Development of 3D Camera Based Relative Navigation for Aerial Screwing Applications

Ryota Shiomi

Master Thesis

Erasmus Mundus Master in
Marine and Maritime Intelligent Robotics

Universitat Jaume I

July 25, 2023

Supervised by: Dipl.-Ing. David Bernstein (TU Dresden), Prof. Angel Pobil (UJI)

Co-funded by the
Erasmus+ Programme
of the European Union

Acknowledgments

Firstly, I'd like to express my gratitude to my Master Thesis supervisor, Dipl.-Ing. David Bernstein. His constant support and guidance have been invaluable to my thesis development.

Also, I am appreciative of the suggestions and opportunities to work provided by Prof. Dr.-Ing. Michael Beitelschmidt, Dipl.-Ing. Micha Schuster, and the other PhD students at TU Dresden.

I would also like to extend my thanks to Prof. Pedro J. Sanz and Prof. Angel Pobil for their support and thorough evaluation of my master's thesis.

Abstract

In this thesis, the feasibility of calculating six degrees of freedom (6DoF) using only a depth camera was explored. The optical flow model is a promising tool for calculating translations along the x and y axes. Furthermore, the floodfill algorithm, capable of identifying a specific plane and its normal vector, showed potential in calculating the remaining three degrees of freedom (translation along the z-axis, pitch (rotation around the y-axis), and yaw (rotation around the z-axis)). Estimation of roll (rotation around the x-axis) requires the integration of another algorithm, such as object detection. By integrating these different algorithms, calculating 6DoF becomes feasible. This combined algorithm could serve as a relative navigation tool for various types of robots.

Contents

CHAPTER 1

INTRODUCTION

Contents

Drones have become increasingly popular in recent years and they are utilized in different applications such as aerial photography and visual aerial performance. In addition to this, drones can be used for other manipulation tasks such as opening a valve and fastening a screw if manipulation task can be attached. To realize such manipulation tasks, it is crucial to know where the drones are and the movement. 3D localization of drones can help for navigation tasks or avoiding obstacles.

This thesis explores the existing state-of-the-art methods for 3D localization, conducts experiments with the current methods, and presents a new approach.

1.1 Work Motivation

The objective for this thesis is determining the pose of a drone or robot. At the TU Dresden research lab where this research was conducted, a drone capable of performing screwing manipulation tasks have been developed, as cited in [1]. However, screwing operations are currently manually executed. To achieve full automation, it's essential to know the exact pose of the drone. This thesis focuses primarily on determining the relative pose of drones.

The distinction between relative pose and absolute pose is primarily in the reference frame used for relative navigation. In relative navigation, the Unmanned Aerial Vehicle (UAV) or drone uses a camera to detect the movement of objects. To determine the

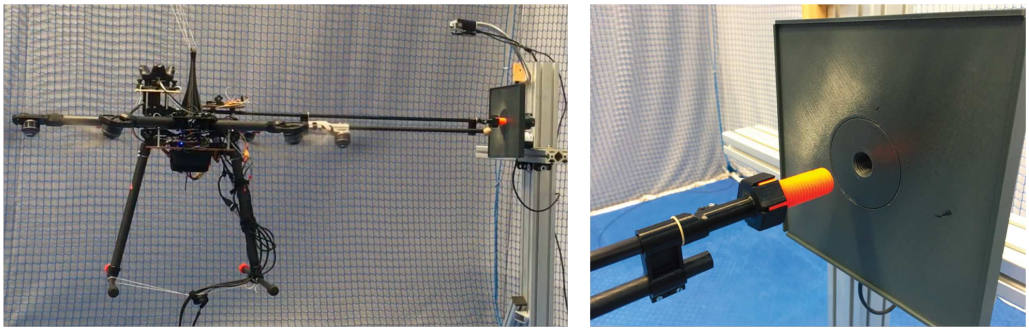

Figure 1.1: The screwing drone developed by TU Dresden [1]

UAV's current position and orientation, it compares a reference point with a prior point. Then, it calculates the precise movement between the two points.

On the other hand, absolute navigation involves the UAV determining its position within a global coordinate system, such as a specific room. This typically necessitates external sources of information such as GPS or pre-constructed 3D maps. Comparing the features observed by the camera with a pre-existing map allows for determining the UAV's absolute position. Also, SLAM (Simultaneous Localization and Mapping) is also often used for this navigation. SLAM can build and update a map of the environment while tracking the location with attached sensors such as LiDAR and a camera.

The focus of this research is on relative navigation for UAVs, a simpler approach compared to absolute navigation. This method is sufficiently suited to the current application of drone-operated automated screwing.

1.2 Objectives

The aim of this thesis is to determine the pose of drones using only a depth camera. A 6DoF (Degrees of Freedom) calculation is required to get the drone's pose. This all information on the 6DoF outlined below. The objectives of this thesis are to acquire this information in real time and with precision.

- Translation x
- Translation y
- Translation z
- Roll (Rotation around x axis)
- Pitch (Rotation around y axis)
- Yaw (Rotation around z axis)

In this thesis, the rotations of roll, pitch, and yaw are defined as shown in Figure 1.2. Roll is the rotation around the camera's line of sight, pitch is the rotation around the lateral (side-to-side) axis, and yaw is rotation around the vertical axis. These definitions are consistent with commonly used camera coordinates found in other references [2].

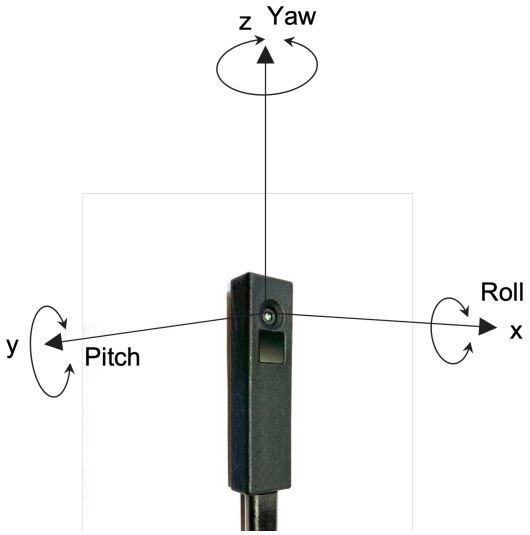

Figure 1.2: 6DoF

In Chapter 2, several state-of-the-art examples are introduced. Chapter 3 discusses the desired system analysis and camera specifications, covering both functional and non-functional requirements. Chapter 4 introduces one of the popular algorithms for pose estimation and includes experimental results. Chapters 5 and 6 discuss other methods, namely Optical Flow and Floodfill. Chapter 7 presents the conclusion and future work. The bibliography and Appendix are located at the end.

CHAPTER 2

State-of-the-art

Contents

Initially, research in the domain of pose estimation for aerial robots was investigated. Next, the state-of-the-art optical flow and floodfill algorithms were investigated. These particular algorithms are considered to be suitable and effective for achieving the goals in this thesis.

2.1 State-of-the-art

2.1.1 Pose estimation with drones

Numerous studies have been conducted on pose estimation for drones, mainly using RGB-D (depth) cameras. While each paper employs its unique algorithm, all papers use combination of different sensors, cameras, and methods to realize relative navigation.

Sun et al. successfully obtained the 6DoF pose of a UAV using an RGB-D camera [3]. To achieve this, they extracted point cloud data of a specific object and reduced the amount of data points to calculate efficiently. This approach allowed them to accurately track the pose of the object in real-time, even in scenarios with high complexity and limited onboard resources. They proposed an ICK (Inter-Frame Consistent Keypoints) track method to track the pose in a more robust way. The name of ICK comes from this algorithm's use of keypoints, which are consistent between the two frames.

Figure 2.1 illustrates the overall flow of the ICK-track algorithm. Initially, two frames are acquired from the RGB-D camera and passed through a semi-supervised learning model to extract only the objects such as bins, cans, or plates. Out of these extracted

objects, only one specified object is chosen. The method of this choice is not explicitly detailed in this paper but it should be extracted depending on an initial setting, such as choosing only a can. From these specific chosen points, the canonical keypoints and the current normal point clouds are selected. Canonical keypoints are a reduced set of points that retain important features. This technique offers a more accurate and robust approach for comparing keypoints. Ultimately, the ICK-Net can identify corresponding points between the canonical points and the current point cloud. In the end, the ICP (Iterative Closest Points) algorithm is applied using these corresponding points.

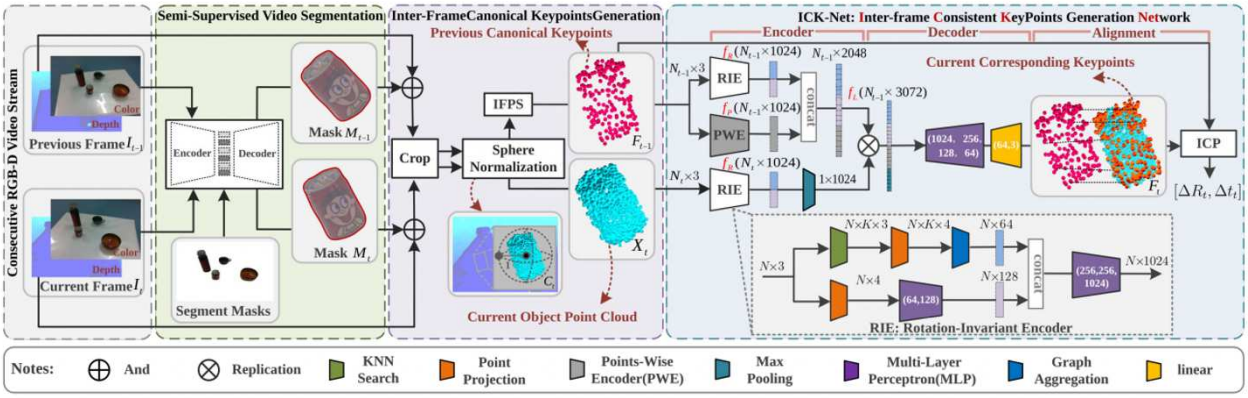

Figure 2.1: ICK-Track Algorithm [3]

Fang et al. proposed a real-time 6DoF localization system for an indoor MAV (Micro Aerial Vehicle) that mainly uses depth information from an RGB-D camera [4]. They realized a robust relative pose estimation (6DoF odometry estimation) mainly using depth images and absolute pose estimation (6DoF localization) using a 3D map with a particle filter algorithm. As this thesis mainly focuses on relative pose estimation, explanation of the section discussing localization in this paper [4] is excluded. Fang et al. also introduced ICP, 3D NDT [5], and a 3D geometric feature-based method [6] for depth-based motion estimation. However, the author points out that these methods are too slow and computationally demanding for Micro Aerial Vehicles (MAVs). Consequently, an alternative approach was proposed, called the range change constraint equation. A detailed explanation of this equation is complex and will thus be avoided in this section. The primary function of this equation is to determine the 6DoF relative motion using two consecutive frames captured by a depth camera. While ICP is based on point cloud points, this equation uses a gradient of the depth image and the depth difference. However, in certain cases, the camera might only capture a ground or wall, resulting in few image features. In such cases, RGB and IMU are combined to compensate for the missing information and minimize the photometric error to get more accurate results.

Perez-Grau et al. utilize images and 3D point clouds to gather odometry information for UAVs. Subsequently, they combine this with other sensor data, such as distances to beacons using UWB (Ultra Wide Band), and predefined 3D maps to pinpoint the

exact location of the UAV [7]. Figure 2.2 depicts the overall pipeline for this system. This odometry is estimated using a stereovision algorithm developed by the author [8], adapted for RGB-D sensors. In this case, x, y, z translations, and yaw are calculated by this algorithm. However, roll and pitch are obtained from the IMU using an accelerometer and gyroscope, as acquiring this information in an indoor environment is challenging. It is worth noting that odometry has a drift error that accumulates over time, and the IMU information can adjust and minimize these errors.

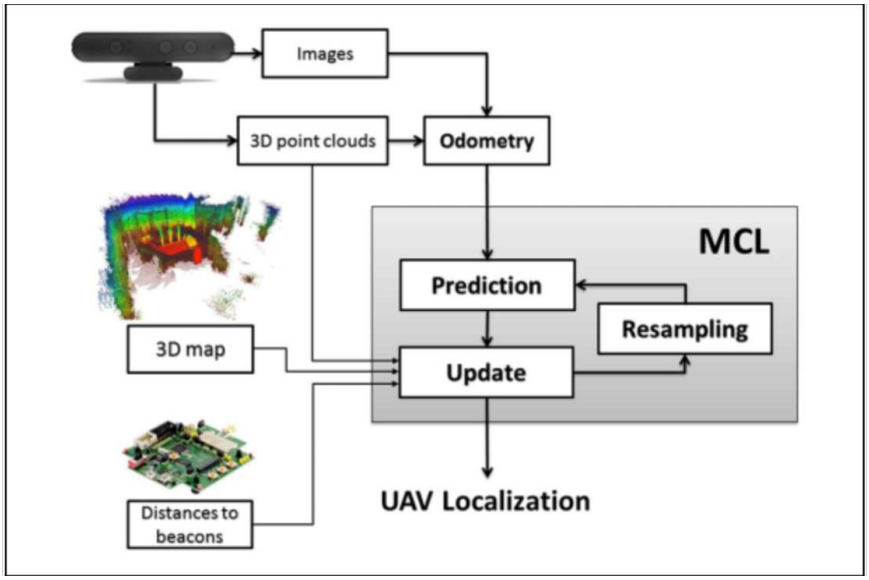

Figure 2.2: Overall system architecture [7]

ICP algorithm is introduced in these references [3], [4], [5], [6] within the papers that are already explained. Based on these references, the ICP algorithm is a popular initial step for pose estimation and serves as a good starting point for understanding the concept of estimation. Consequently, the first experiment conducted with the ICP algorithm. This algorithm will be discussed in detail in Chapter 4.

In addition to this, all introduced research utilize a combination of different sensors and cameras. If it becomes possible to determine 6DoF with only a depth camera, this research could bring a unique novelty.

2.1.2 Getting 6DoF using Optical Flow

Liu et al. proposed a method for estimating 6DoF motion of aircraft utilizing optical flow from two distinct cameras - forward and down-facing [9]. The forward-facing camera calculates rotational velocities and angles, which are subsequently used by the down-facing camera for the estimation of translational velocities. This comprehensive approach enables precise determination of the 6DoF motion. This research proposes that optical flow can estimate roll, pitch, and yaw using the front-mounted camera on an airplane.

However, the optical flow vector alone is not sufficient for a comprehensive calculation. It needs to be paired with more complex methods such as the Unscented Kalman Filter (UKF) [10].

2.1.3 Point cloud other utilization

Arindam et al. proposed a method called FloodFill for point cloud data segmentation [11] [12]. This method is capable of detecting separate objects in scenes where multiple objects are present. Plane segmentation can be widely utilized in numerous applications. For instance, Fangwen Shu et al. developed a SLAM approach to estimate camera pose after segmenting point clouds [13]. Also, a robot can identify walkable paths in a scene, enabling efficient robot navigation. Figure 2.3 illustrates images before and after the application of the floodfill segmentation. This algorithm can distinguish different planes and calculate the normal vector of the plane.

This algorithm could be beneficial for relative navigation as well. Therefore, the a developed version of floodfill algorithm with the pose estimation is presented in Chapter 6.

Figure 2.3: Plane segmentation [12]

CHAPTER 3

System Analysis and Design

Contents

3.1 Requirement Analysis

3.1.1 Functional Requirements

Table 3.1: System Requirements

Task	**Description**
Camera Capture	Capture the environment using a ToF camera
Drone Pose and Location	Determine drone's relative pose (x, y, z + three angles) with the help of close objects
Data Transmission	Publish drone pose on ROS node
Real-time Feedback	Provide real-time feedback to remote operator or central control system on drone pose. Algorithm can work in real-time.
Report accuracy	Provide pose result accuracy

In Table 3.1, these are the functional requirements for estimating the 6DoF position of the drone in the proposed tightening operation [1].

Initially, a camera is needed to capture the environment, in this case a Time-of-Flight (ToF) camera is utilized. In addition to this, the drone should compute its own 6DoF.

In the following step, the drone's self-position should be published on the Robot Operating System (ROS). Simultaneously with the ROS publication, real-time updates should also be provided to the drone operator.

Finally, it is critical to analyze the accuracy of the self-position estimation results. The obtained results can help for improving future accuracy.

3.1.2 Non-functional Requirements

Table 3.2: Non-functional Requirements

Category	**Description**
Performance	The system must be able to capture and process 3D camera data in real-time with minimal latency
Scalability	The system should be capable of operating in larger areas and more complex environments
Reliability	The system should operate reliably with minimal downtime and low error rates
Maintainability	The system should be designed to be easy to maintain and update with clear documentation and modular components

There are 4 different non-functional requirements: Performance, Scalability, Reliability, and Maintainability.

Firstly, regarding performance, the system needs to transmit self-location information in real-time with minimal delay.

Secondly, for scalability, the drone should be capable of functioning without any problems, even when far from the target object or when the object has a complex shape.

Thirdly, the system needs to have reliability. Downtime and errors should ideally be minimized or entirely eliminated.

Finally, the system should have maintainability. It requires making a comprehensive instruction manual. This manual can manage its continued use and allow for necessary updates.

3.2 Camera Specifications

In this thesis, the Flexx2 camera developed by PMD company was utilized. This camera picture is described in Figure 3.1. This camera is also known as Time of Flight (ToF) cameras. A ToF camera operates by projecting light (typically in the infrared spectrum) and measuring the time it takes for this light to reflect off an object and return. Given that the speed of light is constant, the round-trip time allows us to calculate the distance to an object. This camera is capable of generating a depth image (which includes depth

data), a grayscale image, and point cloud data. All detailed specifications are listed in Table 3.3. All data is taken from the website [14].

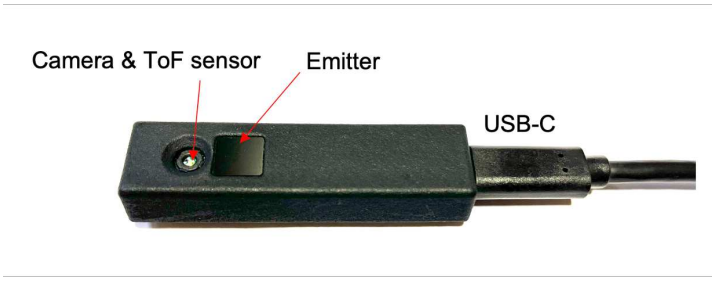

Figure 3.1: flexx2 pmd camera

Table 3.3: PMD Flexx2 Specifications [14]

Specification	**Details**
3D Pixels	38,000
Resolution	224 x 172
Field of View	56 x 44 degrees
Light Source	VCSEL, 940-nm Wavelength
Light Source Independence	Independent of External Light
Port	USB 3.0 Type-C
User Modes	Multiple built-in User Modes
Measurement Depth	From 10cm to 4m
Software	Software Suite “Royale” and API
Size	71.9 x 19.2 x 10.6 mm
ToF Sensor	IRS2381C Infineon® REAL3™ 3D Image Sensor
Illumination	940nm 1 Watt VCSEL (LC1)
Framerate	Up to 60fps (3D frames)
Range	0.1m - 4m
Interface	USB3 Type-C
Accuracy	<= 1% of distance (0.5 –4m @ 5fps), <= 2% of distance (0.1 –1m @ 45fps)
Software	Royale SDK (C/C++ based, supports Matlab, OpenCV, ROS 1, ROS 2)

The field of view of this pmd camera is 56 degrees in a horizontal angle and 44 degrees in a verticle angle. This is described in Figure 3.2.

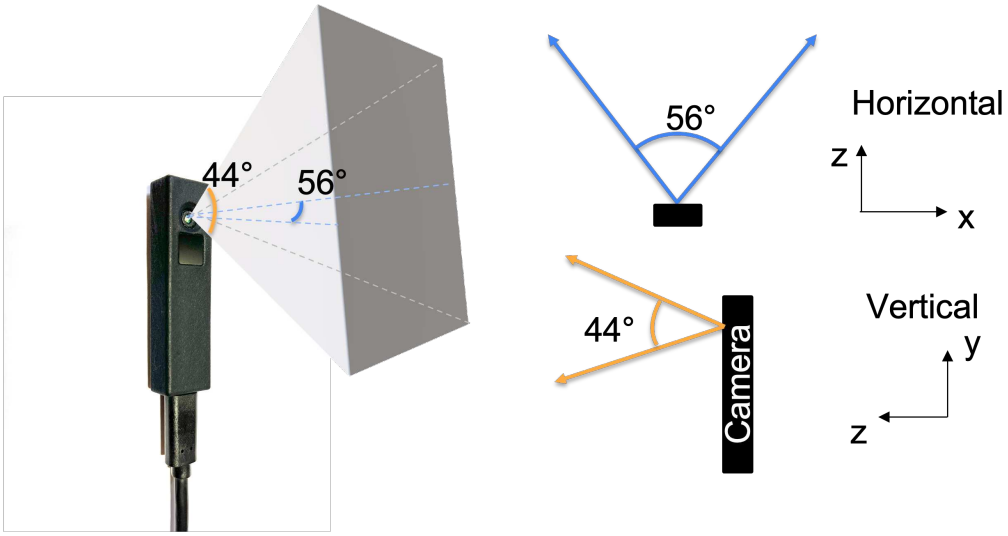

Figure 3.2: Camera's field of view

3.2.1 System Architecture

All experiments involving the depth camera were conducted using MATLAB. Initially, the camera output is published on ROS1 (Robot Operating System 1), and this node is received in MATLAB.

Chapter 4

ICP Algorithm

Contents

As discussed in the state-of-the-art section of Chapter 3, the ICP (Iterative Closest Point) algorithm is a prevalent method for determining robot positioning. To gain an initial tests of ICP's technical aspects and accuracy, an introductory experiment was conducted using a pmd flexx2 depth camera.

4.1 Algorithm

This is a brief introduction to the ICP algorithm.

ICP is typically used with point clouds in a 3D environment. To explain in a more simple way, let's consider the 2D environment. In the Figure 4.1, there are two 2D points frame. One with blue color represents the original point cloud location, and the other with orange color indicates the target points. Considering the real usage of drones using a depth camera, these two frames (original and target) represent point clouds at two adjacent times.

For example, consider a scenario where a point cloud is captured from a depth camera every second and compared. If the current time is 't' seconds, the image frame at 't-1' seconds is considered as the reference, and the frame at 't' seconds is the target. The ICP algorithm performs calculations based on the point clouds at these two distinct times, determining the amount of movement and the angle of rotation.

The ICP's objective is to identify how the original shape needs to be moved and rotated to align with the target shape by comparing two different point clouds. The first step in the ICP algorithm is to find the closest points between the two shapes. The arrows in the ② figure represent these nearest distances, which are called 'errors.' In the beginning, these errors might be large because the two shapes aren't aligned yet. The ICP algorithm then tries to minimize these errors. It calculates a translation vector and a rotation matrix using SVD (Singular Value Decomposition) method explained in Appendix A. The translation vector indicates how much the original frame needs to move, while the rotation matrix shows how much the original points need to rotate.

In a 3D environment, this translation vector and the rotation matrix can be expressed as [15, p.40]:

$$t = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} \tag{4.1}$$

t_x, t_y, t_z: Amount of movement in the x, y and z directions

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \tag{4.2}$$

r_{ij} (i,j = 1,2,3): Values of the rotation matrix

For example, rotation around the x-axis (Roll), y-axis (Pitch), and z-axis (Yaw) by an angle of θ can be expressed as [15, p.42]:

$$R_{\text{roll}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix} \tag{4.3}$$

$$R_{\text{pitch}} = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \tag{4.4}$$

$$R_{\text{yaw}} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.5}$$

Finally, the points, after being translated, can be calculated using the rotation matrix and the translation vector by the following equation [15, p.56].

$$p' = Rp + t \tag{4.6}$$

$$\begin{bmatrix} p' : & \text{Newly translated points after the calculation} \\ R : & \text{Rotation matrix} \\ p : & \text{Original points} \\ t : & \text{Translation vector} \end{bmatrix}$$

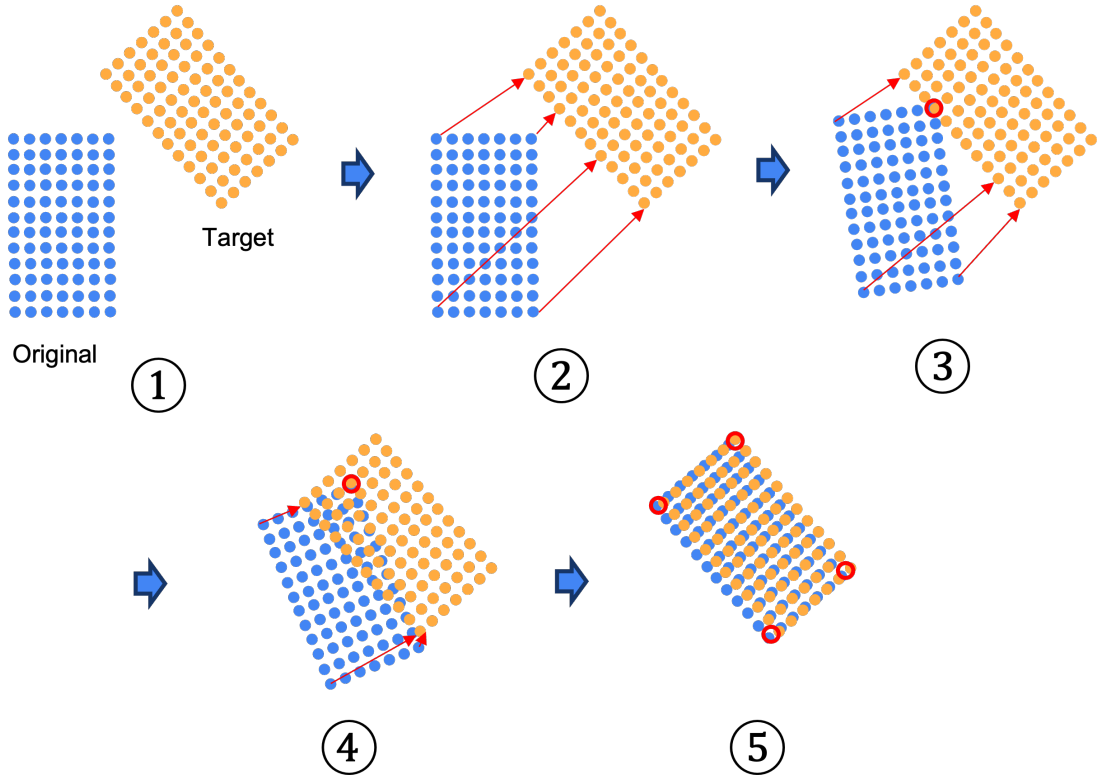

Figure 4.1: ICP algorithm

After doing the move and turn, the first shape gets closer to the goal shape. The errors also get smaller, but it's not a perfect fit yet. This process keeps repeating from ③ to ④: find the nearest points, calculate the errors, then adjust the shape. Each time, the shape gets a bit closer to the goal. After a few rounds, the original shape and the target shape line up nicely (⑤). The errors are now really small, showing that ICP has worked well.

In conclusion, the ICP algorithm stands as a robust technique for aligning two different point clouds. It also can determine precise translation vectors and rotation matrices (6DoF), which are key to understanding the relative positioning and orientation of the robots.

4.2 First experiment with clean data

4.2.1 Experiment Setup

The first experiment was conducted using generated 3D point clouds which are very clean data without noise.

In the beginning, blue 3D points (Data(t-1)) are created in the shape of a box, with 1000 points placed randomly as shown in Figure 4.2. The green points (Data(t)) represent the transformed version of these blue points. The red points are the estimated transformed positions after the ICP calculation. As the red points align well with the green points, it can be concluded that the calculation performed quite well.

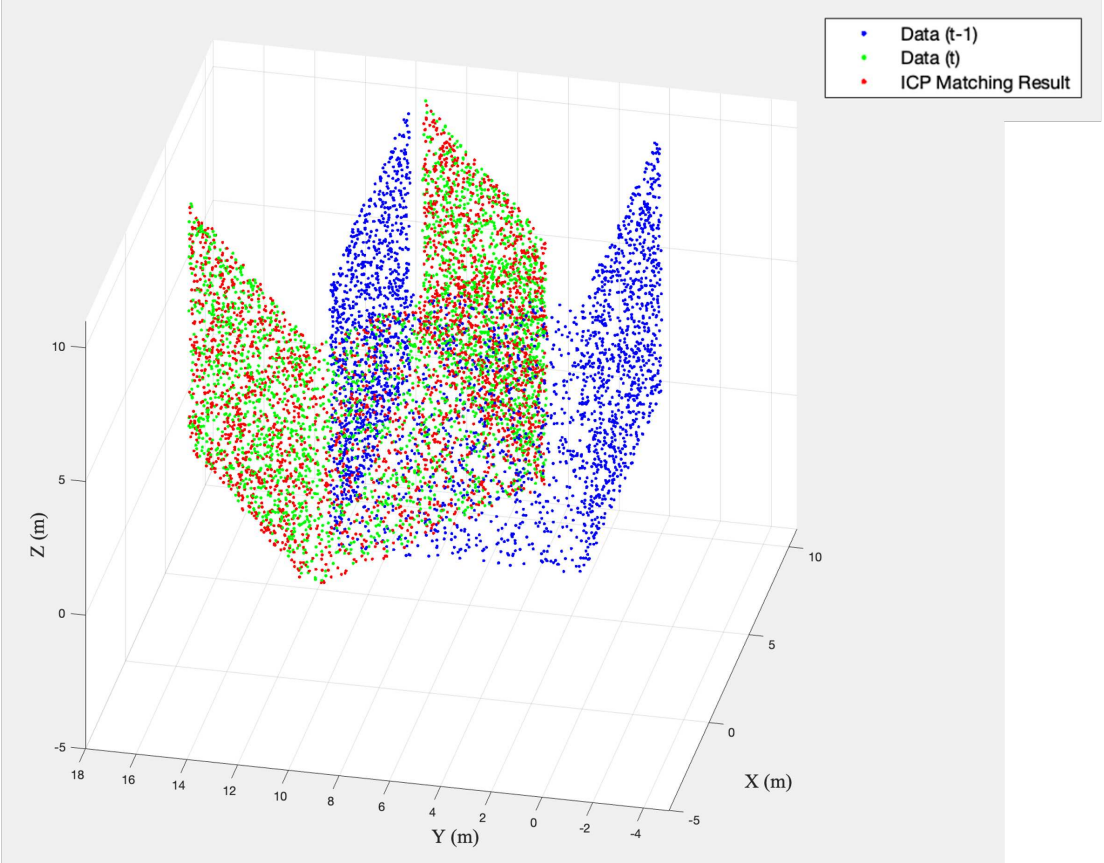

Figure 4.2: ICP applied for clean data

From the blue to green points, the actual translation of x and y and z are $T_{actual} = (2, 2, 2)$. The actual rotation is around z direction and set to 45 degrees.

$$T_{actual} = \begin{bmatrix} 2.00000 & 2.00000 & 2.00000 \end{bmatrix} \tag{4.7}$$

$$R_{actual} = \begin{bmatrix} \cos\left(\frac{\pi}{4}\right) & -\sin\left(\frac{\pi}{4}\right) & 0 \\ \sin\left(\frac{\pi}{4}\right) & \cos\left(\frac{\pi}{4}\right) & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} & 0 \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \approx \begin{bmatrix} 0.707 & -0.707 & 0 \\ 0.707 & 0.707 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.8}$$

4.2.2 Results

After running on a ICP algorithm, the calculation result was:

$$T_{estimated} = \begin{bmatrix} 1.99975 & 1.9960 & 1.8022 \end{bmatrix} \tag{4.9}$$

$$R_{estimated} = \begin{bmatrix} 0.7070 & -0.7072 & 0.0004 \\ 0.7072 & 0.7070 & 0.0000 \\ -0.0003 & 0.0002 & 1.0000 \end{bmatrix} \tag{4.10}$$

To make errors more visible, the result is compared with percentage error, defined as follows:

$$R_{error_{ij}} = \left(\frac{|R_{actual_{ij}} - R_{estimated_{ij}}|}{R_{actual_{ij}}} \right) \times 100 \tag{4.11}$$

i, j refers to the element at the i-th row and j-th column of the each matrix. Translation x, y and z errors (T_{error}) also were calculated in the same way.

The percentage errors are:

$$T_{error} = \begin{bmatrix} 0.0125\% & 0.2\% & 9.89\% \end{bmatrix} \tag{4.12}$$

$$R_{error} = \begin{bmatrix} -0.0142\% & 0.0283\% & - \\ 0.0283\% & -0.0142\% & - \\ - & - & 0\% \end{bmatrix} \tag{4.13}$$

When the actual value is 0, the formula for percentage error becomes undefined and can be misleadingly high. For this reason, the percentage error is not computed and described as a dash symbol "-". The overall errors are very small, indicating that the results are quite precise.

4.3 Second experiment with real data

4.3.1 Experiment Setup

In this second experiment, actual 3D point clouds captured by a depth camera were utilized.

The camera was positioned in front of a small box and captured part of a display as well as the wall. Figure 4.3 illustrates the environment. The black object in the front is

the depth camera. The real data contains considerable noise in the background of the wall, which is excluded in this experiment. Figure 4.4 shows the presence of noise in the background of the wall, even if there is nothing behind it. To eliminate this noise, any point cloud beyond 4 m in the z direction ($z > 4$) is excluded. The wall is located within 4 m.

Figure 4.3: Experiment environment

The exact movement of the camera is 29.7 cm — the length of an A4 paper, only in the x direction. There is no rotation.

$$T_{actual} = \begin{bmatrix} 0.297 & 0.00 & 0.00 \end{bmatrix} \tag{4.14}$$

$$R_{actual} = \begin{bmatrix} \cos(0) & -\sin(0) & 0 \\ \sin(0) & \cos(0) & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.15}$$

4.3.2 Results

Figure 4.5 presents the point clouds as if viewed from above, looking down into the room. The blue points serve as the reference points or, in other words, the starting points. The green points represent moving points, which moved exactly 29.7 cm — the length of an

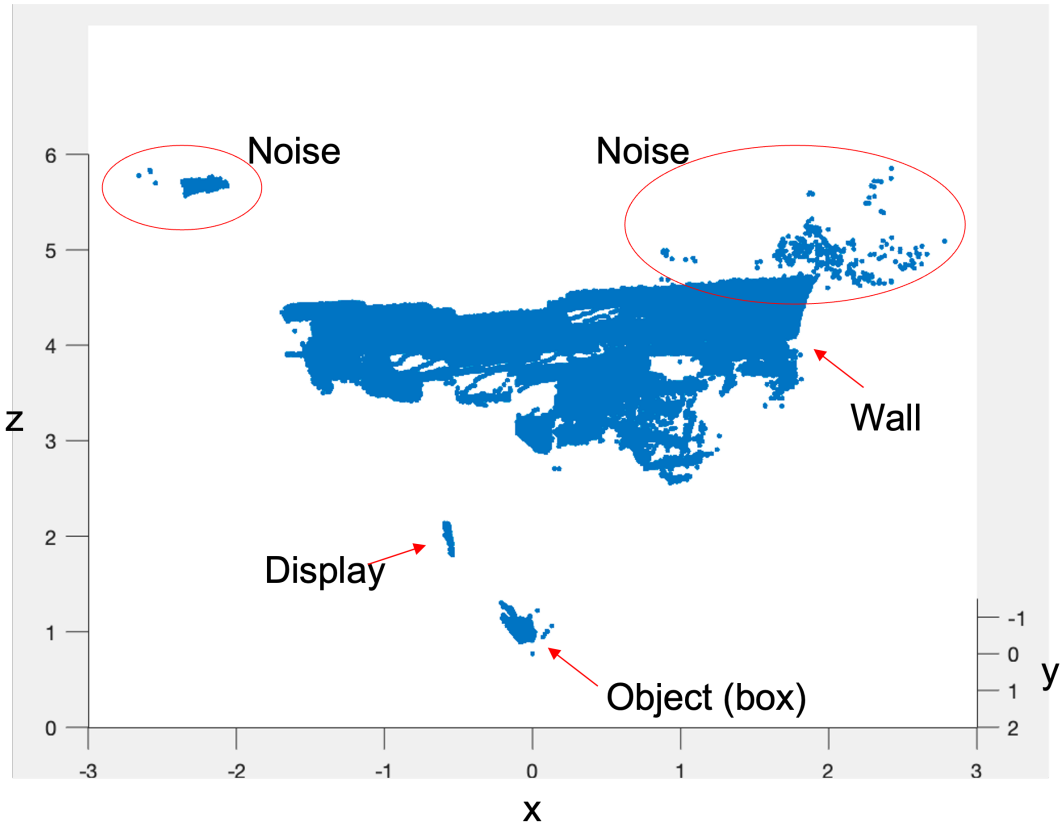

Figure 4.4: Point Cloud Real data

A4 paper. Lastly, the red points are the calculated points following the ICP calculation. Figure 4.6 shows the ICP calculation results as viewed from an angle within the room.

The calculation results are:

$$T_{estimated} = \begin{bmatrix} 0.0198 & -0.0145 & 0.0050 \end{bmatrix} \tag{4.16}$$

$$R_{estimated} = \begin{bmatrix} 0.9988 & -0.0172 & 0.0459 \\ 0.0179 & 0.9997 & -0.0144 \\ -0.0456 & 0.0152 & 0.9988 \end{bmatrix} \tag{4.17}$$

Percentage errors are calculated as:

$$T_{error} = \begin{bmatrix} 93.38\% & - & - \end{bmatrix} \tag{4.18}$$

$$R_{error} = \begin{bmatrix} 0.12\% & - & - \\ - & 0.03\% & - \\ - & - & 0.12\% \end{bmatrix} \tag{4.19}$$

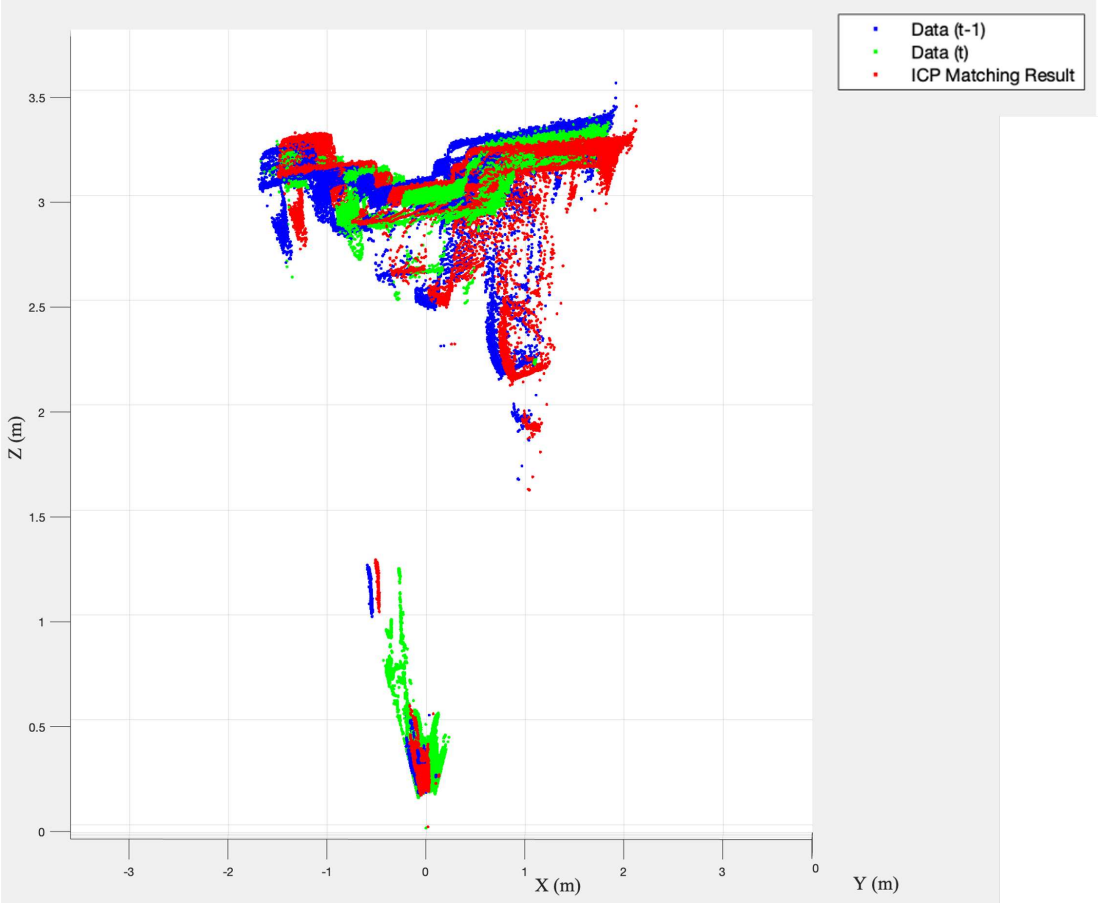

Figure 4.5: ICP calculation results (From above, looking down into the room)

Percentage errors of y and z within T_{error} and some values in R_{error} are undefined because the actual values are zero. Any division by zero results in undefined values to prevent any misleading interpretations.

Thus, the error in the rotation matrix is small, but the error in the translation in the x-coordinate is large. This might be the reason of a lot of noise around the wall.

4.4 Overall Results and Drawbacks of ICP

Table 4.1 presents the comparison of error percentages between clean data (free from noise) and real data acquired from a depth camera (containing noise). Again, the clean data shows higher accuracy, whereas the real data has more errors due to significant noise.

After experimenting ICP algorithm with the real depth camera, it became obvious that there are advantages and disadvantages of the algorithm.

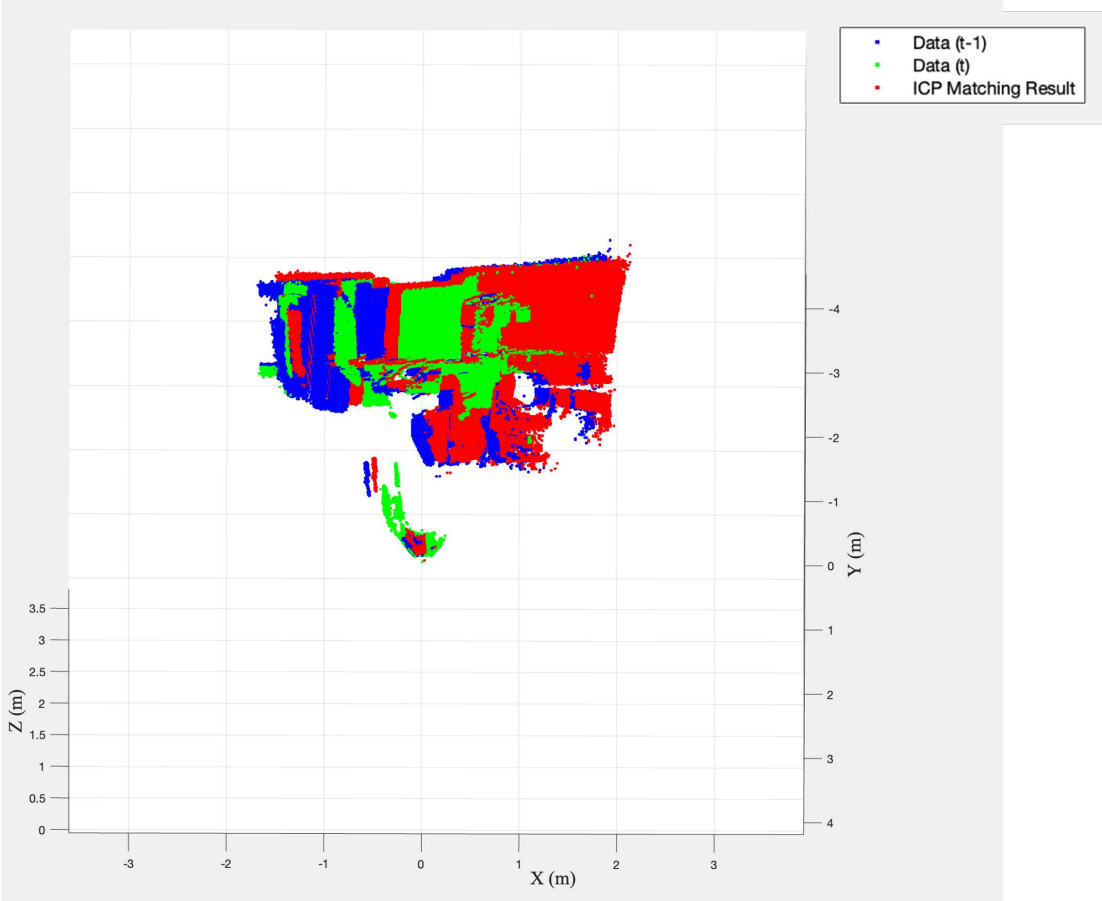

Figure 4.6: ICP calculation results (as viewed from an angle within the room)

Table 4.1: Comparison of errors in clean data vs real data.

	Clean Data	Real Data
Translation Error	$[0.0125\%, 0.2\%, 9.89\%]$	$[93.38\%, -, -]$
Rotation Error	$\begin{bmatrix} -0.0142\% & 0.0283\% & - \\ 0.0283\% & -0.0142\% & - \\ - & - & 0\% \end{bmatrix}$	$\begin{bmatrix} 0.12\% & - & - \\ - & 0.03\% & - \\ - & - & 0.12\% \end{bmatrix}$

Advantage

- Ability to capture small movements and get accurate results if the data is clean

However, data is not clean in real-world scenarios and this advantage only applies to automatically produced point clouds.

Disadvantages

- Computation Time:
  When using all points in a point cloud, the ICP algorithm can be time-consuming. It can take up to 4-5 minutes to perform the calculations. In real scenarios, each point cloud updates within 0.5 seconds. If real-time pose estimation is required, the computation time needs to be less than 0.5 seconds.

- Trade-off Between Accuracy and Efficiency:
  If only a subset of the point cloud is used, the computation time can be reduced. However, this could potentially result in lower accuracy of the outcomes.

- Large Camera Movements:
  If there is significant movement of the camera between frames, it can be challenging to achieve accurate results with ICP.

Table 4.2 shows the overall result for ICP algorithm. It is difficult to use in real scenarios with a depth camera. As already explained in the state-of-the-art section, other researchers do not use the ICP algorithm itself. They combine with other techniques such as semi-supervised learning model to extract specific objects to reduce the point clouds. They also develop their own unique method to find corresponding points easily and overcome the disadvantages of ICP [3]. Due to the fact that the ICP algorithm is not feasible, the next algorithm was investigated, which is optical flow.

Table 4.2: Overall results

6DoF	**ICP with Clean Data**	**ICP with Real Data**
Translation X	○	×
Translation Y	○	×
Translation Z	○	×
Roll	○	×
Pitch	○	×
Yaw	○	×

○: Possible, Very precise
×: Time consuming, Inaccurate.

CHAPTER 5

Optical Flow

Contents

Considering the difficulties of the ICP algorithm, alternative approaches were investigated. I conducted tests using an alternative technique known as optical flow.

5.1 Algorithm

Optical flow is a method that can estimate the motion of objects in a camera scene based on the apparent motion of pixels between consecutive images. Optical flow is mainly calculated based on two assumptions [16].

- Pixel Intensity Consistency: This concept implies that a pixel's color or intensity does not change between two consecutive frames. In other words, the color of a specific pixel remains the same when frames change. (Figure 5.1: Upper two images)
- Neighboring Pixels' Motion Consistency: This concept means that pixels close to each other move similarly. In other words, when an object moves, all neighboring pixels are considered to move in the same direction. (Figure 5.1: Lower two images)

This algorithm estimates the speed vector, which indicates the motion of the pixels, using the gradient of the pixels and the change in pixel intensity over time. Figure

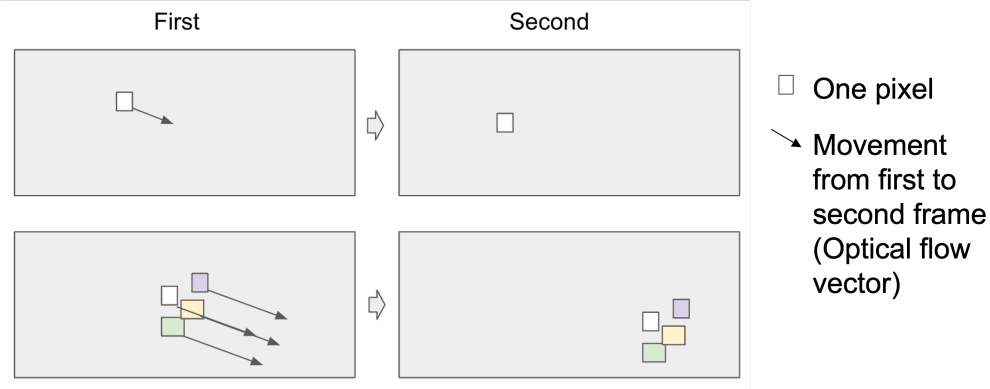

Figure 5.1: Optical Flow Assumptions

5.1 explains the abovementioned assumptions for optical flow. The arrows in Figure 5.1 indicate the vector field of optical flow calculations. Each arrow in the vector field indicates the direction and speed at which a specific pixel has moved.

5.2 Optical Flow for RGB images

5.2.1 Test setup

In the beginning, optical flow was applied for the RGB images captured by a web-cam to test the accuracy of the algorithm. The RGB webcam has better resolution (1920 x 1080) compared to the depth camera (224 x 172). This means its calculation results should be more accurate than those from a depth camera.

Table 5.1: Comparison of Camera Resolutions

Device	**Camera Resolution (pixels)**
Web-cam camera	1920 x 1080
Depth camera (PMD Flexx2)	224 x 172

Figure 5.2 depicts the test environment. The test environment for the y-direction differs from that of the x-direction. There is one black object is added. This is due to black objects yielding superior results with the optical flow. To get a better result on y direction, the different environment was utilized.

5.2.2 Optical Flow Algorithm

In this section, the code for the optical flow algorithm applied to these RGB images is explained. The complete code is located in Appendix B.

1. Firstly, initialize the webcam

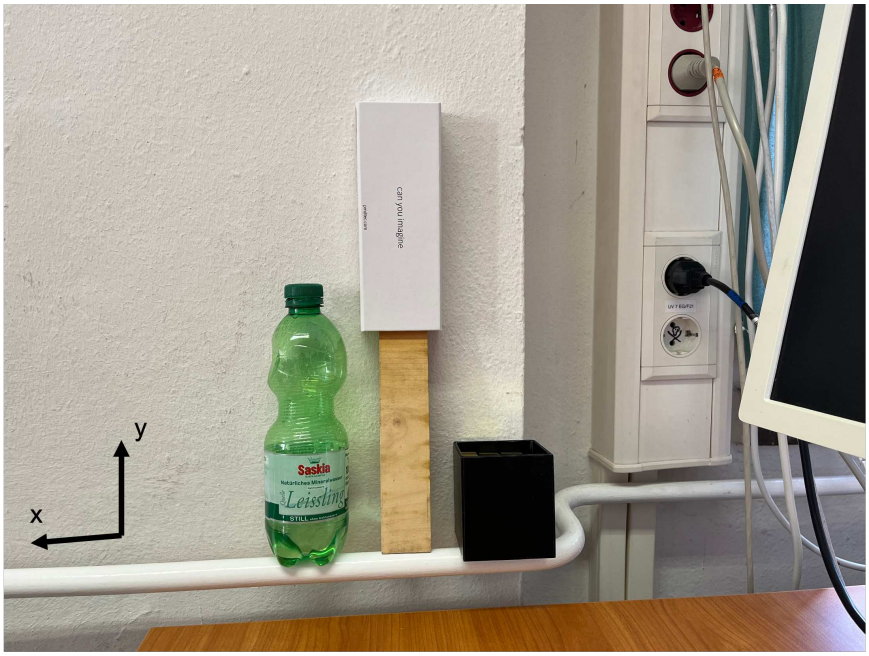

Figure 5.2: Optical Flow RGB Testing environment (x direction)

2. Create optical flow using the Lucas-Kanade method (This method is explained in detail in Section 5.3.2)

3. The image is converted to grayscale. This is a necessary step for applying optical flow, as it simplifies the image and enhances computational efficiency.

4. Calculate the movement vector from the optical flow. This movement vector indicates the movement in the x and y directions.

5.2.3 Results

The results of the optical flow for RGB images are presented in Table 5.2 and Table 5.3. As illustrated in Figure 5.2, a movement of the camera to the left in the x-direction is considered positive. Similarly, upward movement of the camera is also considered to be in the positive direction.

The corresponding video [17] demonstrates the camera moving parallel to the wall in x direction by 10 centimeters, with Optical Flow vectors (blue arrows). The video [18] shows the camera's movement in the y-direction.

The experiment data demonstrates good consistency. After the camera moves 10 cm to the right and left, the calculated pixel changes are -0.0425 (when the camera moves -10 cm) and 0.0307 (when the camera moves 10 cm), respectively. This indicates that the optical flow calculation is accurately capturing the lateral motion (x direction) of the camera.

Additionally, there is no actual movement in the y direction. The calculated pixel change is also very minimal, which accurately reflects the static condition in that direction.

Table 5.2: Experimental Results with Differences in X and Y pixels

	Actual Movement		Optical Flow Calculation		Pixel Difference	
	X (cm)	Y (cm)	X (pixel)	Y (pixel)	ΔX (pixel)	ΔY (pixel)
Starting point	0	0	-0.0021	-0.0049	-	-
Experiment 1	-10	0	-0.0446	-0.0084	-0.0425	-0.0035
Experiment 2	10	0	-0.0139	-0.0054	0.0307	0.0005

The second experiment outlined in Table 5.3 involves movement in the y direction. The camera was moved up and down by 10 cm, resulting in a pixel movement of 0.0340 pixels when moving up and -0.0205 pixels when moving down. As there is minimal pixel movement in the x direction, the overall pixel movement accurately corresponds with the actual physical movement.

After examining the details of the optical flow model, it becomes apparent that optical flow can only estimate the motion of the camera. To determine the exact movement of the camera, additional calculations (using a pin-hole camera model) are required. These calculations will necessitate the focal length, as well as the distance between the camera and the object.

Table 5.3: Second Experimental Results with Differences in X and Y pixels

	Actual Movement		Optical Flow Calculation		Pixel Difference	
	X (cm)	Y (cm)	X (pixel)	Y (pixel)	ΔX (pixel)	ΔY (pixel)
Starting point	0	0	-0.0013	-0.0020	-	-
Experiment 3	0	10	-0.0059	0.0320	-0.0046	0.0340
Experiment 4	0	-10	- 0.0073	0.0185	-0.0014	-0.0205

5.3 Optical Flow for Depth images

5.3.1 Test Setup and Implementation

The pmd flexx2 camera (specifications are in Table 6.1) was used in this experiment. These are four steps for optical flow algorithm on a depth image.

1. Get depth image from a camera
2. Noise reduction for the image to eliminate all noises
3. Resize the depth image and calculate the gradient of the image
4. From the gradient, estimate the optical flow. This optical flow indicates the movement of the object in the image

5.3.2 Noise Reduction methods

When a depth camera captures an environment, a lot of noise are present as shown in Figure 5.3. This noise can reduce the effectiveness of the optical flow. To overcome this problem, noise reduction methods are implemented in three ways and compared each other. All three ways showed significant improvement.

- Median, Gaussian Filters
- Using another optical flow model
- Removing points with 0 data

Median and Gaussian Filters

MATLAB's functions are utilized for these filters. In the code, the 'medfilt2' function is a median filter. The median filter can eliminate noise by replacing each pixel's value with the median value of the neighboring 3x3 pixels. This filter can smooth the depth image.

A Gaussian filter is used via the 'imgaussfilt' function. The Gaussian filter can also smooth the image, based on the Gaussian distribution.

In MATLAB function, these filters can be written as:

```
1 depthImage = medfilt2(depthImage); % Apply median filter to depth image
2 depthImage = imgaussfilt(depthImage, 1); % Apply gaussian filter to depth image
```

The test environment is the same as Figure 5.2, which includes a bottle, a box, and charging cables.

Figure 5.3 displays depth images before and after applying the median filter. The images on the right are raw depth images taken from a depth camera. The images on the left are the depth gradient magnitude, which signifies the magnitude of the gradient (i.e., the rate of depth change) of the depth image. Parts of the image with a strong white color indicate a large gradient. At this point, the optical flow is applied to the depth gradient strength, not to the raw depth image obtained directly from the camera. For these images, the Lucas-Kanade method is used for optical flow in both cases, but no optical flow vectors are generated. (A detailed explanation will be provided in the next section.) The images with the blue background are depth images after implementing the median filters. As shown in Figure 5.3, the median filter was successfully reducing background noise.

Figure 5.4 displays a depth image after applying the Gaussian filter. In comparison to the median filter, the depth gradient magnitude becomes blurred and less visible. For the depth camera, the median filter proves to be a better choice than the Gaussian filter.

Using another optical flow model

This experiment initially used a optical flow model called Lucas-Kanade. The optical flow vector was not calculated in areas with lower white gradients. On the other hand,

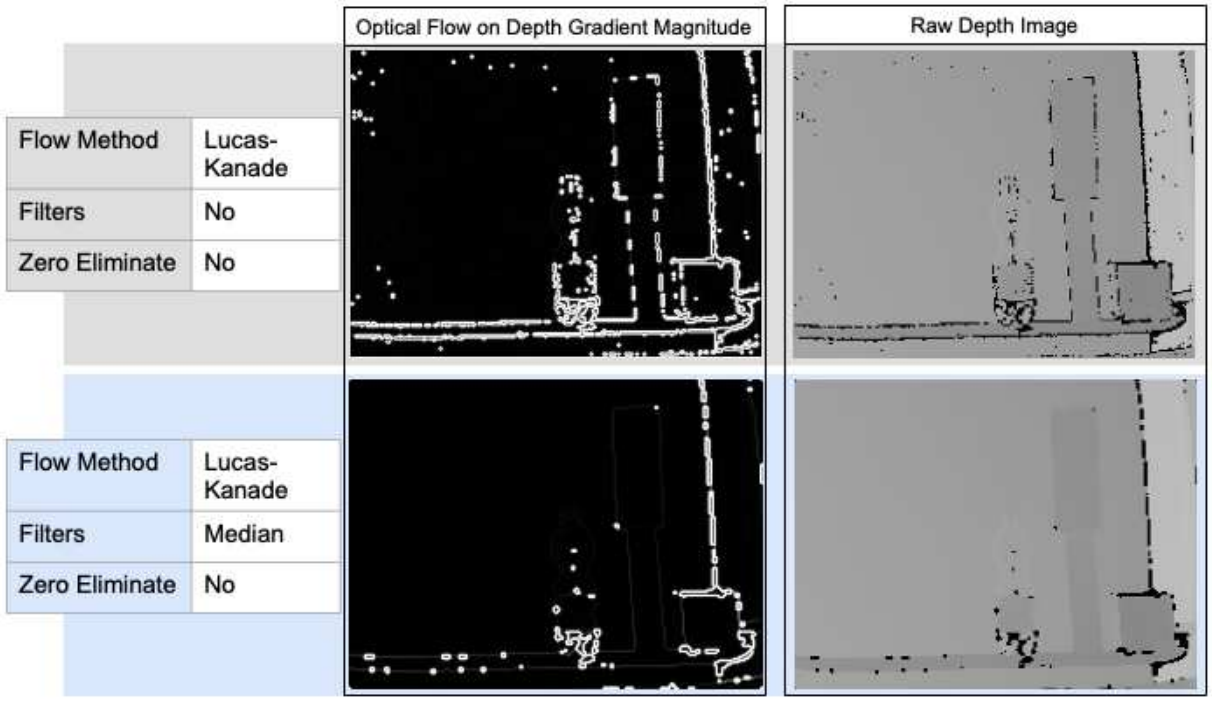

Figure 5.3: Raw images vs Images after applying the Median filter

using Horn-Schunck, motion vectors can be calculated even when the depth difference is relatively small.

In MATLAB function, these models can be written as:

```
opticFlow = opticalFlowLK('NoiseThreshold', 1.5); %Lucas-Kande model
opticFlow = opticalFlowHS; %Horn-Schunck model
```

Figure 5.5 shows a comparison between images without any filters using the Lucas-Kanade method, and images with median filters using the Horn-Schunck method. In comparison to the Lucas-Kanade method, the Horn-Schunck method was able to generate optical flow. The red arrows represent the movement vectors. Thus, it can be stated that the Horn-Schunck method is suitable for applications involving depth cameras.

It is believed that the reason for this is due to the fact that the two methods have different concepts.

The Lucas-Kanade method tracks specific, small features in an image. It assumes that every pixel within a neighborhood moves at the same speed and in the same direction. It is difficult to pinpoint the exact reason why the optical flow was not detected when using the Lucas-Kanade method. However, it may be due to the difficulties in accurately detecting fine movements unless the feature is strong enough, or in other words, unless the magnitude of the gradient is significant.

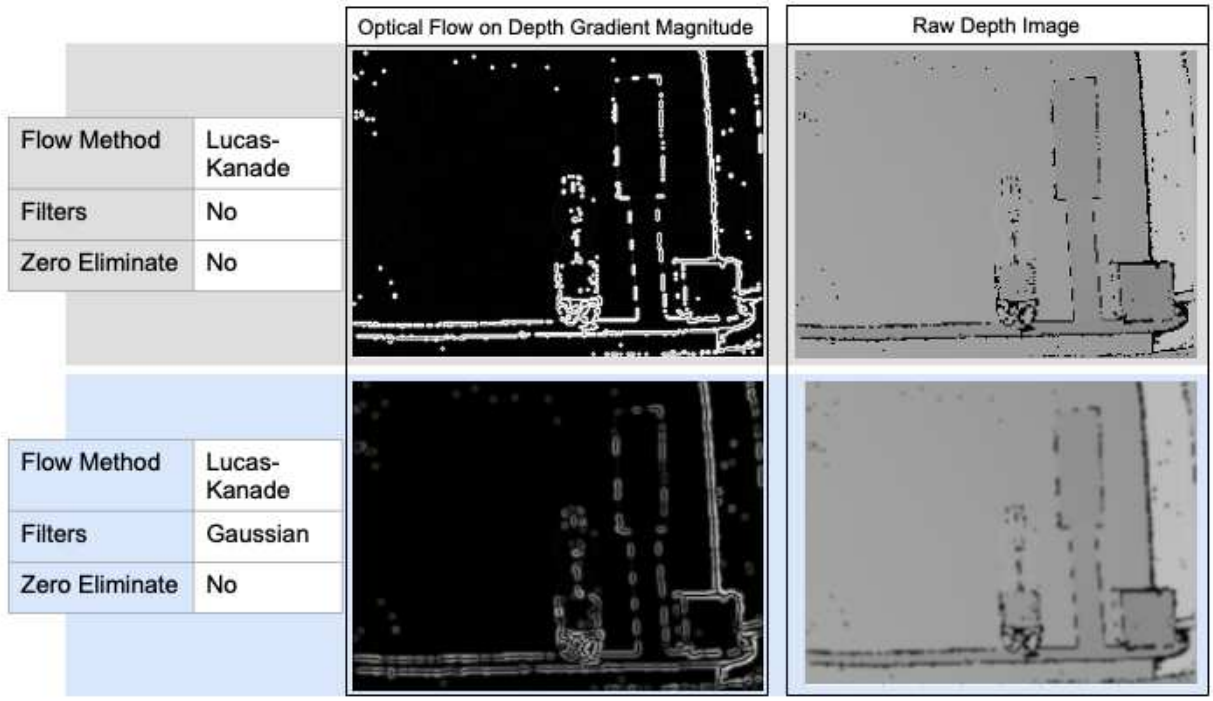

Figure 5.4: Raw images vs Images after applying Gaussian filters

On the other hand, the Horn-Schunck method estimates the overall motion pattern, not focusing on specific features. Even if the gradient is not significant, motion can still be detected by estimating the overall consistency of motion in the region. It appears that the Horn-Schunck method was able to produce results even in areas with low gradient magnitudes. Also, another possibility is that the optical flow was generated due to the presence of certain noises. It is also important to note that this method is sensitive to noise [19].

Zero data elimination

Figure 5.5 and other previous figures indicate that the depth images still include many zero pixel values, which are the small black points in the raw depth image. A pixel with a value of zero creates a noticeable gradient magnitude around it, as it significantly differs from adjacent pixels with non-zero values. These sparse gradient errors reduce the effectiveness of optical flow. To overcome this issue, a new approach was developed and tested, which involves replacing zero-valued pixels with the minimum non-zero values from their vicinity. The steps (1) to (6) below detail this process, and Table 5.4 provides example pixels for easier understanding.

- (1): In the beginning, let's consider 5×5 pixels as an example to understand better in this algorithm. The example pixels of (1) in Table 5.4 represent 5×5 pixels

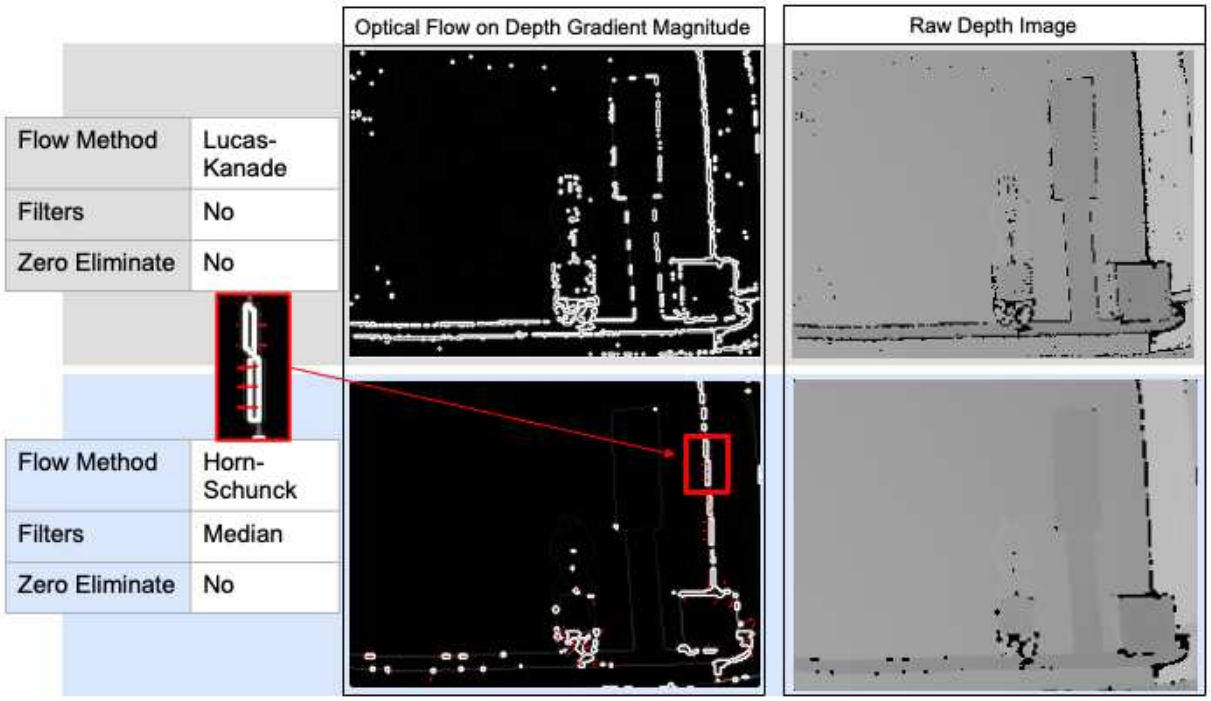

Figure 5.5: Raw images vs Images with different optical flow method

gained from a depth camera. There are 9 zero pixels in total within 25 pixels.

- (2): To eliminate these zero values, NaN to the surrounding pixels are added as shown in (2) in Table 5.4. This is because all edge pixels have no neighbors and NaN values become these neighbors. Then, starting from the top left pixel, check whether the value is 0 or not. The first 0 pixel is in the 1st row and the 3rd column (the number in red color).

- (3): Then the algorithm examines the 0-valued neighbors. It starts with 3×3 (blue parts in (3) Table 5.4) and then expands the neighborhood further, up to 5×5, if there are no non-zero pixels in its neighborhood. The zero value will be replaced by the smallest non-zero value contained in its neighboring pixels. For example, in the situation shown in pixels (3), the minimum value of the neighboring pixels is 1.

- (4): The 0 pixel (1st row and 3rd column) is replaced with 1.

- (5): This is repeated for each pixel. If a zero value is present, it is replaced by the minimum non-zero value of its neighboring pixels. Replaced non-zero values will not be used for the next replacement. In the end, the pixels look like this, and the 0 pixels disappear.

- (6): In the end, any NaN values have been removed and the pixels now contain no zeros.

Table 5.4: Six steps for eliminating zero values

(1)

2	4	0	1	2
0	3	5	2	1
1	0	0	2	0
0	0	0	1	0

(2)

N	N	N	N	N	N	N
N	2	4	**0**	1	2	N
N	0	3	5	2	1	N
N	1	0	0	2	0	N
N	0	0	0	1	0	N
N	N	N	N	N	N	N

(3)

N	N	**N**	**N**	**N**	N	N
N	2	**4**	**0**	**1**	2	N
N	0	**3**	**5**	**2**	1	N
N	1	0	0	2	0	N
N	0	0	0	1	0	N
N	N	N	N	N	N	N

(4)

N	N	N	N	N	N	N
N	2	4	**1**	1	2	N
N	0	3	5	2	1	N
N	1	0	0	2	0	N
N	0	0	0	1	0	N
N	N	N	N	N	N	N

(5)

N	N	N	N	N	N	N
N	2	4	1	1	2	N
N	1	3	5	2	1	N
N	1	1	1	2	1	N
N	1	1	1	1	1	N
N	N	N	N	N	N	N

(6)

2	4	1	1	2
1	3	5	2	1
1	1	1	2	1
1	1	1	1	1

Figure 5.6 presents the results after applying zero-data elimination using the Horn-Schunck method. Although the object features are less visible in the depth gradient magnitude, most of the noise is reduced, simplifying the image. The red arrows, representing the optical flow motion, are visible and appear more accurate due to fewer features on the gradient magnitude. Only areas with significant depth differences are visible, while shorter distances less than 20 cm are not apparent in the depth gradient magnitude. An exception is the bottom of the bottle, which remains slightly visible due to the light reflection on the water. Some small optical vectors are detected, as shown within the red square in Figure 5.6.

5.3.3 Results

The optical flow motion test was performed by using the depth image gradient magnitude after reducing noise. The camera was manually moved in the x-direction, both left and right, with increments of 20 cm. Another experiment was conducted moving in the y direction, both up and down, with increments of 20 cm. The experimental results of this process are presented in Table 5.5.

The pixel difference is calculated as:

$$\Delta Y_{\text{Exp1}} = Y_{\text{Exp1}} - Y_{\text{Starting point}} \tag{5.1}$$

$$\Delta X_{\text{Exp2}} = X_{\text{Exp2}} - X_{\text{Starting point}} \tag{5.2}$$

$$\Delta Y_{\text{Exp2}} = Y_{\text{Exp2}} - Y_{\text{Starting point}} \tag{5.3}$$

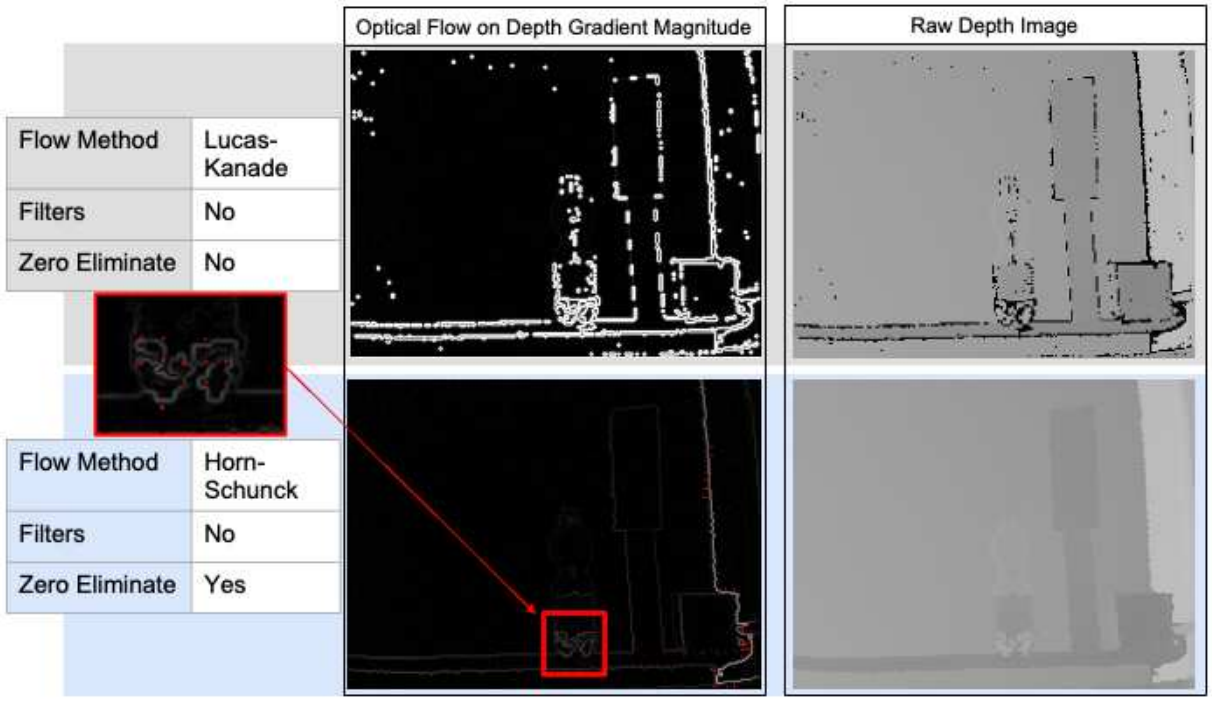

Figure 5.6: Raw images vs Images with zero elimination

One result was a pixel movement of -0.0017 for a shift of -20 cm in the x direction. Similarly, for movements in the y direction, the calculated pixel difference was almost zero. One of the reasons is that pixels continuously move back and forth, even when the camera remains stationary.

Table 5.5: Experimental Results with Differences in X and Y pixels

	Actual Movement		Optical Flow Calculation		Pixel Difference	
	X (cm)	Y (cm)	X (pixel)	Y (pixel)	ΔX (pixel)	ΔY (pixel)
Starting point	0	0	-0.0035	0.0001	-	-
Experiment 1	-20	0	-0.0052	-0.0004	-0.0017	-0.0005
Experiment 2	20	0	-0.0035	-0.0032	0.0017	0.0028
Experiment 3	0	20	-0.0059	-0.0032	-0.0024	0
Experiment 4	0	-20	-0.0004	-0.0019	0.0055	0.0013

Table 5.6 presents the results of the z-direction experiment. The test was conducted with the x and y direction tests at the same time. Despite no movement in the z direction, with a constant distance of 60 cm maintained between the wall and the camera. The results were very precise. The experiment consistently calculated the exact distance to the wall, with negligible errors between different tests.

These results suggest that it could be feasible to calculate x and y movement with op-

Table 5.6: Experimental Results with Differences in Z movement (Actual Distance to the wall is 0.60 [m])

	Actual Movement	Depth Camera Result	Movement Difference
	Z (m)	Z (m)	ΔZ (m)
Starting point	Keeping 0.6 m	0.5695	-
Experiment 1	Keeping 0.6 m	0.6777	0.1082
Experiment 2	Keeping 0.6 m	0.6189	-0.0588
Experiment 3	Keeping 0.6 m	0.6363	0.0174
Experiment 4	Keeping 0.6 m	0.6559	0.0196

tical flow for a depth camera. However, the current version of the optical flow algorithm still needs updating. This experiment video can be accessed here [20].

5.4 Conclusion

Table 5.7 presents the overall potential for optical flow on both RGB and depth images.
For optical flow on RGB images,

- Translations in x and y can be calculated in pixel terms. While this method is not extremely precise, it is relatively effective. The calculation remains pixel-based and it is necessary to convert to length using the pin-hole camera model.
- Translation in z direction can be calculated very well.
- A roll calculation was also examined. However, as the optical flow vectors do not consistently cover all the pixels, an additional sensor like an IMU (Inertial Measurement Unit) is necessary to obtain an accurate roll rotation. Alternatively, employing an object detection system to identify the frame of the object and compute its alignment could be another possible approach.
- Determining pitch and yaw would be quite challenging as RGB images exist in a 2D environment.

The application of optical flow on depth images presents challenges, as pixel calculations are not highly accurate and require more precise methodologies. This may also involve considering the use of a higher-resolution camera or different algorithm. The parts where the RGB image fails to calculte, the depth image might not offer a solution either. This is because both types of images rely on the same algorithm.

Table 5.7: Overall possibilities (Optical Flow)

6DoF	**RGB Optical Flow**	**Depth Optical Flow**
Translation X	○	△
Translation Y	○	△
Translation Z	×	○
Roll (around x axis)	×	×
Pitch (around y axis)	×	×
Yaw (around z axis)	×	×

○: Possible, Relatively precise △: Inaccurate ×: Not possible.

Chapter 6

FloodFill

Contents

In this chapter, floodfill algorithm is explained. This algorithm is mostly based on this paper [12] and developed by one of the other Master's students at TU Dresden. Sections 6.1 and 6.2 are the algorithm developed and owned a copyright by the chair of Dynamics and Mechanism Design at TU Dresden. Section 6.3 is a developed algorithm by myself.

6.1 Algorithm

The flood-fill algorithm, which utilizes a 3D point cloud in a way similar to the Iterative Closest Point (ICP) algorithm, is explained below in four steps, along with Figure 6.1.

Firstly, let's imagine the camera capturing an image of a PC display directly from the front. A wall is positioned in the background, so there is a distance difference between the display and the background. This algorithm can identify the largest plane from the point clouds captured by the depth camera.

To clarify, the outermost rectangle represents the camera frame, while a smaller rectangle within the camera frame represents the plane of an object. In this case, this plane is a PC display.

① In the beginning, one point at [109, 86] is selected from the whole 3D point clouds, represented here as a red dot. In the Figure 6.1, the initial position is shown slightly above and to the right, but the actual first point is located at the center of the frame. This camera resolution is 224 × 172 and the middle point is at [109, 86].

② The neighboring points around the starting point are then examined. If these points satisfy certain conditions listed below, they are considered as the same group, forming part of the same plane. Certain conditions are listed below with Figure 6.2.

- **Point-to-Point Distance**: This condition checks whether the distance between two points is within a certain threshold. If the distance is greater than the threshold, those points are likely not on the same plane. The threshold setting is described in detail in the Appendix.
- **Point-to-Plane Distance**: This condition calculates the distance between a point and the plane passing through the seed point, and checks if it is within a certain threshold. If the distance is greater than the threshold, the one point is likely not on the same plane.

If the calculated distances for both of these measures are within their respective thresholds, then the point is considered to be on the same plane. These newly included points which are on the same plane are represented in blue color.

③ This second process (②) is repeated around newly added points, which becomes expanding from the initial seed point.

④ Once there are no neighboring points that satisfy the given conditions, the first step of identifying one plane concludes. Finally, the overall pixel plane equation, centroid, and normal vector of the plane can be calculated. After examining all the points, the new process starts from ①. In this new process, the starting point is set as the centroid of the plane.

This process is also illustrated in the flowchart depicted in Figure 6.3.

6.2 Calculation Detail

The final process of computing the plane equation, the centroid, and the normal vector of the plane can be accomplished as follows.

- **Centroid**:
  The centroid can be calculated by extracting all the points (x, y, z) that lie on the same plane and computing the mean values for C_x, C_y and C_z.

$$C_x = \frac{x_1 + x_2 + \ldots + x_n}{n} \tag{6.1}$$

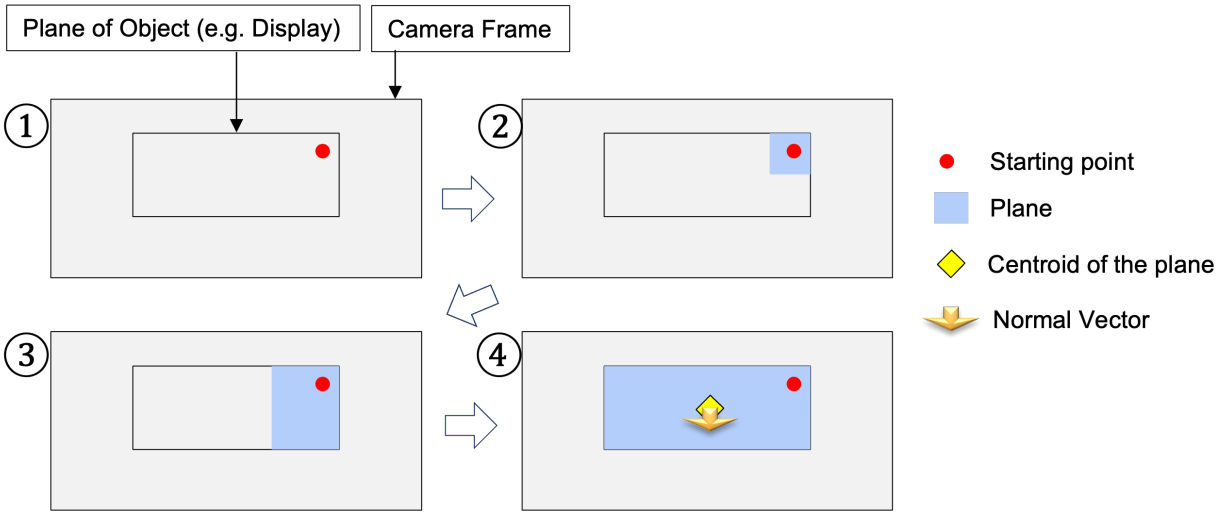

Figure 6.1: FloodFill algorithm how it works

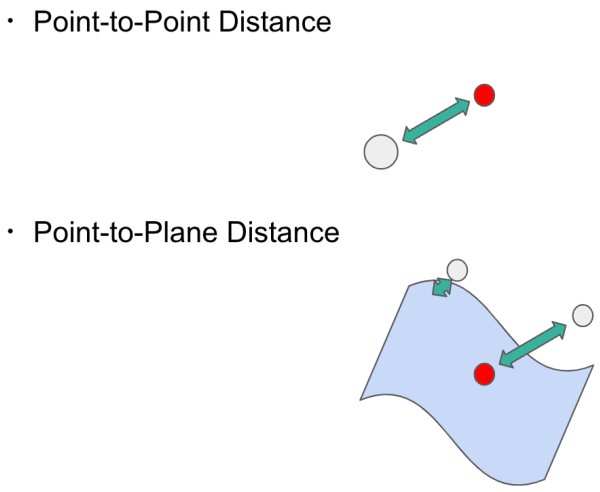

Figure 6.2: FloodFill algorithm conditions

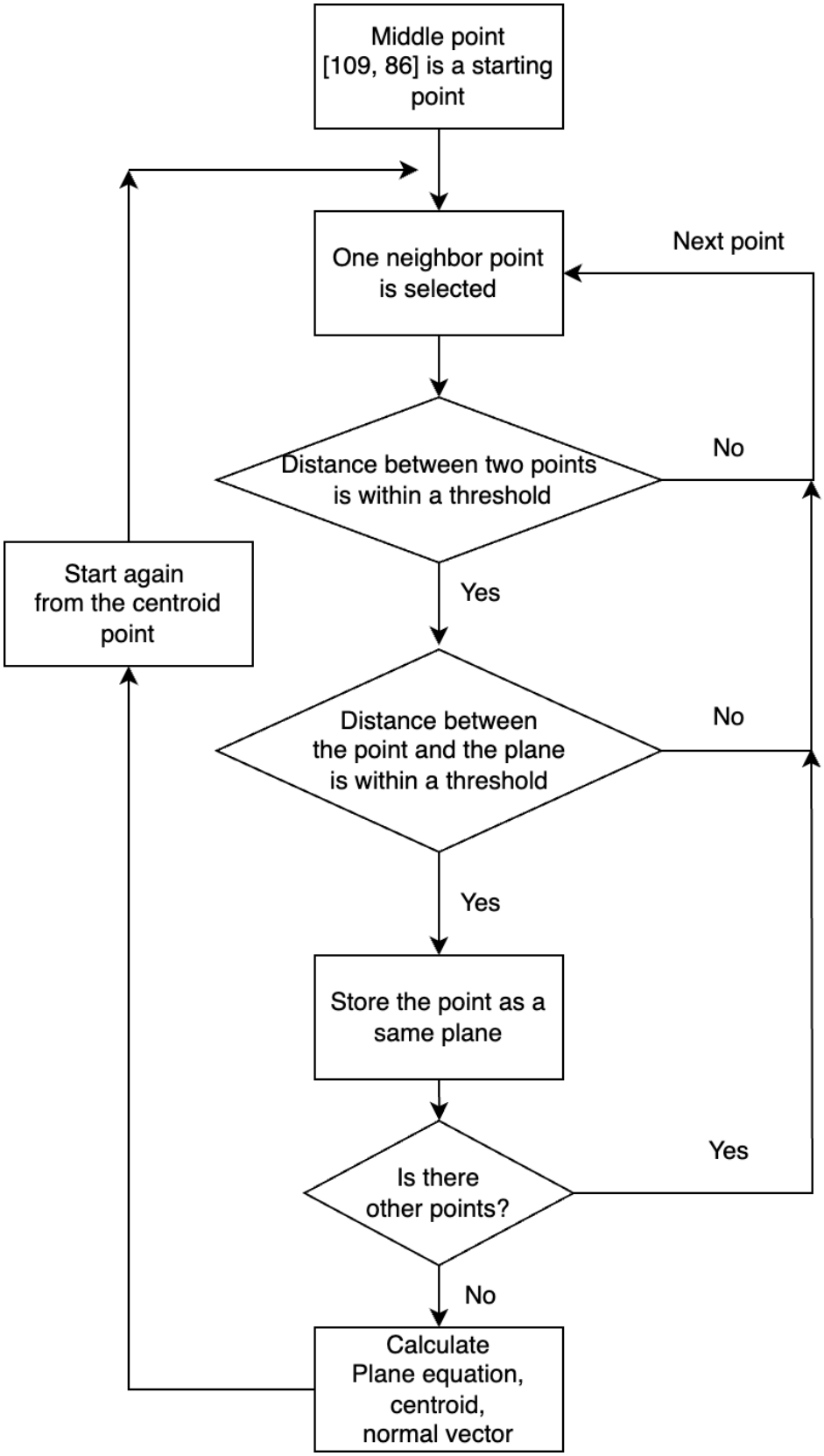

Figure 6.3: FloodFill FlowChart

$$C_y = \frac{y_1 + y_2 + \ldots + y_n}{n} \tag{6.2}$$

$$C_z = \frac{z_1 + z_2 + \ldots + z_n}{n} \tag{6.3}$$

n: Number of points on the same plane

- **Normal Vector of the plane**:
  From all the points on the same plane, construct a matrix A and a vector b.

$$A = \begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ \vdots & \vdots & \vdots \\ x_n & y_n & 1 \end{bmatrix} \tag{6.4}$$

$$b = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix} \tag{6.5}$$

  Next, using matrix A and vector b, compute the least squares solution. The least squares method is a statistical procedure to find the best fit line or plane for a set of data points [21]. The process for this calculation can be found in the reference [22, p. 182].

$$p = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix} = (A^T A)^{-1} A^T b \tag{6.6}$$

  The variable p represents the coefficients of the plane that best fits the point cloud.

  Also, the normal vector of the plane can be calculated using these coefficients by normalizing the vector, This calculation can make its length equal to 1.

$$\text{plane\_normal} = \frac{[p_1, p_2, -1]}{\sqrt{p_1^2 + p_2^2 + 1}} \tag{6.7}$$

$\sqrt{p_1^2 + p_2^2 + 1}$: Length of the vector (norm)

- **Plane Equation**:

  From the calculation above, the plane equation can be defined using the coefficient p.

$$p_1 * x + p_2 * y + p_3 = z \tag{6.8}$$

6.3 Getting 6DoF from this algorithm

This section has been independently developed by myself, with the aid of the floodfill algorithm.

As illustrated in Figure 6.4, the testing environment is situated in front of a display. As explained in the previous chapter, the floodfill algorithm can compute the normal vector from a captured plane. When the depth camera captures the display, the floodfill algorithm is capable of calculating the plane of the display itself and determining the normal vector from it.

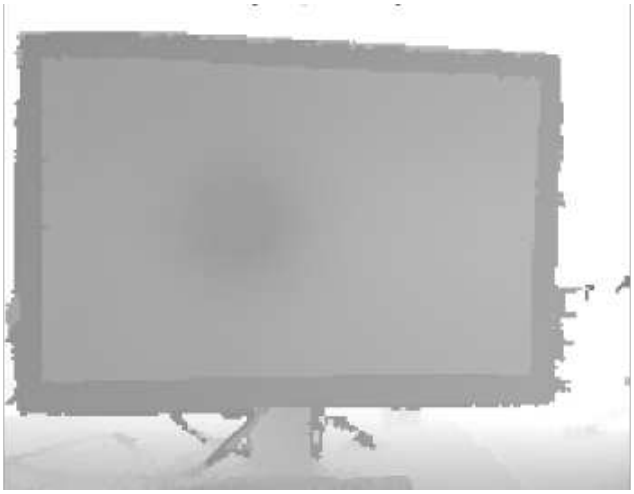

Figure 6.4: Testing environment taken by the depth camera

The relationship between the normal vector and other angles (pitch and yaw) is depicted in Figure 6.5. Theoretically, if the angle of the normal vector is known, other angles such as pitch and yaw can be calculated.

The conversion from the normal vector can be calculated as follows:

$$\text{pitch} = \arctan\left(\frac{z}{\sqrt{x^2 + y^2}}\right) \tag{6.9}$$

$$\text{yaw} = \arctan\left(\frac{y}{x}\right) \tag{6.10}$$

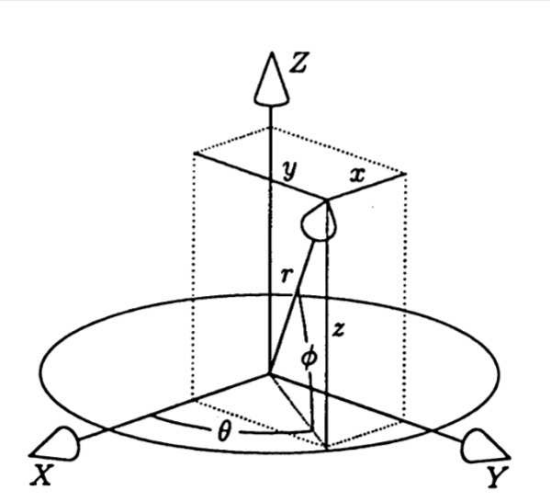

Figure 6.5: Relation between normal vector and other 3DoF. Here, ϕ represents pitch and θ represents yaw [23].

The arctangent of x is the inverse function of the tangent. This is also written as $\tan^{-1}(x)$. The reference for this calculation can be found in Appendix C, page 460, of the book by Grewal [23].

Also, this calculation can be written in MATLAB as:

```
%Pitch
pitch = atan2(-plane_normal.z, sqrt(plane_normal.x^2 + plane_normal.y^2));
%Yaw
yaw = atan2(plane_normal.y, plane_normal.x);
```

6.4 Results

Three experiments were conducted: front view, tilted position, and real-time. In these experiment, the accuracy and feasibility of calculating pitch and yaw from the normal vector were evaluated. The testing environment, situated in front of a display, is depicted in Figure 6.4.

6.4.1 First experiment - Front view -

Initially, the depth camera captured a display from the front.

The floodfill algorithm can gain the normal vector, the distance to the plane from the camera, and the plane equation. These results are summarized in Table 6.1.

With the given surface parameters, the equation for the plane can be formulated as follows:

Table 6.1: Values of Normal Vector and Surface Parameters

Parameters	**Values**
x (normalVector)	-0.18069809007501506
y (normalVector)	0.0712561498993067
z (normalVector)	-0.9809540057233923
distance	0.5681423946307732
p_1 (SurfaceParameters)	-0.18420648574829102
p_2 (SurfaceParameters)	0.07263964414596558
p_3 (SurfaceParameters)	0.5791733264923096

$$p_1 * x + p_2 * y + p_3 = z \tag{6.11}$$

where
$p_1 = -0.18420648574829102$,
$p_2 = 0.07263964414596558$,
$p_3 = 0.5791733264923096$.

In Figure 6.6, the image on the left depicts the plane result based on the plane equation. The middle image illustrates the position of the camera relative to the plane. The camera was situated directly in front of the display. The image on the right displays the calculated normal vector. The blue dot represents the direction of the vector, which is pointing toward the camera's direction.

The pitch and yaw were computed from the normal vector as follows:

$$\text{pitch} = \arctan\left(\frac{z}{\sqrt{x^2 + y^2}}\right) = -71.5902^\circ$$
$$\text{yaw} = \arctan\left(\frac{y}{x}\right) = 158.4788^\circ$$

where
$x = -0.18069809007501506$,
$y = 0.0712561498993067$,
$z = -0.9809540057233923$.

To validate these pitch and yaw calculations, the Figure 6.8 illustrates the actual location of the normal vector in the $\sqrt{x^2 + y^2}$ - z plane and x - y plane.

The vector angle in the $\sqrt{x^2 + y^2}$ - z plane matches the calculated pitch angle of -71.5902 degrees. Similarly, the vector angle in the x - y plane aligns with the calculated yaw angle of 158.4788 degrees. Overall, these figures suggest that the calculations for pitch and yaw are accurate.

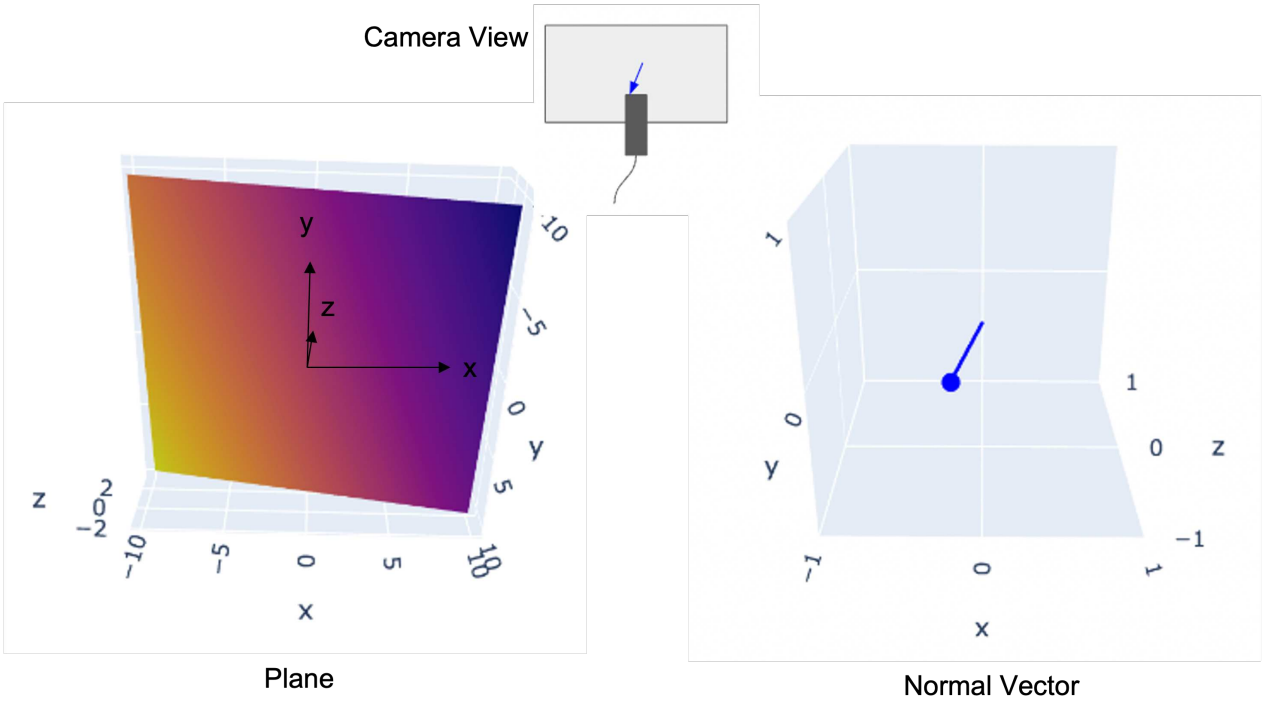

Figure 6.6: Results of the floodfill algorithm from front view

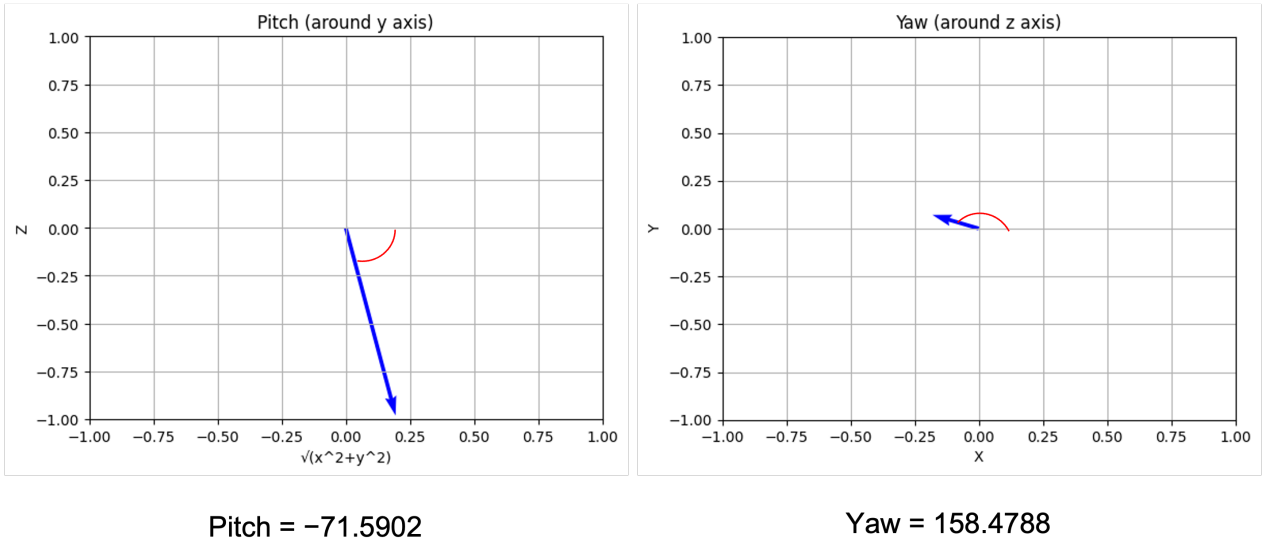

Figure 6.7: Actual locations in the $\sqrt{x^2 + y^2}$ - z plane and x - y plane

6.4.2 Second Experiment - Tilted position -

The second experiment is capturing images with the camera in a tilted position. In the first experiment, the normal vector was directed toward the camera's position. However, the normal vector is tilted toward the right in this case.

Figure 6.7 presents images of the plane, the camera view, and the normal vector.

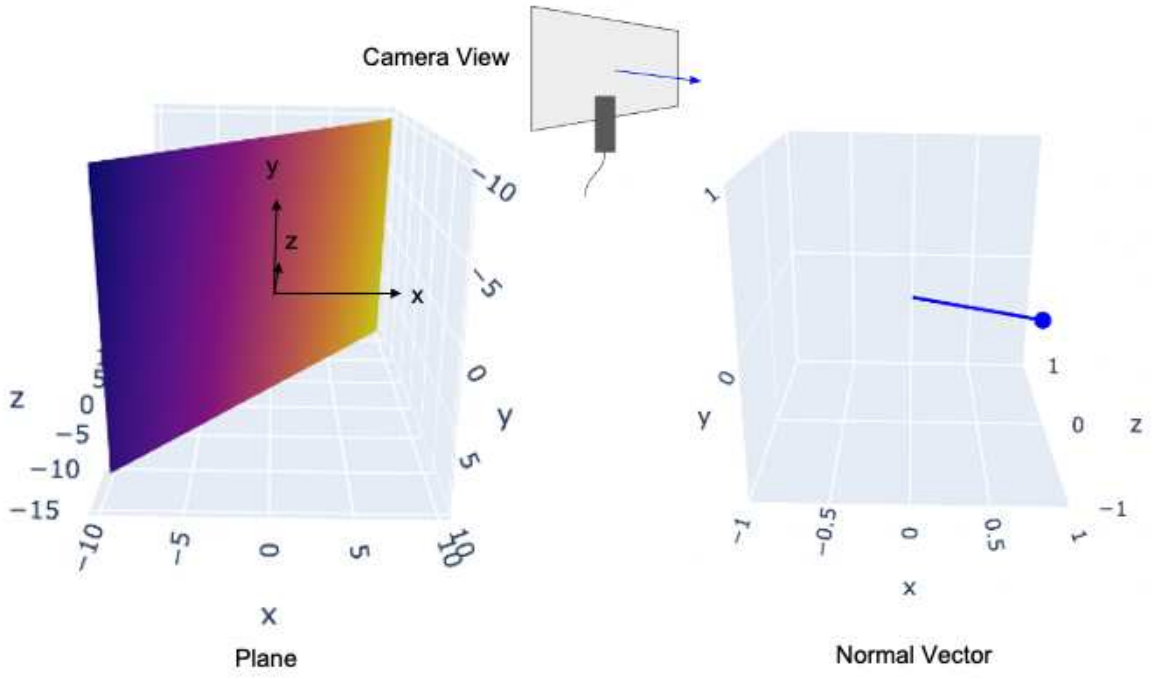

Figure 6.8: Results of the floodfill algorithm 2

The actual values of the plane and normal vector are in Table 6.2.

Table 6.2: Values of Normal Vector and Surface Parameter

Parameter	Value
x (normal vector)	0.8076119444238561
y (normal vector)	0.0848636675903085
z (normal vector)	-0.5835761348333566
distance	0.4041732896490905
p_1 (SurfaceParameters for x)	1.383901596069336
p_2 (SurfaceParameters for y)	0.14542004466056824
p_3 (SurfaceParameters for z)	0.6925802230834961

With the given surface parameters, the equation for the plane can be formulated as follows:

$$p_1 * x + p_2 * y + p_3 = z \tag{6.12}$$

where
$p_1 = 1.383901596069336$,
$p_2 = 0.14542004466056824$,
$p_3 = 0.6925802230834961$.

The calculated values for pitch and yaw are:

$$\text{pitch} = \arctan\left(\frac{z}{\sqrt{x^2 + y^2}}\right) = -35.7025^\circ$$
$$\text{yaw} = \arctan\left(\frac{y}{x}\right) = 5.9986^\circ$$

where
$x = 0.8076119444238561$,
$y = 0.0848636675903085$,
$z = -0.5835761348333566$.

To validate these pitch and yaw calculations, the Figure 6.9 illustrates the actual location of the normal vector in the $\sqrt{x^2 + y^2}$ - z plane and x - y plane.

The vector angle in the $\sqrt{x^2 + y^2}$ - z plane matches the calculated pitch angle of -35.7025 degrees. Similarly, the vector angle in the x - y plane aligns with the calculated yaw angle of 5.9986 degrees. Overall, these figures suggest that the calculations for pitch and yaw are accurate.

6.4.3 Third Experiment - Real-time -

To evaluate this pitch and yaw calculation algorithm, a real-time test was conducted. The camera moved in five directions. Figure 6.10 illustrates these five camera positions:

- Up - The camera lens is pointed toward the top of the display, potentially including a portion of the wall in the frame.
- Right - The camera is oriented to capture the right side of the display.
- Straight - The camera is positioned directly in front of the display. It covers all areas of the display.
- Left - The camera is focused on the left side of the display.
- Down - The camera lens is aimed downwards, potentially capturing a part of the ground as well as the display.

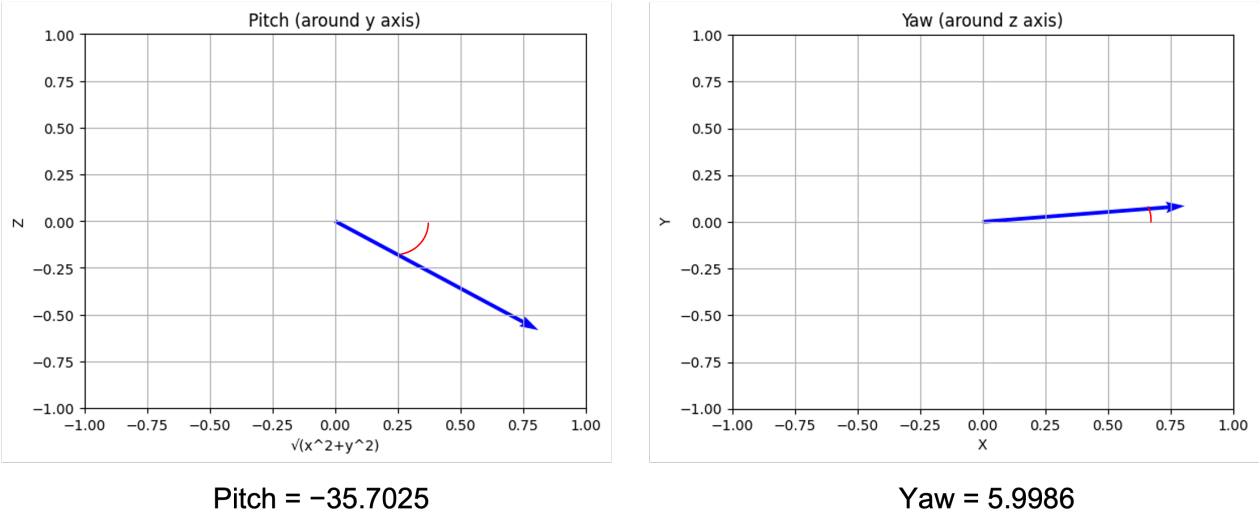

Figure 6.9: Actual locations in the $\sqrt{x^2 + y^2}$ - z plane and x - y plane

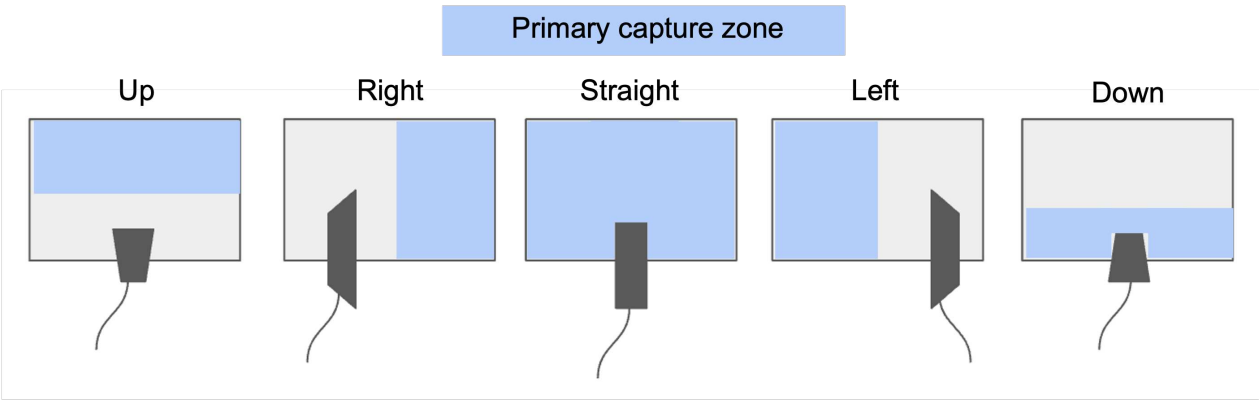

Figure 6.10: The camera position of this experiment

Table 6.3: Real-time Pitch and Yaw Calculation Results

	Up	Right	Straight	Left	Down
Pitch (around y axis)	62	42	72	50	70
Yaw (around z axis)	-84	-17	-160	-153	62

Table 6.3 contains the calculation results for pitch and yaw. The gray areas indicate that the values are expected to change according to the camera's movement.

- Pitch:
  The pitch angles should change when the camera moves up or down. There isn't a significant difference in pitch when the camera moves from a straight to a downward position. However, when the camera moves up, the angle changes by a negative 10 degrees. This result may not be reliable, as the calculated plane equation differed from the real environment. The floodfill algorithm needs further refinement.

- Yaw:
  The yaw angles should adjust when the camera moves left or right. When the camera moves left, the angle changes by +7 degrees. Conversely, when the camera moves right, the calculation indicates a change of +143 degrees. Similarly to the pitch calculation, these results might not be reliable due to the floodfill algorithm's current limitations in accurately estimating the correct plane. This will require further testing and updates.

This video illustrates the camera movement and floodfill algorithm results, including the normal vector, distance, surface parameters, pitch, yaw, plane depth image, and real-time estimated plane [24]. As shown in the video, the estimated plane still requires correction. Especially, when the camera moves to a higher position, the angle of the estimated plane shifts to the wrong position as shown in Figure 6.11.

6.4.4 Conclusion

Table 6.4 presents an overall feasibility assessment of the floodfill algorithm for calculating 6DoF.

- Translation along the X and Y axis: It is not feasible with this algorithm.

- Translation along the Z-axis (depth):
  It appears possible because the distance to the plane can be accurately determined.

- Roll (rotation around the x-axis):
  It might be calculable if the algorithm incorporates object detection features. If the camera can detect the edges of the display or plane, roll calculation might be enabled while tracking the movement of these edges or corners. However, in real-world scenarios, the object edges or corners are not always within the camera's field of view. The roll calculation is not feasible for this algorithm.

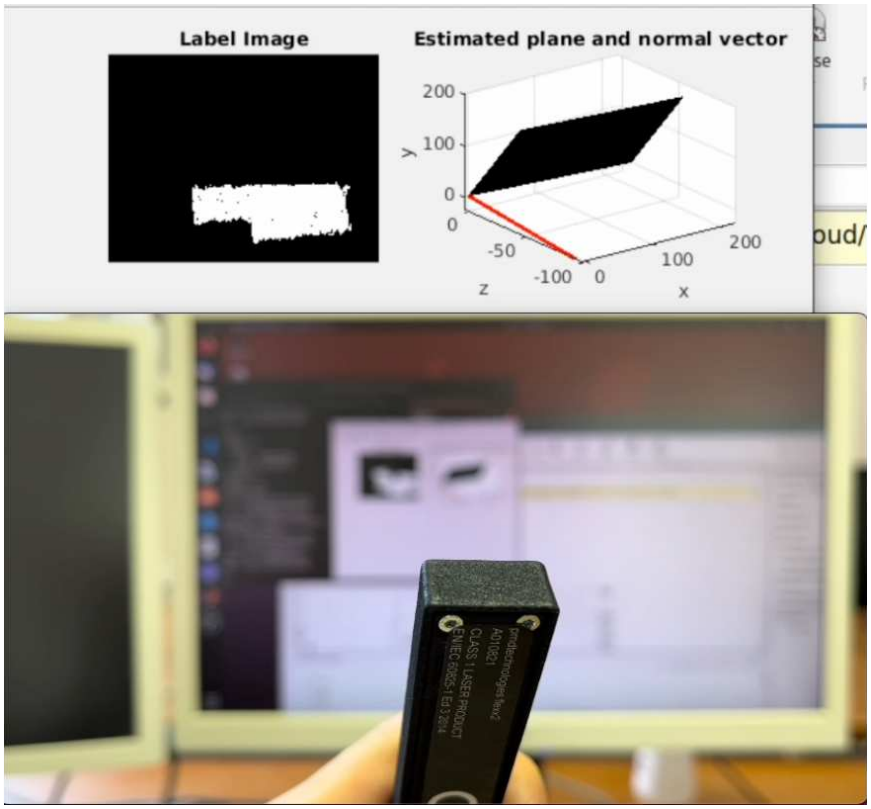

Figure 6.11: FloodFill Realtime camera and plane movement - When the camera moves up, the plane shifts to the opposite side

- Pitch and Yaw:
  It can be calculated with the theory previously explained. While the floodfill algorithm currently has minor issues with reliability, it is necessary to update and improve the algorithm to make real-time calculation possible.

Table 6.4: Results for FloodFill

6DoF	**FloodFill**
Translation X	×
Translation Y	×
Translation Z	○
Roll (around x axis)	×
Pitch (around y axis)	△
Yaw (around z axis)	△

△: still requires testing and updating but should be feasible.

CHAPTER 7

Conclusions and Future Work

Contents

In this thesis, three methodologies - the Iterative Closest Point (ICP) algorithm, optical flow, and the floodfill algorithm - were investigated for estimating the 6 Degrees of Freedom (6DoF). The objective was to validate their effectiveness and found out their capabilities and limitations for a relative navigation.

7.1 Conclusion

In conclusion, depth cameras can potentially estimate 6DoF using optical flow and floodfill algorithms. Table 7.1 summarizes the overall possibilities for calculating the 6DoF.

The Iterative Closest Point (ICP) algorithm performs very well when dealing with clean point cloud data such as those generated artificially. However, the algorithm encounters several challenges when applied to real-world scenarios. First of all, it struggles when dealing with noisy point clouds, and it requires a significant amount of computation time. It took about four to five minutes for calculating within two frames. As a result, real-time and accurate estimation of 6DoF becomes challenging. To overcome these limitations, it is necessary to employ additional techniques, such as object detection or deep learning. An example usage is in the state-of-the-art section. By using these methods, it would be possible to extract specific objects from the point clouds, reduce computation time, and improve estimation accuracy.

Table 7.1: Overall results

6DoF	ICP Real Data	RGB OpFl	Depth OpFl	FloodFill
Translation X	×	○	△	×
Translation Y	×	○	△	×
Translation Z	×	×	○	○
Roll (around x axis)	×	×	×	×
Pitch (around y axis)	×	×	×	△
Yaw (around z axis)	×	×	×	△

○: Possible, precise
△: Feasible, but need to be tested and updated
×: Not possible.

Optical flow using an RGB camera can calculate the x and y translations with a good degree of precision on a pixel basis. While this still needs to be converted from pixel to meter, it provides a precise amount of the movement of pixels in the x and y directions. After the experiment, it was clear that estimating the roll angle using only optical flow is challenging. An alternative approach would be integrating with object detection and calculate the tilt angles. This approach could fill the gap of roll calculation in this overall system.

Applying optical flow to depth images could potentially calculate the x, y, and z translations. The z translation can be accurately measured because of the depth camera. However, determining precise positions for the x and y translations remains challenging due to the camera's limited resolution and the presence of noise in the depth image.

The Floodfill algorithm can measure the z direction in real-time as it uses a depth camera. However, it is unable to calculate the x and y translations as it works with point clouds and this algorithm does not have any features needed for these calculations. Similarly, it cannot calculate roll because it depends on detecting the movement of the plane itself, which is difficult without state-of-the-art object or feature detection applications. The algorithm could potentially calculate pitch and yaw, but it still requires further development to improve precision. Overall, while the current version of the floodfill algorithm needs updating, it shows promise for future feasibility.

7.2 Future work

For future work, the uncertain sections in Table 7.1 (marked by △) need to have further real-time testing to check feasibility and improve accuracy. In addition to this, another roll calculation algorithm is also required to get full 6DoF. Precise object detection algorithm should be investigated. If all 6DoF can be accurately calculated in real-time, the next step would involve mounting the depth camera on a drone. This would enable the drone to self-determine its 6DoF and conduct manual operations such as screwing.

7.2.1 Potential Extension: Underwater Applications

The application of this algorithm in underwater environments might have additional challenges due to the blurring effects of water and wave motions. After achieving satisfactory results on land, the algorithm could then be adapted and optimized for underwater applications.

Bibliography

[1] Micha Schuster, David Bernstein, Paul Reck, Salua Hamaza, and Michael Beitelschmidt. Automated aerial screwing with a fully actuated aerial manipulator. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3340–3347, 2022. doi: 10.1109/IROS47612.2022.9981979.

[2] L.C. Inocencio, M.R. Veronez, Jr. da Silveira, L.G., F.M.W. Tognoli, L.V. de Souza, J. Bonato, and J.L. Diniz. 3-d reconstruction of rock samples via structure-from-motion for virtual reality applications: A methodological proposal. *Geosciences*, 13 (1):5, 2023. doi: 10.3390/geosciences13010005.

[3] Jingtao Sun, Yaonan Wang, Mingtao Feng, Danwei Wang, Jiawen Zhao, Cyrill Stachniss, and Xieyuanli Chen. Ick-track: A category-level 6dof pose tracker using inter-frame consistent keypoints for aerial manipulation. URL `https://www.ipb.uni-bonn.de/wp-content/papercite-data/pdf/sun2022iros.pdf`. This has not yet been published in other official publications.

[4] Zheng Fang and Sebastian Scherer. Real-time onboard 6dof localization of an indoor mav in degraded visual environments using a rgb-d camera. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5253–5259, 2015. doi: 10.1109/ICRA.2015.7139931.

[5] T. Stoyanov, M. Magnusson, H. Andreasson, and A. J. Lilienthal. Fast and accurate scan registration through minimization of the distance between compact 3d ndt representations. *The International Journal of Robotics Research*, 31(12):1377–1393, 2012. doi: 10.1177/0278364912460895.

[6] Kaustubh Pathak, Andreas Birk, Narunas Vaskevicius, and Jann Poppinga. Fast registration based on noisy planes with unknown correspondences for 3-d mapping. *Robotics, IEEE Transactions on*, 26:424 – 441, 07 2010. doi: 10.1109/TRO.2010.2042989.

[7] F. J. Perez-Grau, F. Caballero, A. Viguria, and A. Ollero. Multi-sensor three-dimensional monte carlo localization for long-term aerial robot navigation. *International Journal of Advanced Robotic Systems*, 14(5), 2017. doi: 10.1177/1729881417732757.

[8] F. J. Pérez-Grau, F. R. Fabresse, F. Caballero, A. Viguria, and A. Ollero. Long-term aerial robot localization based on visual odometry and radio-based ranging. In *2016 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 608–614, 2016. doi: 10.1109/ICUAS.2016.7502653.

[9] My. Liu, Y. Wang, and L. Guo. 6-dof motion estimation using optical flow based on dual cameras. *J. Cent. South Univ.*, 24:459–466, 2017. URL https://doi.org/10.1007/s11771-017-3448-2.

[10] Yanling Hao, Zhilan Xiong, Feng Sun, and Xiaogang Wang. Comparison of unscented kalman filters. volume 3, pages 895 – 899, 09 2007. doi: 10.1109/ICMA.2007.4303664.

[11] Arindam Roychoudhury, Marceli Missura, and Maren Bennewitz. Plane segmentation using depth-dependent flood fill. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2210–2216, 2021. doi: 10.1109/IROS51168.2021.9635930.

[12] Arindam Roychoudhury, Marcell Missura, and Maren Bennewitz. Plane segmentation in organized point clouds using flood fill. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 13532–13538, 2021. doi: 10.1109/ICRA48506.2021.9561325.

[13] Fangwen Shu, Yaxu Xie, Jason Rambach, Alain Pagani, and Didier Stricker. Visual slam with graph-cut optimized multi-plane reconstruction, 2022.

[14] pmdtechnologies ag. pmd flexx2 3d camera development kit. https://3d.pmdtec.com/en/3d-cameras/flexx2/. Accessed: June 18, 2023.

[15] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo. *Robotics: Modelling, Planning and Control*. Springer, 2009.

[16] Kevin Chavez, Ben Cohen-Wang, Garrick Fernandez, Noah Jackson, and Will Lauer. Lecture on optical flow and its applications. URL http://vision.stanford.edu/teaching/cs131_fall1718/files/17_notes.pdf. Accessed: 2023-07-09.

[17] Video link: Optical flow for rgb images in x direction, . URL https://drive.google.com/file/d/18XmLl3qXQ3_sp6FcnJ2EZz0hNUlUJpso/view?usp=sharing.

[18] Video link: Optical flow for rgb images in y direction, . URL https://drive.google.com/file/d/1xVdlC6kBIMpuv43xA4VOeXXXVkEFA1oR/view?usp=sharing.

[19] Neeta Nemade and Vinaya Gohokar. Comparative performance analysis of optical flow algorithms for anomaly detection. *SSRN Electronic Journal*, 01 2019. doi: 10.2139/ssrn.3419775.

[20] Video link: Optical flow for depth images in x and y direction (z direction result is also included), . URL https://drive.google.com/file/d/1xv1MShuvbx8QaQnM4wGCr-aRf_xMR27p/view?usp=sharing.

[21] Will Kenton. Least squares method: What it means, how to use it, with examples, 13 June, 2023. URL https://www.investopedia.com/terms/l/least-squares-method.asp.

[22] Gilbert Strang. *Linear Algebra and Its Applications*. Cengage Learning, 4th edition, 2005.

[23] M. S. Grewal, L. R. Weill, and A. P. Andrews. *Global Positioning Systems, Inertial Navigation, and Integration*. 2007. URL https://onlinelibrary.wiley.com/doi/book/10.1002/0470099720.

[24] Video link: Floodfill algorithm testing with camera movement and the estimated plane, . URL https://drive.google.com/file/d/1acJMOdnFYf0hyWa7S0lC5K2CwY-mWyOq/view?usp=sharing.

[25] Shinji Oomori, Takeshi Nishida, and Shuichi Kurogi. Point cloud matching using singular value decomposition. *Artificial Life and Robotics*, 21:149–154, 06 2016. doi: 10.1007/s10015-016-0265-x.

[26] Icp svd algorithm code. URL https://github.com/AtsushiSakai/MATLABRobotics/blob/master/SLAM/ICP/ICPsample.m.

APPENDIX A

Other considerations

A.1 ICP algorithm in detail

1. **Preprocessing and Loading point clouds**: First, the script loads data from two .fig files (data1 and data2). Then, this program create a mask which can filter out the points where z coordinate is over 4 m. This mask can reduce the computational costs and ignore noise in the background.

2. **ICP Matching Loop**: This loop keeps continuing until the total error (sum of distances between data1 and data2) is less than 0.001 or a maximum number of iterations is reached.

    - Find nearest points: For each point in data1, find the closest point in data2 and calculate the error (sum of distances).
    - Estimate motion: Use the Singular Value Decomposition (SVD) method to estimate the motion (rotation and translation) that best aligns the matched nearest points.
    - Update transformation: The estimated transformation is applied for the ICP calculation with data1. The rotation and translation matrices are updated and the calculated points will move closer to data2 with each iteration.

The key function within this ICP algorithm is the Singular Value Decomposition (SVD) motion estimation.

This algorithm follows these steps [25]:

1. Compute Nearby Points: Calculate the point at post-movement position a_i that is closest to the initial value b_i

$$\mathbf{a}_i = \text{argmin}_{\mathbf{a}\in \text{A}} ||\mathbf{a} - \mathbf{b}_i||_2 \tag{A.1}$$

2. Calculate the centroid of those two points

$$\bar{\mathbf{a}} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{a}_i \tag{A.2}$$

$$\bar{\mathbf{b}} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{b}_i \tag{A.3}$$

3. Compute covariance matrix

$$\mathbf{W} = \sum_{i=1}^{n} (\mathbf{b}_i - \bar{\mathbf{b}})(\mathbf{a}_i - \bar{\mathbf{a}})^T \tag{A.4}$$

4. Singular value decomposition of the covariance matrix

$$\mathbf{W} = \mathbf{U}\mathbf{A}\mathbf{V}^T \tag{A.5}$$

$$A = \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_n \end{bmatrix} \tag{A.6}$$

5. Rotation matrix and translation vector can be calculated based on U and V obtained from this. U and V calculation can be done by svd function on Matlab.

```
[U,A,V] = svd(W);
```

$$\mathbf{R} = \mathbf{U}\mathbf{V}^T \tag{A.7}$$

$$\mathbf{t} = \bar{\mathbf{b}} - \mathbf{R}\bar{\mathbf{a}} \tag{A.8}$$

6. Convergence judgment based on error. repeat the above until convergence

$$\text{if } |\text{error}_i - \text{error}_{i-1}| < \epsilon \text{ then converge} \tag{A.9}$$

A.2 FloodFill algorithm in detail

A.2.1 Threshold - point to point distance -

Here is the details of threshold and how distances can be calculated to determine if both points are on the same plane.

The threshold is computed as:

$$t = point_{distance} \times f \times point(3) \tag{A.10}$$

where

$point_{distance} = 0.0047m$

$f = 1.8$ (some factor)

$point(3)$: points in the z-direction, depth

The threshold is not constant. Initially, $point_{distance} = 0.0047m$ is set based on the calculation of the camera's field of view and resolution. This value is known as spatial resolution. In the camera's case, 0.0047 m is the minimum distance between two points at which these points can be separated and distinguished.The coefficient $f = 1.8$ is set by the author and adjusted depending on the testing environment.

As depth increases, the threshold also increases. This is because, points that are further away from the camera are spaced more distantly from each other.

A.2.2 Threshold - point to plane distance -

The threshold for point to plane distance is calculated as:

$$t = f \times 3 \times \sigma \times \text{point}(3) \tag{A.11}$$

where

$\sigma = 0.01$

$f = 1$ (some factor)

$point(3)$: points in the z-direction, depth

The parameters σ and f are set by the author based on the camera settings and the environment. σ is set with the intent of eliminating errors in depth measurement. It can account for 1% differences in depth measurement errors. Similar to the "point-to-point distance" case, the threshold increases as the depth increases for the same reason.

Appendix B

Source code

B.1 ICP Algorithm code

This code is modified and adapted the code to fit my specific 3D scenario, using this GitHub code as a reference [26].

```
%ICP algorithm for estimating 6DoF
function []=ICP()
    close all;
    clear all;

    % Get original point clouds
    data1 = openfig('A4first_place.fig', 'invisible');
    data1 = gcf;
    axObjs = data1.Children;
    dataObjs = axObjs.Children;
    x = dataObjs(1).XData;
    y = dataObjs(1).YData;
    z = dataObjs(1).ZData;

    % Get point clouds within this mask
    mask = z < 4;

    % Apply the mask
    x_filtered = x(mask);
    y_filtered = y(mask);
    z_filtered = z(mask);
```

```
22
23      % Original point clouds within the mask
24      data1 = [x_filtered; y_filtered; z_filtered];
25
26      % Get target point clouds
27      data2 = openfig('A4second.fig', 'invisible');
28      data2 = gcf;
29      axObjs2 = data2.Children;
30      dataObjs2 = axObjs2.Children;
31      x_2 = dataObjs2(1).XData;
32      y_2 = dataObjs2(1).YData;
33      z_2 = dataObjs2(1).ZData;
34
35      % Create a mask
36      mask2 = z_2 < 4;
37
38      % Apply the mask
39      x2_filtered = x_2(mask2);
40      y2_filtered = y_2(mask2);
41      z2_filtered = z_2(mask2);
42
43      %Target point clouds with the mask
44      data2 = [x2_filtered; y2_filtered; z2_filtered];
45
46      % Run ICP algorithm to estimate motion
47      [R,T] = ICPmatch(data2, data1);
48
49      % Display motion estimation results
50      disp('Estimated Motion [m m m deg deg deg]:')
51      Est=[T' rad2deg(rotm2eul(R))]
52
53      % Plot results
54      figure(1);
55      set(gca, 'fontsize', 16, 'fontname', 'times');
56      plot3(data1(1,:),data1(2,:),data1(3,:),'.b');hold on;
57      plot3(data2(1,:),data2(2,:),data2(3,:),'.g');hold on;
58
59      z=R*data1+T;
60
61      plot3(z(1,:),z(2,:),z(3,:),'.r');hold off;
62
63      xlabel('X (m)', 'fontsize', 16, 'fontname', 'times');
64      ylabel('Y (m)', 'fontsize', 16, 'fontname', 'times');
```

```
65      zlabel('Z (m)', 'fontsize', 16, 'fontname', 'times');
66      legend('Data (t-1)', 'Data (t)','ICP Matching Result');
67      grid on;
68      axis([-9 13 -9 18 -1 11]);
69
70
71  function [R, t]=ICPmatch(data1, data2)
72
73      preError=0; %Error value of the previous iteration
74      ErrorD=1000; %Error difference
75      epsilon=0.001; %Convergence
76      max_iteration=100;
77      count=0;
78
79      R=eye(3); %Rotation matrix
80      t=zeros(3,1); %Translation vector
81
82      while ~(ErrorD < epsilon)
83          count=count+1;
84
85          [ii, error]=FindNearestPoint(data1, data2); %Find nearest points
86          [R1, t1]=SVDmotion(data1, data2, ii); %Movement estimation using SVD
87
88          % Update R and T using the calculated value
89          data2=R1*data2;
90          data2=[data2(1,:)+t1(1); data2(2,:)+t1(2); data2(3,:)+t1(3)];
91          R = R1*R;
92          t = R1*t + t1;
93
94          ErrorD=abs(preError-error);
95          preError=error; %Save previous error
96
97          if count > max_iteration %when fails to achieve convergence
98              disp('Max Iteration');return;
99          end
100     end
101
102 %Find nearest points between data1 and data2
103 function [index, error]=FindNearestPoint(data1, data2)
104
105     % Number of points (data1: 38497, data2: 38136)
106     m1=size(data1,2);
107     m2=size(data2,2);
```

```
    index=[];
    error=0;

    for i=1:m1 %All points
        dx=(data2-repmat(data1(:,i),1,m2));
        dist=sqrt(dx(1,:).^2+dx(2,:).^2+dx(3,:).^2); %Calculate distance
        [dist, ii]=min(dist);
        index=[index; ii];
        error=error+dist;
    end

%Compute the centroid of each point cloud
function [R, t]=SVDmotion(data1, data2, index)
    M = data1;
    mm = mean(M,2);
    S = data2(:,index);
    ms = mean(S,2);

    Sshifted = [S(1,:)-ms(1); S(2,:)-ms(2); S(3,:)-ms(3)];
    Mshifted = [M(1,:)-mm(1); M(2,:)-mm(2); M(3,:)-mm(3)];

    W = Sshifted*Mshifted';

    [U,A,V] = svd(W);

    R = (U*V')';
    t = mm - R*ms;
```

B.2 Optical Flow for RGB image

```
% Get video from the camera
cam = webcam();

% Define frame size
frameSize = [480 640];

% Create an optical flow object
opticFlow = opticalFlowLK('NoiseThreshold', 0.05);

% Initialize the camera's position and posture
pos = [0; 0; 0]; % position [x; y; z]
rot = eye(3); % rotation matrix
```

```
13
14 % Infinite loop
15 while true
16     % Get an image from the camera
17     frame = snapshot(cam);
18
19     % Resize the image
20     frame = imresize(frame, frameSize);
21
22     % Convert the image to grayscale
23     frameGray = rgb2gray(frame);
24
25     % Estimate optical flow
26     flow = estimateFlow(opticFlow, frameGray);
27
28     % Display arrows
29     imshow(frameGray);
30     hold on;
31     plot(flow, 'DecimationFactor', [5 5], 'ScaleFactor', 16);
32     hold off;
33     drawnow;
34
35     % Calculate the movement amount from the optical flow
36     moveVec = [mean(flow.Vx(:)); mean(flow.Vy(:))];
37     pos = pos + [moveVec(1); moveVec(2); 0]; % update the position
38
39     % Display the camera's position and posture
40     disp('Camera Position:');
41     disp(pos);
42
43 end
44
45 % Close the camera
46 clear cam;
```

B.3 Optical flow for depth image with non-data elimination method

```
1 % ROS subscriber for the depth image topic
2 depthImgSub = rossubscriber('/royale_camera_driver/depth_image');
3
```

```
% Define frame size
frameSize = [480, 640];

% Create optical flow object / There are two methods
%opticFlow = opticalFlowLK('NoiseThreshold', 1.5);
opticFlow = opticalFlowHS;

% Define figures and axes
myDepthsFigure = figure(1); hold on; myDepthsAxes = axes(myDepthsFigure);
myGradientFigure = figure(2); hold on; myGradientAxes = axes(myGradientFigure);

% Initialize the camera's position and posture
pos = [0; 0; 0]; % position [x; y; z]
rot = eye(3); % rotation matrix

% Loop to process the depth images in real-time
while true
 % Receive a message from the subscriber
 depthImgMsg = receive(depthImgSub);

  % Convert the ROS image message to a MATLAB image
 depthImage = readImage(depthImgMsg);

 % Get the depth value (z) at a specific pixel
 row = 112;
 col = 86;
 depthZ = depthImage(row, col);

 % Fill missing values in the depth image
 depthImage = fillDepthImage(depthImage);
 imshow(depthImage,'Parent',myDepthsAxes);
 title(myDepthsAxes,'Only depth image');

 %Median and Gaussian Filters
 %depthImage = medfilt2(depthImage);
 %depthImage = imgaussfilt(depthImage, 1);

 % Resize depth image to match frame size
 resizedDepthImage = imresize(depthImage, frameSize);

  % Calculate depth gradient
 [Gx, Gy] = imgradientxy(resizedDepthImage);

```

```
 % Create a gradient magnitude image
 gradMagnitude = sqrt(Gx.^2 + Gy.^2);

  % Create a gradient magnitude image
 gradMagnitudeX = sqrt(Gx.^2);
 gradMagnitudeY = sqrt(Gy.^2);

  % Estimate optical flow on the gradient magnitude image
 flow = estimateFlow(opticFlow, gradMagnitude);

  % Visualization
 step = 13; % Step size
 [X, Y] = meshgrid(1:step:frameSize(2), 1:step:frameSize(1));
 flow_Vx = flow.Vx(1:step:end, 1:step:end);
 flow_Vy = flow.Vy(1:step:end, 1:step:end);

 imshow(gradMagnitude,'Parent',myGradientAxes);
 hold(myGradientAxes, 'on');
 quiver(myGradientAxes, X, Y, flow_Vx, flow_Vy, 'r', 'LineWidth', 1);
 hold(myGradientAxes, 'off');
  axis tight;
 axis off;
 title(myGradientAxes,'Optical Flow on Depth Gradient Magnitude');
  disp('Estimated translation vector:');

  % Calculate the movement amount from the optical flow
  moveVec = [mean(flow.Vx(:)); mean(flow.Vy(:))];
  pos = pos + [moveVec(1); moveVec(2); 0]; % update the position

   % Display the camera's position and posture
  disp('Camera Position:');
  disp(pos);
  disp(depthZ);

  % Pause for image updates
  pause(0.1);
end

function filledImage = fillDepthImage(depthImage)

  % Add NaN padding
  depthImagePad = nan(size(depthImage) + 2);

```

```
%Insert depth image inside of the NaN padding
depthImagePad(2:end-1, 2:end-1) = depthImage;

 % Create a copy
filledImage = depthImagePad;

 % Get the size of the image
[height, width] = size(filledImage);

 % Initialize a flag for first step
zerosReplaced = true;

 while zerosReplaced %while true
    % Assume no zeros will be replaced during this pass
    zerosReplaced = false;

    % Loop over each pixel in the depth image
    for i = 2:height-1
        for j = 2:width-1
            % If the pixel is a zero, replace it with the minimum value of the neighbors
            if filledImage(i, j) == 0
                % Start with a 3x3 neighborhood
                neighbors = filledImage(i-1:i+1, j-1:j+1);
                %Valid neighbor points are non NaN and non zero
                validNeighborpoints = neighbors(~isnan(neighbors) & neighbors ~= 0);

                % If no non-zero neighbors, expand to 5x5 neighborhood
                if isempty(validNeighborpoints) && i > 2 && j > 2 && i < height - 1 && j < width - 1
                    %Do the same thing
                    neighbors = filledImage(i-2:i+2, j-2:j+2);
                    validNeighborpoints = neighbors(~isnan(neighbors) & neighbors ~= 0);
                end

                % Replace zero with minimum of non-zero neighbors if available
                if ~isempty(validNeighborpoints)
                    filledImage(i, j) = min(validNeighborpoints);
                    zerosReplaced = true;
                end
```

```
128                 end
129             end
130         end
131     end
132      % Remove NaN padding
133     filledImage = filledImage(2:end-1, 2:end-1);
134 end
```