chapter 7.

CHAPTER 7

Discussion

This section will discuss the results and try to draw conclusions based on results and experimental setup. It will also compare the results from this work to some of the related work from section 3. The section starts with a general overview of the accomplishment of the project and will in later chapters more details and sources of errors.

7.1 Registration quality and performance

This section will be general discussion about the overall accomplishment of this project which is the registration quality and performance.

7.1.1 Registration quality

Figures 6.1 and 6.2 show that the accumulated point clouds generated from the registration algorithms are good. As both of these are run using the GPU version of the ICP algorithm (using kD-tree search and point to plane lls transformation estimation on GPU) it verifies that the GPU implementation does in fact work. As additional proof of this, the estimated error measures in table 6.1 show that the relative pose error is relatively small. Figure 6.3 show that this error makes sense, and that the registration is relatively good. The video also show that the registration can be performed while the robot operates and give a good registration model. The registration performed in the video work on full resolution point clouds. The main takeaway here is that the accumulated point clouds are sufficiently good for a robot scanner application, and a robot is expected to be able to interact with the objects based on what is shown in figure 6.1.

7.1.2 Registration performance

It can discussed whether this is actually real-time or not. It is able to operate on the fly, but real-time is often defined as algorithms running with at least 20-30 fps. However, given results from section 6.4.5 show that using a relatively low frame-rate still gives acceptable registration results for the robot-mounted camera application. Considering that the robot moves around the object it scans, it is creating a view of the model, and will require to use some time obtaining a complete view of the object. This give the algorithm enough time to register a few images from multiple views of the object for a complete object reconstruction.

Considering the robot scanner application, the registration can be considered real-time. Given the results from figure 6.22 and 6.24, it is clear that the running times are significantly reduced while still keeping a relatively good registration quality. This is also shown in the attached video. Similarily, figure 6.18 and 6.19 show that the stack size can be lowered significantly without drastic loss of precision.

Figure 6.8 show that the implemented kD-tree search for this project is about four times slower than the library implementation in PCL. Again comparing with the GPU implementation of the same algorithm, it is about 30 times slower for the same kD-tree implementation, showing the great speedup potential. The GPU implementation is about 8 times faster than the PCL library implementation. In this project, only a few optimization tricks were performed, and there are probably a lot more optimizations that could be implemented both on the GPU and kD-tree search algorithm. This could potentially get a speedup of approximately 30 times the library PCL implementations. All of these comparisons are done using the timings from 128 iterations and a stack size of 20.

Another thing to note is that the stack size can be tuned to give a good trade-off between performance and precision. Compared to the search method used by Qiu et al. (2009) the method presented in this thesis seems to perform slightly worse when using smaller stack sizes. This is probably due to the priority based queue used in Qiu et al. (2009), which keeps nodes higher up in the tree if it has a shorter distance than the other ones further down in the tree. This is causing the precision increase. However, when comparing the 30x speedup observed here, it is not that far from the implementation by Qiu et al. (2009), but given the use of a smaller stack size, a speedup of twice that can be realistic, and is not that far from the GPU implementation by Qiu et al. (2009) (though this is not guaranteed will be observed if changing the search algorithm). For a future implementation, the priority queue search method is advised before the standard kD-tree implementation done for this project.

7.1.3 Free-hand scans

While discussion in section 7.1.1 and 7.1.2 is based on the robot scans, the registration quality of the free-hand registrations will briefly be discussed here.

As there is no reference scan, the free-hand scans shown in figure 6.2 are simply evaluated by looking at the registration result and verifying that the trajectory shown in figure 6.6 looks like the actual trajectory (which is the case). As all these figures look sensible, the conclusion is that the registrations of the free-hand scans were successful using parameters similar to that of the robot based scans. The goal of the free-hand scans was to show that it was possible to do, and the results fro figure 6.1, show that this goal is achieved.

7.1.4 Wall time

The wall time measure was used and showed stable behaviour. Though not shown in the reference section, multiple runs with the same algorithm was performed, which gave similar results. As the real application is real-time registration, the computer the registrations was run on is similar to something that could be used for a robot, and wall time is a measure that gives an idea of the actual performance. No strange timings was observed due to the interference with other programs, and considering the long running times of the program, such processes would probably not have visible impact on the timings. Therefore, the wall time is considered a trustworthy measure given the experimental setup.

7.2 Voxelized ICP variations

This section is separated from section 7.1 mainly because the results was not that comparable to the other algorithms, but also because it contains many variations that require some discussion in themselves.

7.2.1 Voxelized ICP algorithm

This section is about the standard voxelized version of the ICP algorithm. First of all, figures 6.14 shows that the CPU implementation of the algorithm is the fastest of the non-GPU variations, but the performance is not impressive when looking at the error in figure 6.15. The algorithm generally seems to give worse registration results than the point based algorithms. This trend is also seen through sections 6.4.2, 6.4.3 and 6.4.4.

Note that the voxelized ICP algorithm works quite differently from the other evaluated registration methods. This means that the registration might be slightly different from the other algorithms. Sections 6.4.2, 6.4.3 and 6.4.4 have parameters that are setup to be used with a point-to-point based registration, which can have penalized the voxelized variations. However, section 6.7 have more voxel-friendly parameters, and slightly less pessimistic results for the voxelized variations.

Comparing the algorithm to the work by Marden and Guivant (2012), the algorithm performs a lot worse for the robot scanning application than the dataset used by Marden and Guivant (2012). The main differences is the dataset, and the fact that Marden and Guivant (2012) used the Inertial Measurement Unit (IMU) to estimate an initial pose. The last difference could be used for this project by using the estimated robot position. However, the dataset will remain different. The dataset used by Marden and Guivant (2012) was a room scan with relatively large and smooth objects. The voxelization loose the small features, and the alignment fails because of this.

A final note on this is the use of a voxel grid filter. Comparing figure 6.22 with 6.21 and 6.24 with 6.23, it shows that similar performance can be obtained by using a voxel grid filter and then applying the registration using a kD-tree based GPU implementation.

7.2.2 Voxel size

This section will briefly discuss the results for voxel size in the voxelized ICP variations. This is mainly observed in figures 6.21, 6.23. The important observation here is that the error and performance has an optimal voxel size given the registration quality. The same is the case for the performance, though it is a bit flatter for the GPU implementations. This section will not conclude any further as these parameters is highly dependent on the application and object sizes. The following two subsections will discuss the suggested improvements, though it's not convincing that they are better than the standard voxelized ICP.

7.2.3 Multilevel search

The results from figure 6.21a and 6.23a show that there is a relatively high cost to perform a multilevel search, especially on the CPU implementations. Figure 6.23a show that registration precision can be improved somewhat for more voxel sizes. Anyway, the basic voxelized strategy shows a just as good registration quality for a specific voxel size. The algorithm is probably converging anyway because the object is centered, and the distance from the camera to the object is more or less fixed. Therefore, the standard voxelized approach work well for this particular scan. The multi-resolution approach might be better if the camera does not center the object, or if the camera change distance to the object.

7.2.4 Pre-search method

The argumentation is the same as the previous section. The pre-search use quite a lot of time (though it is faster on the GPU). However, the registration quality generally seems to be improved using this technique by looking at figure 6.23b.

7.2.5 Race condition in Voxelized ICP

As mentioned in section 4.4.6, the GPU implementation of the voxel correspondence search method. This occurs because the algorithm is parallelized on the data cloud, and multiple points in the data cloud might be inserted into the same voxel, resulting in a race condition. As explained in section 4.4.6, the race condition is not going to cause any problems given that the voxelized cloud has a random-like distribution. This is however not the case when dealing with point clouds, and there was observed issues with the method. The observation was that the point cloud was slightly skewed for each iteration of the ICP algorithm. This is most likely caused by an unbalanced weighting of the elements within a voxel, meaning that the vector going from the center of the voxel to the computed centroid is not 0. This is a problem with this kind of voxelization in the first place because the average offset from the voxel center to the voxel centroid might not be 0. This was especially seen when using the GPU version because of the race condition. The race condition because multiple of the insertions where discarded, and because the centroid might not be valid in the first place. This was probably less of an issue in the work by Marden and Guivant (2012) because the room-scale features was able to counteract the voxel-skew effect. In the robot-scanner application, the plane allowed for skew as there was no larger features to ”hold” the data cloud at the right position. This effect will probably be less significant when using larger objects.

Based on many observations and results in section 6, the voxelized strategy proposed by Marden and Guivant (2012), seems to be a suboptimal solution for real-time registration even considering the very fast runtime. This does not mean that voxelization in general is a bad idea.

7.2.6 Comparing the Voxelized CPU and GPU implementations

Comparing the performance in figure 6.21, there is a big difference in running time between the GPU implementation and the CPU implementation. Because of the voxelized datastructure, the problem is highly parallelizeable when searching for correspondences, giving the tenfold speedup shown. This speedup is somewhat less than what is observed with the full resolution kD-tree search. This is probably due to the algorithm being faster in itself, giving

7.3 Dataset quality

This section will discuss why a new and custom dataset was used for this project compared to an existing dataset. It will also discuss the quality of the dataset.

7.3.1 Using a custom dataset

There are many standard datasets available for 3D registration applications (for example the ASL datasets (https://projects.asl.ethz.ch/datasets/doku.php?id=iros2017)). These were not used in this project because they generally capture larger scenes with large and clear features. For the robot scanning application these datasets are based on applications and considerations that are unrealistic for a robot scanner. A robot scanner is typically placed in a sterile environment, and for a general scanning algorithm to work for this type of application, it should not only work on room-scale datasets which is typically the case for these datasets.

Some of the related work like Magnusson (2013) also use simulated datasets. This gives an opportunity to get precice data for many examples. The same argument applies for existing datasets as mentioned above. Creating a new simulated dataset would be time consuming, and given that the final application would be to use a real-world camera, this was not done. A simulated dataset usually contain some unrealism that makes the transfer to real data less smooth.

7.3.2 The robot based datsets

The new test set created with the robot arm is considered a realistic setup. The robot arm is placed in a sterile environment with only the object as a distinct feature. The camera is also a standard camera that easily could be used in a real-world robot application.

Figure 6.5, the robot trajectory seems to move as expected. Given the control method described in 4.1 the linear accelerating movement between waypoints is as expected. Looking at the video, and the actual movement of the robot, this also seems to match. The figure generated in figure 6.5 is generated from the internal position estimate of the robot, but there is still room for errors as there is no external calibration system to verify this, meaning that the trajectory could be wrong.

Seeing figure 6.3 and table 6.1, it looks like the estimated trajectory is relatively close to the robot trajectory. Considering the results shown in figure 6.1 the final registration looks relatively good, meaning that the estimated trajectories are probably quite good. Given the accuracy of the registrations and the path, most of the error of these figures are caused by a wrongly estimated camera path. A slightly different measured angle might be hard to see in the accumulated point cloud, but can cause large drifts relative to the reference path when far away from the object.

All in all, the robot based test-sets are good enough to give a somewhat quantitative measure of how good a registration is. Due to some calibration differences with the camera and robot arm, some offset is expected, even for a perfect registration. The robot arm trajectory is the one that is most likely the correct path and can be used to evaluate the quality of a registration.

7.3.3 Free-hand datasets

The free-hand datasets does not contain any reference path, so it is hard to tell any quantitative measures on how well a registration has performed. A reference path could be generated by using an external tracking system of some kind, but because the robot paths give a reference path, the free-hand scans are included to give a qualitative evaluation of the registration algorithms in a different setting than the robot scanner. The objects that are scanned are somewhat limited, but is based on what is considered for the project (as introduced in section 1.2.1).

7.4 Robot arm movement

This section is included to give a brief discussion of the method described in section 4.1. The basis for this analysis is figure 6.5. Robot movement method described obviously has a few drawbacks, especially the quaternion to euler-angle-velocity conversion and robot velocity control. Looking at figure 6.5 the described problems doesn't seem to be a big issue for the scanning, and the trajectory looks as expected based on the set waypoints. Another thing to note is that start and ending point is the same for figures 6.5a, 6.5b and 6.5e which indicates that either the trajectory is very precise, or that it's able to catch up with the errors (which the controller is designed to do). The main argument for the trajectories is that it really doesn't matter because the values shown in figure 6.5 is taken from the robot's estimated position and not from the wanted path. As long as the trajectory is known, the actual correctness of the path is irrelevant (more about comparison of the estimated robot trajectory and camera trajectory in section 7.3.2).

7.5 Evaluation metrics

Most of the related work (section 3) use the fitness epsilon to evaluate convergence. Other approaches (such as Kinect fusion Newcombe et al. (2011)), does a trajectory estimation, but as the scanner is free-hand it is not compared to a reference model. By using a robot arm, this project has compared the registration algorithms from an external viewpoint, which is concerned with how well the registration is based on the transformation it computes.

In figure 6.20, this project has shown some of the challenges by using the traditional fitness value. By comparing with figure 6.25, it shows how hard it can be to compare two different algorithms using the fitness value. Especially when the correct transformation may not be at the minima. This is because of the partial overlaps that the optimizer and fitness metric might overlook. If the fitness computation also considered the point correspondence rejections, it would in this project show that the convergence would be continuously decreasing, but this is not necessarily the case, and the other problems with the fitness epsilon is present when comparing different approaches.

7.6 Camera differences

As described in section 2.1, the two cameras used for this project (Intel RealSense D435 and SR300) are based on two slightly different technologies, which is clearly shown in figure 6.4. The observed distortion effect is most likely caused by some reflection or absorption. A similar effect was seen on the casted object (figure 6.1j) and around thin plates when looking on along the surface (looking down on the surface edge). The effect is therefore most likely caused by lacking parts of the projected pattern, or that the pattern is shifted in some way. The meat and casted objects might have different optical properties on the wavelength used by the infrared projector of the Intel RealSense SR300 camera, making the pattern different than what is expected. This effect could also be present for the Intel RealSense D435 camera, but as it is a stereo camera, it is still able to reconstruct the depth based purely on the image itself. Another thing to consider is to use a depth camera like the SR300 that use a different wavelength that is better for meat-tissue.

7.7 Point correspondence rejector

Figure 6.26 shows the impact of using a correspondence rejector. No quantitative measures was given to prove that this was an improvement, though this figure clearly shows the point. The ICP algorithm might get irrelevant correspondences in other parts of the cloud (for example the background when scanning the pork-chop). Therefore, the correspondence rejector is able to remove those irrelevant correspondences that does affect the transformation estimator.

7.8 ICP variations

There was a few ICP variations tested. Through the termination criteria tests, it showed that the point to plane lls transformation estimator generally was better than the others considering the number of iterations required. The dual quaternion seems to generally give more stable results (as a closed form solution is guaranteed). The point correspondence estimators showed that there was some difference in how they converged, but it was a less significant impact compared to the transformation estimator. This is based on sections 6.4.2, 6.4.3 and 6.4.4 and the related figures.

7.8.1 Stack size

Figures 6.18, 6.19, and 6.20 show that there is possible to trade precision for performance. Figure 6.18 shows that halving the stack size also gives approximately half the running time. Looking at the errors in figure 6.19, the error is significantly reduced (based on the transformation estimator). This section will not draw any more concluding remarks as the stack size depends on the application.

As a final note on the stack size, is that the dynamic queue used by Qiu et al. (2009) is a better strategy for a kD-tree implementation on the GPU. The dynamic queue prioritizes the currently closest found, which is more likely to give a closer neighbour than using the fixed size stack in this project.

7.9 Scan methods and object variations

The tested registrations methods used in this master thesis was applied on different scan methodologies and objects. The different scan methods showed that similar registration parameters could be used to perform registrations using multiple camera trajectories. The errors observed in figure 6.17 are a few exceptions where some algorithms can fail drastically if provided with the wrong arguments (and if they are ”unlucky”).

When it comes to the object scans, the results were quite surprising. Both the ”plastic” and apple test sets gave good registrations. The apple was expected to give erroneous registrations because of it's rotational symmetry. The ”plastic” object was expected to get a bad 3D reconstruction that made it hard to register. The reason they worked is probably due to the underlying floor (a cardboard floor) that was not completely flat.

CHAPTER 8

Conclusion and further work

In this chapter, based on the results from section 6, I draw some conclusions with regard to 3D registration methods shown in this thesis. In addition, I recommend some topics as a future work in connection to this interesting area of research in section 8.2.

8.1 Conclusion

The goal of this project was to implement and evaluate state of the art solutions to real-time point cloud registration for a camera mounted on a robot arm. This was broken down into subgoals which included implementation of a robot arm controller, test set creation, registration algorithm implementations and algorithm evaluation.

These goals were reached in this project which included a robot arm controller implementation and a 3D scanner capture program that captured RGB-D images while the robot was moving, or when used in free-hand scans. The robot controller was concluded to not be very elegant, but certainly good enough for evaluation of a robot based 3D scanner. The test sets created for this project was concluded to be relevant and realistic considering the environment the robot will work in.

When it comes to the implementations of the algorithms, all of the mentioned algorithms from section 1.2.2 were implemented and evaluated. The brute-force based algorithms was left out of the main results and discussion as they were a preliminary step for the kD-tree ICP and voxelized ICP implementations (they are always slower than the kD-tree search and not performing better). These algorithms were implemented in a registration framework developed for this project that works as a 3D scanner that can be used from the command line. This registration tool was both useful for exploring parameters for the registration and 3D scans as well as evaluation of the algorithms. The algorithms was evaluated using a python tool that computed the timings and RPE based error metrics based on the log files provided by the registration framework. The RPE based error metrics used in this project provided an external evaluation strategy of the registration algorithms that enabled different types of registration algorithms to be evaluated. This is different from the fitness metric (average point to point distance between point clouds) that commonly used when evaluating registration algorithms. The external RPE based evaluation strategy was enabled by the pose estimations obtained from the robot while the scans was performed.

The algorithms themselves included GPU implementations of two transformation estimators. These was based on a quaternion method and a point to plane linear least squares optimizers. Additionally, a voxelized correspondence estimation strategy was implemented together with a couple of suggested improvements. These improvements included a multi-resolution voxelized search and a pre-search strategy in the voxelized space. The final implementation done for the registration step itself was a point correspondence rejector that was based on specifying a box around the object of interest. The rejection step was based on rejection of correspondences that was outside of this box.

The point to plane linear least squares optimizer was shown to be the best transformation estimator used in this project. This estimator gave quick convergence, and correctly aligned the objects of interest. This was compared to the dual quaternion optimizer which was shown to be slightly faster and more stable per iteration, but was required a lot more iterations than the point to plane linear least squares optimizer. The different evaluated correspondence estimators was shown to have less impact on registration quality. Only the nearest point was implemented for the GPU, but the normal shooting and back projection estimators could be considered for future implementations as they seem to give slightly faster convergence in certain cases. The closest point estimator was shown to be good enough for the registrations performed in this project.

The GPU implementations was shown to be about 33 times faster than a similar CPU implementation and 8 times faster than similar library implementations. This without any loss of accuracy. By reducing the fixed stack size, even better performance can be achieved by some loss of accuracy. Additional speedup is expected to be possible through search and code optimizations. The voxelized search strategy was shown to generally give worse results than the other methods, but is the fastest algorithm tested. This also was also the case for the suggested improvements to the voxelized ICP algorithm. The increased speedup could also be achieved through downsampling of the point clouds, and still providing better registration results. Therefore, this is the suggested method if low-latency registrations are desired.

The scan method was concluded to have little impact on the registration quality in general (except for the voxelized ICP algorithms). The most important thing to note when scanning objects is to try to keep the object centered (though it's not a requirement, it makes the problem a lot easier) and use a box-correspondence rejector that is placed around the object of interest. The implemented registration framework was able to perform registrations of multiple objects using similar arguments for all the objects. The main surprise was that the scanner was able to register both the ”apple” and ”plastic” scene, which was considered very hard registrations because of the small object sizes, rotational symmetry and 3D reconstruction with the plastic object. Because the apple in principle is impossible to scan, the scanner was probably using the curvature of the underlying plane to register these two objects.

The implementations of the 3D registration completed and shown in this thesis, show that doing registrations in general is a challenging task. It usually requires a lot of parameter tuning and trial and error to get good registrations. Section D is about general guidelines for registration, and can be a good starting point for point cloud registration, but for a new scanner application, some time is expected to get familiar with the new registration domain. Another thing that took a lot of time in this project was the setup of PCL for use with GPU. This was due to old compiler compability and CUDA versions. The computer setup is significantly more complicated and time consuming when working in CUDA.

The work performed here has enabled faster registration. This can be useful for faster prototyping of 3D scanner and registration parameters. This can enable faster 3D reconstructions of object is the future thanks to a faster feedback loop. However, the most important application of this is to give robots fast perception of the environment. This enables the robot to make quicker decisions. Even higher value can be added to the robot if it is able to perform scanning decision as it goes thanks to a faster registration algorithm.

8.2 Further work

This section will look at short term, medium term and long term suggested future work and improvements/extensions. These will be presented in the following list.

- **(Short term)** Implementations of more transformation estimators, correspondence estimator and point correspondence rejectors. The GPU implementations implemented in this project was mostly based on the methods found most efficient in the pre-project for registration of smaller food-like objects. PCL comes with a few other transformation estimators, correspondence estimators and correspondence rejectors that could be added to the GPU framework presented in this project. The current implementations are probably good enough for object scans like the ones done in this project, but for other applications, it might be an idea to add a few more.

- **(Short term)** Using estimated robot pose as initial alignment. This is a simple addition that can enable faster search time and registration correctness.

- **(Short term)** kD-tree construction on GPU. While this project didn't look at that, Hu et al. (2015) showed that great speedup could be obtained by constructing kD-tree on the GPU. kD-tree construction on GPU has not yet been evaluated for a registration task, tough it's not expected to give any drastic changes.

- **(Short term)** Search optimizations. With the current implementations, it was able to perform real-time registration considering the robot application. However, there is probably a huge potential in further optimizations of the GPU algorithms and the kD-tree search. This thesis showed that the kD-tree implementation for CPU was slightly slower than the library implementations. It is probably an additional 10x speedup that can be obtained with minor algorithmic and GPU based optimizations.

- **(Medium term)** A color based scanner. In this project, only the point positions were considered for the registration. Alternatively, the RGB data could be included in the correspondence estimation step, giving better correspondences.

- **(Medium term)** A point removal strategy. This comes from the fact that an accumulated point cloud quickly can grow out of proportions if points are not removed. This project simply used a voxel-grid filter with a small voxel size. However, given that the scanner position is known, it is possible to find which regions of the point cloud that can be removed based on occlusions. While it's hard to say which occluded parts are never visible again, it is possible to remove points that should have been seen by the scanner if they were correctly positioned.

- **(Medium/long term)** Optimal scan trajectories. The scan trajectories proposed in this work was included to show some properties of the registration itself, and is not necessarily good registration paths in themselves. A future work could be to evaluate more scan methodologies and conclude which is best for object scans given the ICP implementations done for this project.

- **(Long term)** Another suggestion to scan trajectories is to use a smart scanner that evaluates the next best view while wile it scans the object. There are multiple techniques that could be used for this, but the main point is that it can more efficiently get a complete view of the object of interest.

APPENDIX A

Acronyms

ATE	Absolute Trajectory Error
BMP	BitMaP image file
CPU	Central Processing Unit
DPT	Raw DePTh file
FLANN	Fast Library for Approximate Nearest Neighbors
FPFH	Fast Point Feature Histograms
FPS	Frames per second
GPU	Graphics Processing Unit
SAC-IA	SAmple Consensus Initial Alignment
ICP	Iterative Closest Point
IMU	Inertial Measurement Unit
IO	Input/Output
IR	Infrared
kD	(As in kD-tree) k-dimensional (as in k dimensions)

LIDAR Light Detection and Ranging

NDT Normal Distributions Transform

OpenCL Open Computing Language

OpenNI Open Natural Interaction (standard 3D scanner interface)

PCL Point cloud library

RANSAC RAndom SAmple Consensus

RGB Red, green, blue

RGB-D Red, green, blue - depth

RMSE Root-Mean-Square Error

RPE Relative Pose Error

SAC SAmple Consensus

SAH Surface Area Heuristic

SDF Signed distance function

SIMD Single Instruction Multiple Data

SIMT Singe Instruction Multiple Threads

SM Streaming Multiprocessor

SVD Singular-Value Decomposition

TSDF Truncated Signed Distance Function

WP3 Work package 3

APPENDIX B

Quaternion operators

This chapter will describe and define some quaternion operators used in the thesis. As there are many aspects of quaternions, this section will only include the quaternion operators used in the thesis.

B.1 Quaternion/matrix rotation magnitude

This section will show how the the magnitude of a rotation can be computed using two quaternions (there are other ways to do this as well). This section starts with a matrix M, which is converted to a unit quaternion in which the rotational magnitude is computed. See section B.2 on how the quaternions are computed from the matrix.

$$||\text{rot}(M)|| = \sqrt{log(q_0^{-1}q_1) \cdot log(q_0^{-1}q_1)} \tag{B.1}$$

q_0 is the unit quaternion $q_0 = 1.0 + 0i + 0j + 0k$ and q_1 is the quaternion representing the rotation of the M matrix. The log operator is the quaternion logarithm operator. This is defined as

$$\text{log(q)} = \text{log}(\sqrt{q \cdot q}) + \frac{(q_i i + q_j j + q_k k)}{||(q_i i + q_j j + q_k k)||} \cos^{-1} \frac{q_{real}}{\sqrt{q \cdot q}} \tag{B.2}$$

q is a quaternion described as $q_{real} + q_i i + q_j j + q_k k$. i, j and k are the imaginary units of the quaternion. The $||q_i i + q_j j + q_k k||$ operator is the euclidean vector length of the vector created from the imaginary components described by the following

$$||q_i i q_j j + q_k k|| = \sqrt{q_i^2 + q_j^2 + q_k^2} \tag{B.3}$$

The final step to deduce this formulation is how the quaternions are converted from the M matrix which is shown in section B.2

B.2 Converting from a quaternion to a matrix

This section will show how to convert a matrix M to a quaternion q. This is based on finding the eigenvector $\vec{v}$ corresponding to the largest eigenvalue. The eigenvectors are not computed from M, but from the following matrix K:

$$K = \frac{1}{3}\begin{bmatrix} M_{1,1} - M_{2,2} - M_{3,3} & 0 & 0 & 0 \\ M_{1,2} + M_{2,1} & M_{2,2} - M_{1,1} - M_{3,3} & 0 & 0 \\ M_{1,3} + M_{3,1} & M_{2,3} + M_{3,2} & M_{3,3} - M_{1,1} - M_{2,2} & 0 \\ M_{3,2} + M_{2,3} & M_{1,3} - M_{3,1} & M_{2,1} - M_{1,2} & M_{1,1} + M_{2,2} + M_{3,3} \end{bmatrix} \tag{B.4}$$

Finally, the quaternion is found by extracting the eigenvectors sorted such that the larger corresponding eigenvalues have lower indices (index as in v_i).

$$q = v_4 + v_1 i + v_2 j + v_3 k \tag{B.5}$$

B.3 Quaternion circular interpolation (slerp)

The quaternion circular interpolation is done using the following equation

$$\begin{cases} \text{slerp}(q_1, q_2, t) = (1-t) \cdot q_1 + t \cdot q_2 & \text{if} \quad |abs(q_1 \cdot q_2)| >= 1 - \epsilon \\ \text{slerp}(q_1, q_2, t) = \frac{\sin(1-t)\arccos|q_1 \cdot q_2|}{\sin\arccos|q_1 \cdot q_2|} \cdot q_1 + \frac{\sin t \arccos|q_1 \cdot q_2|}{\sin\arccos|q_1 \cdot q_2|} \cdot q_2 & \text{otherwise} \end{cases} \tag{B.6}$$

B.4 Quaternion to euler conversion

The quaternion to euler conversion done is using the following definition.

$$\text{quaternionToEuler}(q) = \begin{bmatrix} \text{arctan2}(2(w(q)x(q) + y(q)z(q)), 1 - (2(x(q)x(q) + y(q)y(q)))) \\ \arcsin(\text{clamp}(2(w(q)y(q) - z(q)x(q)), -1, 1)) \\ \text{arctan2}(2(w(q)z(q) + x(q)y(q)), 1 - (2(y(q)y(q) + z(q)z(q)))) \end{bmatrix} \tag{B.7}$$

APPENDIX C

Registration framework arguments

This appendix is a list of the arguments that can be passed to the registration framework. It also includes the default values for the arguments that are not specified from the command line. Each list entry is the argument and the default value. Consult the source code ”readme” for how these arguments are actually set in the final program. Because the project is an extension from the pre-project, multiple of the argument descriptions are taken from the pre project. Before listing all the arguments, a quick intro to the program usage will be given.

C.1 Program usage

The following line is a program usage example:

```
./registration --capture_from_camera --color --use_previous_transform
--visualize_realtime --filter=voxel_grid --leaf_size=0.001
--epsilon=1e-8 --fitness_epsilon=1e-8 --iterations=50
--algorithm=icp_kd_tree_gpu --transformation=point_to_plane_lls
```

This is a good starting point for doing registrations with the registration framework and show the most important values when doing registrations. The last part of this chapter will explain these arguments. Note that the arguments can come in any order.

The 1’st argument specifies the program to run, which is the registration framework. The 2’nd argument specifies that the program shall read directly from a RealSense camera connected to the computer. This could alternatively be exchanged with a list of ”.dpt” or ”.pcd” files that will be registered. The 3'rd argument specifies that the program should read the color data from the camera. The 4'th argument specifies that the registration should use the previous transformation found in the registration as the initial alignment to the next registration. This is almost always a good idea to use. The 5'th argument will setup a window that shows the registration image for image. The 6'th argument will setup a voxel grid filter. The 7'th argument is a filter specification, which will be applied to the previously specified filter argument. In this case, it is sets up the voxel grid filter to use a voxel size of 0.001 m. The 8'th, 9'th and 10'th argument specifies the termination criteria. The defaults are often a bit to high, so it's a good idea to set them a bit lower (which is done here). The 11'th argument specifies to use the fast GPU implementation of the ICP algorithm. The 12'th argument specifies that the ICP algorithm should use the point to plane lls transformation estimator, which is shown to be the best for the robot scanning application. Note that this argument will make the registration framework compute normals with the standard radius search of 0.005 m.

C.2 Arguments

The following list contain the arguments that can be used with the 3D scanner application developed for this project. It also contain default values that are used if the arguments are not specified.

Algorithm = ICP This specifies which registration algorithm to use. The possible values are the following:

ICP Using the ICP implementation provided with the PCL package.

ICP_KD_TREE Using the kD-tree implementation used for this project. This algorithm is explained in section 4.4.3.

ICP_KD_TREE_GPU The same as the previous, only this one will use the GPU implementation. This algorithm is explained in section 4.4.4.

ICP_GPU A GPU implementation using a brute force point correspondence search as explained in section 4.4.2.

ICP_VOXEL Using a CPU implementation of the voxelized ICP algorithm explained in section 4.4.5.

ICP_VOXEL_GPU Using a GPU implementation of the voxelized ICP algorithm explained in section 4.4.6.

Other algorithms It is also possible to select some other registration algorithms that are not very well tested for the registration framework. They are based on PCL implementations. These include ICP_NL (None linear ICP as described in section 2.10.2), FPCS (4-points congruent sets for pairwise surface registration), NDT (Normal distributions transform) and GICP (Generalized ICP).

Iteration count = 50 This specifies how many iterations the registration algorithm should run for.

Leaf size = 0.05 m This is the size of the voxels as used by the voxel grid filter. See section 2.4 for more details about voxel grid filters. This value is measured in meters.

Correspondence method = Euclidean closest point This is the point correspondence method used. More details on this can be found in 3.1.2. Possible values are

correspondence_estimation A standard closest point estimator.

back_projection

normal_shooting

brute_force

Transformation method = SVD This is the method used to compute the transformation from one point cloud to another. More details are found in section 3.1.5. Possible values are:

point_to_plane_lls

dual_quaternion

SVD Singular-value decomposition

LM Levenberg Marquardt

Max corresponence distance = 0.1 m This is the maximum distance the ICP algorithm will search for corresponding points.

Max similar iterations = 10 If the ICP does this number of similar iterations, it will terminate.

Transformation epsilon = 0 If the squared transformation distance drops below this value between consecutive iterations, the registration is considered as converged.

Transformation rotation epsilon = 0 If the rotational difference between consecutive iterations drops below this value, the registration is considered converged.

Euclidean fitness epsilon = -inf If the euclidean fitness between two consecutive scans drop below this value, the registration is considered as converged. The euclidean fitness is error metric as defined in section 5.1.3.

Pass through min value = 0 m This is used by a pass through filter. All points with a z value smaller than this value is removed from the point cloud. This value is measured in meters.

Pass through max value = 10.0 m This is used by a pass through filter. All points with a z value larger than this value is removed from the point cloud. This value is measured in meters.

Use previous alignment = false When this parameter is set, the registration algorithm will use the previous transform as an initial alignment before applying the next transformation.

Calculate normals = false Can force the algorithm to calculate normals. However, if not specified, the registration program will compute them if it is required by the registration algorithm as described in section 2.5.

Calculate GPU normals = fales Works similar to the previous argument. This will do the normals computation on GPU instead of the CPU. This is a faster approach, but gives slightly different results.

RANSAC threshold = 0.01 m A threshold value for the RANSAC plane model. All points within this distance from the plane model are removed from the point cloud when the best fitting plane is found using the RANSAC algorithm for plane fitting. More details on this is found in section 2.6.1. This value is measured in meters.

Use previous point cloud = false If specified, the registration algorithm will align the point cloud to the previously registered point cloud. The alternative is to align to the accumulated model cloud which is the default.

Use previous transform = false This will use the transformation found from the previous registration as an initial alignment to the next registration.

Normals radius = 0.01 m The radius of neighbourhoods used when computing normals in the model. More details can be found in section 2.5. This value is measured in meters.

Cluster tolerance = 0.01 m This is the maximum allowable distance for points to be considered the same cloud. This explained in section 2.6.2. This value is measured in meters.

Min cluster size = 100 All clusters found that are smaller than this are removed from the model.

Max cluster size = 25000 All clusters found that are larger than this are removed from the model.

Filter This has no default value as it is an optional argument. These can be added in chains. The other arguments that affect a filter will be applied to the last specified filter. The following filters can be specified: voxel_grid, plane_model_segmentation, bandpass, euclidean_clustering, box.

Post filter This is the same as the filter argument, but is applied on the accumulated point cloud after the registration is done.

Box min/max x/y/z = -1.0/1.0 m This argument is used together with a box filter. The min values are -1 and the max values are 1. Use the 6 variations of this argument to specify a box. All points outside of this box will be filtered away.

Correspondence box min/max x/y/z = -1.0/1.0 m This argument is used to specify a box. The min values are -1 and the max values are 1. The box rejector is enabled by using at least one of these arguments. Use the 6 variations of this argument to specify a box. Point correspondences within the bounds of this box are kept, while correspondences outside are removed in the registration step. The box is transformed with the point clouds as the algorithm iterates. This makes this box constant relative to the source cloud in a registration.

voxel Size = 0.1 This is the voxel size used by the voxelized ICP algorithm.

voxel pre search range This argument specifies the distance the voxel pre-search strategy should look. Pre-search is disabled if this argument is not specified.

Dynamic voxel levels The number of different resolutions to use for the dynamic voxel search. For each level, the new resolution is computed by multiplying the currrent with the dynamic voxel factor (next argument).

Dynamic voxel factor This argument is used to find the resolution of the next level of the multi-resolution voxelized ICP algorithm. The next level is found by taking the current voxel size and multiplying it with the dynamic voxel factor.

Stack size This is the stack size used by the kD-tree based search algorithms implemented for this project (described in sections 4.4.3 and 4.4.4).

APPENDIX D

Registration suggestions

This chapter will include some experiences and suggestions for doing point-cloud registrations. The chapter is based on my personal experiences when performing point cloud registration.

D.1 Scanning

Try to keep the object centered. Translational movement (relative to center of camera) is problematic for many algorithms. If this is unavoidable, some initial translation alignment is suggested. Try to get some features of the image that can do an initial alignment.

D.2 Filtering

In order to get a good registration, it's recommended to either use a filtering strategy or correspondence rejector (section D.5) to focus the algorithm on the object of interest. Generally, filters remove points and thereby reduce the running time of the registration step and can improve registration quality. The following sections will go through the filters used in this project. A combination of the following filters can be applied to filter all points not part of the object.

D.2.1 Box filter

The first filter one can apply is a box filter. This filter removes all points outside (or inside) a user specified box. This box is typically packed around the object, such that other irrelevant parts of the point-cloud is removed. Use this to focus the attention on the object of interest.

D.2.2 Voxel grid filter

Another filter one can apply is a voxel grid filter. This downsamples the cloud. In certain cases, it can help the registration find the correct transformation, but mostly, this reduces the point cloud size and can greatly speed up the registration step.

D.2.3 Plane model segmentation filter

A plane model segmentation can be applied if the object or other things in the environment can be described with a mathematical expression. This can for example be very useful to remove flat surfaces. This usually can help the registration algorithm to focus on relevant points.

D.2.4 Euclidean cluster filter

A final filtering step is to use an euclidean cluster filter. This filter will cluster the point cloud into regions based on a simple euclidean distance metric before it removes or keeps the regions based on number of points in the region. This is typically used after a plane model segmentation to remove remaining regions that is not part of the object.

D.3 Transformation and correspondence estimation

Generally, for small food-like objects, the point to plane lls transformation estimator is the best considering both performance and precision. It is a bit slower per frame, but gives superior convergence time compared to svd and dual quaternion estimators (based on section 7.8). Correspondence estimation has less effect on the end result. The advice is to use something that enables quick lookup. In the case of unorganized point clouds, the nearest point is probably going to be the best choice. Note that some of the estimators might require to compute normals. This takes additional time, but is usually at a much lower scale than the registration itself. The point to plane lls transformation estimator requires to compute normals, but given the faster convergence, this is outperforms other estimators that don't require normals.

D.4 Convergence criteria

The fitness epsilon 5.1.4 might vary from the different correspondence estimators and transformation estimator. Therefore, there is no general advice to give on how low this should be, because it is highly dependent on the point clouds, transformation estimation and corresponence estimation. Try to keep this value very low (Should be below 10^{-8}), such that it doesn't terminate on this criteria.

The transformation epsilon is a more general convergence criteria. The size of this depends on the scale of the point cloud. Try to keep it around $1 \cdot 10{-}8$ when working objects on the scale of 10 cm (as done in this project). Scale this parameter with the scale of the point cloud and objects that are scanned.

The maximum number of iterations criteria depends on the transformation estimator and correspondence estimator used. If using the point-to-plane metric approximately 20-30 should be more than enough. Consider going slightly higher when using the dual quaternion transformation estimator. Here, you should probably go above 50 iterations.

D.5 Point correspondence rejectors

This is a very important part of having a good registration. It's adviced to either use a good correspondence rejector or a good filtering (section D.2) to enable the registration algorithm to focus on the important regions. As adviced by Rusinkiewicz and Levoy (2001) it's recommended to use a boundary rejection strategy. Examples of this can be the organized boundary rejector in the PCL library or the box correspondence rejector used in this project. Generally, the point to plane lls transformation estimator might perform as good without the rejector, but it depends on the setup. If the SVD or dual quaternion transformation estimators are used, it is more important to use a correspondence rejector as it tends to ”prefer” to align the boundaries instead of the actual object of interest. Filtering is another alternative.

D.6 Additional advice

This section will cover a few tricks that might help the registration.

D.6.1 Trial and error

When doing point cloud registration, there is no single method that is always the best. The advice given in the appendix can be a good starting point, but for a specific application one will probably have to tune the parameters for the application. A lot of time is spent on finding good registration arguments in this project, which is expected when applying registration on a specific domain. Trial and error is the way to go!

D.6.2 Initial alignment

As mentioned in section 2.10.1, it's important to have a good initial alignment for the ICP algorithm. When registering consecutive point-clouds, using the transformation for the previously found transformation usually is a good initial alignment given that the incremental positional change of the camera is small. If this is not the case, it might be an idea to use some other initial alignment strategy (for example doing a registration using keypoints is the point cloud.)

D.6.3 Using GPU for registration

While using a GPU for registration might speed up the registration process, it can be a tedious process to setup the environment (a lot of time was spent on compiling and configuration of PCL for GPU usage in this project). Another thing to note is that the GPU implementations in this project use a similar, but slightly different workflow than that of PCL. Due to the limited time of the project, not all registration implementations in PCL was implemented. Only the point to plane lls transformation estimation and quaternion based method (as Besl and McKay (1992)) was implemented.

Bibliography

Artec3D (2018). Portable 3d scanners. https://www.artec3d.com/portable-3d-scanners.

Arun, K. S., Huang, T. S., and Blostein, S. D. (1987). Least-squares fitting of two 3-d point sets. *IEEE Trans. Pattern Anal. Mach. Intell.*, 9(5):698–700.

Arya, S. and Mount, D. M. (1993). Algorithms for fast vector quantization. In *[Proceedings] DCC '93: Data Compression Conference*, pages 381–390.

ASUS (2018). Asus xiton pro live. https://www.asus.com/3D-Sensor/Xtion_PRO/.

Benjemaa, R. and Schmitt, F. (1997). Fast global registration of 3d sampled surfaces using a multi-z-buffer technique. In *Proceedings of the International Conference on Recent Advances in 3-D Digital Imaging and Modeling*, NRC '97, pages 113–, Washington, DC, USA. IEEE Computer Society.

Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517.

Besl, P. J. and McKay, N. D. (1992). A method for registration of 3-d shapes. *IEEE Transactions on pattern analysis and machine intelligence (PAMI)*, page 239–256.

Biber, P. and Strasser, W. (2003). The normal distributions transform: a new approach to laser scan matching. In *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003) (Cat. No.03CH37453)*, volume 3, pages 2743–2748 vol.3.

Blacker, R. (2016). *Use LibRealSense and Point Cloud Library (PCL) to Create Point Cloud Data.*

Blais, G. and D. Levine, M. (1995). Registering multiview range data to create 3d computer objects. *IEEE Trans. Pattern Anal. Mach. Intell.*, 17(8):820–824.

Chen, X., Ma, H., Wan, J., Li, B., and Xia, T. (2016). Multi-view 3d object detection network for autonomous driving. *CoRR*, abs/1611.07759.

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

Creaform (2018). Portable 3d scanners: Go!scan 3d. https://www.creaform3d.com/en/metrology-solutions/portable-3d-scanners.

Dorai, C., Wang, G., Jain, A. K., and Mercer, C. (1998). Registration and integration of multiple object views for 3d model construction. *IEEE Trans. Pattern Anal. Mach. Intell.*, 20(1):83–89.

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

Franka (2017). Franka website. https://www.franka.de/.

Franka Emika GmbH (2017). libfranka on github. https://github.com/frankaemika/libfranka.

Franka Emika GmbH (2018). *Franka Control Interface Documentation.*

Friedman, J. H., Bentley, J. L., and Finkel, R. A. (1977). An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226.

Guy Godin, Marc Rioux, R. B. (1994). Three-dimensional registration using range and intensity information. In *Proceedings Volume 2350, Videometrics III*, volume 2350, pages 2350 – 2350 – 12.

Harris, M. (2018). Optimizing parallel reductions in cuda. developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.

Hoppe, H., DeRose, T., Duchamp, T., McDonald, J., and Stuetzle, W. (1992). Surface reconstruction from unorganized points. In *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '92, pages 71–78, New York, NY, USA. ACM.

Horn, B. K. P. (1987). Closed-form solution of absolute orientation using unit quaternions. *J. Opt. Soc. Am. A*, 4(4):629–642.

Hu, L., Nooshabadi, S., and Ahmadi, M. (2015). Massively parallel kd-tree construction and nearest neighbor search algorithms. In *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2752–2755.

IDI (2018). Datateknologi, fordypningsprosjekt. https://www.ntnu.no/studier/emner/TDT4501#tab=omEmne

Intel (2017). https://software.intel.com/en-us/realsense/sr300.

Intel (2018a). Intel realsense librealsense on github. https://github.com/IntelRealSense/librealsense.

Intel (2018b). Intel® realsense™ camera sr300. https://software.intel.com/en-us/realsense/sr300.

Intel (2018c). Intel® realsense™ depth camera d435. https://click.intel.com/intelr-realsensetm-depth-camera-d435.html.

Intel® RealSense™ (2016). *Intel® RealSense™ Camera SR300 product datasheet*, 1.0 edition.

Intel® RealSense™ (2018). *Intel® RealSense™ Camera D400 Series (DS5) Product Family datasheet*, 002 edition.

Isachsen, U. J. (2017). Real-time point cloud registration from rgb-d camera mounted on a robot arm using gpu acceleration. Technical report, NTNU. GPU implementations were not actaully done in pre-project.

J. Berkmann, T. C. (1994). Comparison of surface normal estimation methods for range sensing applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16.

Johnson, A. E. and Hebert, M. (1997). Surface registration by matching oriented points. In *Proceedings of the International Conference on Recent Advances in 3-D Digital Imaging and Modeling*, NRC '97, pages 121–, Washington, DC, USA. IEEE Computer Society.

Karimi, K., Dickson, N. G., and Hamze, F. (2010). A performance comparison of CUDA and opencl. *CoRR*, abs/1005.2581.

Klaas Klasing, Daniel Althoff, D. W. M. B. (2009). Comparison of surface normal estimation methods for range sensing applications. In *2009 IEEE International Conference on Robotics and Automation*.

Low, K.-L. (2004). Linear least-squares optimization for point-to-plane icp surface registration.

Lu, F. and Milios, E. E. (1994). Robot pose estimation in unknown environments by matching 2d range scans. In *1994 Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pages 935–938.

Magnusson, M. (2013). *The Three-Dimensional Normal-Distributions Transform — an Efficient Representation for Registration, Surface Analysis, and Loop Detection*. PhD thesis, Örebro University.

Marden, S. and Guivant, J. (2012). Improving the performance of icp for real-time applications using an approximate nearest neighbour search. In *Proceedings of Australasian Conference on Robotics and Automation*, Victoria University of Wellington, New Zealand. Australian Robotics and Automation Association.

Masuda, T. (2002). Registration and integration of multiple range images by matching signed distance fields for object shape modeling. *Comput. Vis. Image Underst.*, 87(1-3):51–65.

Microsoft dev Center (2017). Kinect hardware. https://developer.microsoft.com/en-us/windows/kinect/hardware.

Microsoft developer (2018). Microsoft kinect. https://developer.microsoft.com/en-us/windows/kinect.

Montesano, L., Minguez, J., and Montano, L. (2005). Probabilistic scan matching for motion estimation in unstructured environments. In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3499–3504.

Newcombe, R. A., Izadi, S., Hilliges, O., Molyneaux, D., Kim, D., Davison, A. J., Kohi, P., Shotton, J., Hodges, S., and Fitzgibbon, A. (2011). Kinectfusion: Real-time dense surface mapping and tracking. In *2011 10th IEEE International Symposium on Mixed and Augmented Reality*, pages 127–136.

Nüchter, A., Lingemann, K., Hertzberg, J., and Surmann, H. (2007). 6d slam&mdash;3d mapping outdoor environments: Research articles. *J. Field Robot.*, 24(8-9):699–722.

NVIDIA (2018). Cuda c programming guide. https://www.nvidia.co.uk/gpu-cloud/.

Open Perception Foundation (2017a). Getting started / basic structures. http://pointclouds.org/documentation/tutorials/basic_structures.php#basic-structures.

Open Perception Foundation (2017b). Pcl documentation. http://https.www.pointclouds.org/documentation/.

Open Perception Foundation (2017c). Point cloud library (pcl) developers mailing list. http://www.pcl-developers.org/.

Open Perception Foundation (2018a). The pcl registration api. http://pointclouds.org/documentation/tutorials/registration_api.php.

Open Perception Foundation (2018b). Point cloud library (pcl) on github. https://github.com/PointCloudLibrary/pcl.

perception foundation, O. (2017). Using kinfu large scale to generate a textured mesh. http://pointclouds.org/documentation/tutorials/using_kinfu_large_scale.php.

Pirovano, M., Newcombe, R. A., Izadi, S., Hilliges, O., Molyneaux, D., Kim, D., Davison, A. J., Kohli, P., Shotton, J., Hodges, S., and Freeman, D. (2013). Kinfu – an open source implementation of kinect fusion + case study: implementing a 3d scanner with pcl. In *[!!!Unknown book title]*.

Point Cloud Library (2017). About point cloud library. http://pointclouds.org/about/.

Pulli, K. (1999). Multiview registration for large data sets. In *Proceedings of the 2Nd International Conference on 3-D Digital Imaging and Modeling*, 3DIM'99, pages 160–168, Washington, DC, USA. IEEE Computer Society.

Qiu, D., May, S., and Nüchter, A. (2009). Gpu-accelerated nearest neighbor search for 3d registration. In Fritz, M., Schiele, B., and Piater, J. H., editors, *Computer Vision Systems*, pages 194–203, Berlin, Heidelberg. Springer Berlin Heidelberg.

Rusinkiewicz, S. and Levoy, M. (2001). Efficient variants of the ICP algorithm. In *Third International Conference on 3D Digital Imaging and Modeling (3DIM)*.

Rusu, R. B. (2009). *Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments*. PhD thesis, Institut für Informatik der Technischen Universität München.

Rusu, R. B. and Cousins, S. (2011). 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China.

samotab (2015). Forum post by samotab on how the realsense camera works. https://software.intel.com/en-us/forums/realsense/topic/537872#comment-1810928.

Simon, D. A. (1996). *Fast and Accurate Shape-based Registration*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA. AAI9838226.

Sintef-Ocean (2018). iprocess innovation. http://iprocessproject.com/.

Sintef Ocean (2018). Wp3 flexible processing automation. http://iprocessproject.com/wp-3-flexible-processing-automation/.

Skotheim, O. and Couweleers, F. (2004). Structured light projection for accurate 3d shape determination. In *ICEM12- 12th International Conference on Experimental Mechanics*.

Sturm, J., Engelhard, N., Endres, F., Burgard, W., and Cremers, D. (2012). A benchmark for the evaluation of rgb-d slam systems. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 573–580.

Turk, G. and Levoy, M. (1994). Zippered polygon meshes from range images. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, pages 311–318, New York, NY, USA. ACM.

VISP, V. S. P. (2018). *Tutorial: Camera extrinsic calibration.*

Walker, M. W., Shao, L., and Volz, R. A. (1991). Estimating 3-d location parameters using dual number quaternions. *CVGIP: Image Underst.*, 54(3):358–367.

Weik, S. (1997). Registration of 3-d partial surface models using luminance and depth information. In *Proceedings of the International Conference on Recent Advances in 3-D Digital Imaging and Modeling*, NRC '97, pages 93–, Washington, DC, USA. IEEE Computer Society.

Wikipedia (2017). Voxel. https://en.wikipedia.org/wiki/Voxel.

Zhou, K., Hou, Q., Wang, R., and Guo, B. (2008). Real-time kd-tree construction on graphics hardware. In *ACM SIGGRAPH Asia 2008 Papers*, SIGGRAPH Asia '08, pages 126:1–126:11, New York, NY, USA. ACM.

Zivid labs (2016). 3d color camera from zivid labs demonstrated live at vision 2016. https://www.youtube.com/watch?v=JWxy4FO9yMI.

Zivid labs (2018). Zivid website. http://www.zividlabs.com/.