



Otto-von-Guericke-University Magdeburg

## Faculty of Computer Science

Institute for Intelligent Cooperating Systems

Comparison of Real-Time Plane Detection  
Algorithms on Intel RealSense

# Bachelor Thesis

Author:

Lukas Petermann

Examiner:

Prof. Frank Ortmeier

2nd Examiner:

M.Sc. Marco Filax

Supervisor:

M.Sc. Maximilian Klockmann

Magdeburg, 32.13.2042

# Contents

<b>List of Figures</b>	<b>4</b>
<b>1 Introduction</b>	<b>5</b>
1.1 Real-Time Plane Detection . . . . .	5
1.2 Intel RealSense . . . . .	6
<b>2 Background</b>	<b>7</b>
2.1 SLAM . . . . .	7
2.2 Intel Realsense . . . . .	8
2.3 Plane Detection . . . . .	9
2.4 Plane Detection Algorithms . . . . .	10
2.4.1 Robust Statistics approach for Plane Detection . . . . .	10
2.4.2 Oriented Point Sampling . . . . .	12
2.4.3 3D-KHT . . . . .	12
2.4.4 OBRG . . . . .	13
2.4.5 PEAC - Probabilistic Agglomerative Hierarchical Clustering . . . . .	15
2.4.6 CAPE - Fast Cylinder and Plane Extraction . . . . .	15
2.4.7 SCH-RG - Plane Extraction using Spherical Convex Hulls . . . . .	15
2.4.8 D-KHT - Hough Transform for Real-Time Plane Detection . . . . .	15
2.4.9 DDFF - Depth Dependent Flood Fill . . . . .	15
2.4.10 PlaneNet . . . . .	15
2.4.11 PlaneRecNet . . . . .	15
2.4.12 PlaneRCNN . . . . .	15
2.5 Data Formats . . . . .	15
2.5.1 Common Input Types . . . . .	15
2.5.2 Common Plane Formats . . . . .	16
2.6 Datasets . . . . .	16
2.6.1 2D-3D-S . . . . .	16
2.7 Evaluation Metrics . . . . .	17
<b>3 Concept</b>	<b>19</b>
3.1 Selection of Plane Detection Algorithms . . . . .	20
3.1.1 Criteria . . . . .	20
3.1.2 Plane Detection Algorithms . . . . .	21

*Contents*

---

3.2 Datasets . . . . .	23
3.3 Definition Real-Time . . . . .	24
3.4 Summary . . . . .	25
<b>4 Implementation</b>	<b>26</b>
4.1 System Setup . . . . .	26
4.2 Plane Detection Algorithms . . . . .	26
4.2.1 RSPD & OPS . . . . .	26
4.2.2 3D-KHT . . . . .	27
4.2.3 OBRG . . . . .	27
4.3 2D-3D-S . . . . .	28
4.4 FIN Dataset . . . . .	29
<b>5 Evaluation</b>	<b>31</b>
5.1 Protocol . . . . .	31
5.1.1 Metrics . . . . .	31
5.1.2 Parameterization of Algorithms . . . . .	32
5.2 Results . . . . .	35
5.2.1 2D-3D-S . . . . .	35
5.2.2 FIN . . . . .	38
5.2.3 Comparison . . . . .	41
5.3 Summary . . . . .	41
<b>6 Conclusion and Future Work</b>	<b>42</b>
<b>Bibliography</b>	<b>43</b>

# List of Figures

2.1	RTAB-MAP Block Diagram . . . . .	8
2.2	Hough Transform Accumulators . . . . .	13
3.1	Concrete Concept Graphic . . . . .	19
3.2	Dynamic Datasets . . . . .	24
4.2	Dynamic Ground Truth Generation . . . . .	30
5.1	Time per Cloud size 2D-3D-S . . . . .	37
5.2	Time Results Hallway . . . . .	40

# 1 Introduction

Man-made environments usually contain planar structures to a large extent. They are a central component in numerous use cases in the fields of Augmented and Virtual Reality, as well as robotics.

## 1.1 Real-Time Plane Detection

### **introduction**

1. aktueller stand: gute und schnelle ebenenfindung wird oft gebraucht, gibt es auch schon/ist möglich
2. problem: oft sind die speziellen sensoren sehr kostenspielig
3. Daher die frage: (wie gut) ist das ganze auf bezahlbarer hardware möglich?
4. Nötig, um die Frage zu beantworten:
  - Welche Kamera(s)?
  - Welcher algorithmus?
  - Was heisst "real-time" überhaupt?
5. Problem an letzterem: nicht möglich einen einheitlich besten algorithmus auszuwählen, da ...
6. Lösung: wir wählen algorithmen aus und vergleichen diese einheitlich um die frage aus 3. zu beantworten

## 1.2 Intel RealSense

- Wir müssen zuerst sensoren auswählen, mit denen wir die umgebung aufnehmen
- Da, wie vorher angesprochen, der preis des sensors oft ein problem ist, wählen wir eine relativ billige (im vergleich)
- die intel sensoren sind vergleichsweise bezahlbar.
- genauer gesagt nutzen wir ...
- Zu den sensoren wird eine kostenfreie software bereit gestellt
- Über diese software lassen sich die kameras ansteuern. dazu ist in dieser software ein slam algorithmus namens rtabmap implementiert
- mit rtabmap können wir den strom aus rohdaten zu einer bestehenden karte verarbeiten, was uns ermöglicht ebenen der kompletten umgebung zu finden anstatt nur von dem aktuellen blickwinkel

# 2 Background

In this chapter, we present relevant literature needed to completely understand the proposed concept of chapter 3.

## 2.1 SLAM

SLAM (Simultaneous Localization And Mapping) algorithms aim to solve a usual problem in the field of unmanned robotics; A robot finds itself in an unknown environment and attempts to build a coherent map while keeping track of its location. The robot uses use-case-specific sensors to obtain a snapshot of its current surroundings, which it then uses to update and enhance its known map (Mapping). The robot then attempts to accurately estimate its position based on the updated map. The new information about its position is processed during the next map update. Over decades of research, varieties of different (combinations of) sensors have been employed to solve this problem more accurately and efficiently. Internal odometry sensors alone can be unreliable if the robot moves over uneven or slippery surfaces. For that reason, visual SLAM(V-SLAM) methods like *MonoSLAM*[6] or *Dense Visual SLAM*[13] integrate additional visual input of camera sensors into their algorithm.

### RTAB-MAP

RealSense-ROS internally uses a SLAM algorithm for map building, namely RTAB-MAP (Real-Time Appearance-Based Mapping)[14]. Unlike purely visual-based SLAM algorithms, RTAB-MAP also takes input from odometry sensors, as well as an optional additional input in form of two- or three-dimensional lidar scan. All these inputs are combined during a synchronization step, and the results thereof are passed to RTAB-MAP's *Short-Term-Memory* (STM). The STM assembles a new node from the new inputs and inserts it into the map graph. Based on the newly inserted node, RTAB-MAP attempts to determine if the current location has already been visited earlier, also known as *loop closure*. If a loop closure is detected, i.e., RTAB-MAP detects the re-visiting of a known location, the map graph is optimized and thus minimized. In addition, the global map is reassembled in correspondence with the new information.

The resulting map is published in the form of an unorganized point cloud. RTAB-MAP's general workflow is shown below in Figure 2.1:

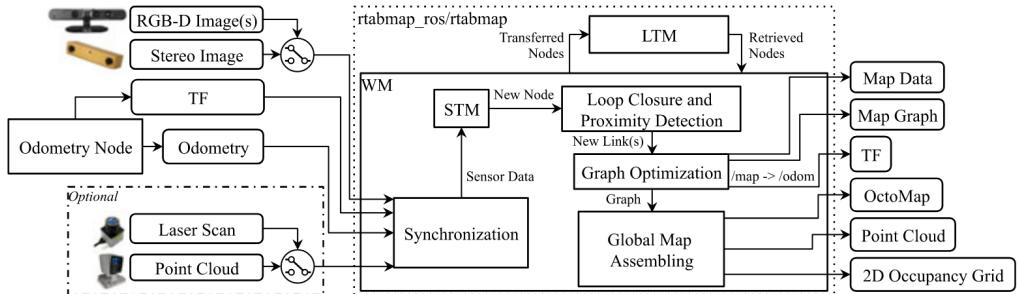


Figure 2.1: Block diagram of RTAB-MAP's main node. Taken from [14, Figure 1]

## 2.2 Intel Realsense

In this work, we use the Intel RealSense tracking camera T265 and the RGB-Depth(RGB-D) camera D455. A tracking camera is generally used to observe the environment and usually has a wider field of view (FOV). The primary motivation for using RGB-D cameras is depth perception. The primary differences and similarities between the T265 and the D455 are reported in Table 2.1. Beide Kameras sind stereo, die T265 hat 2 fisheye lenses und die D455 hat 2 imagers. Dazu hat die D455 noch einen RGB sensor und einen infrarot sensor. Mit dem IR sensor und den beiden imagern wird ein tiefenbild berechnet. Durch die fisheye lenses hat die T265 mit 163° ein deutlich breiteres Sichtfeld als die D455 mit nur 111°. Die maximale FPS anzahl der D455 ergibt sich aus den individuellen FPS werten der imager sensoren und dem RGB sensor, welche beide einen maximalwert von 90 haben. Dazu sei gesagt, dass bei steigender auflösung die maximale Framerate sinkt und 90FPS nur mit einer maximalen auflösung von 640x480 möglich ist. Furthermore, both cameras have an integrated Inertial Measurement Unit (IMU) which is used to compute its position in combination with visual input.

Intel provides a software development kit, namely RealSense SDK, which allows easy and efficient use of the cameras. The SDK runs on both Windows and Ubuntu, and a ROS(Robot Operating System <sup>1</sup>) adaptation is also provided in Intel's Github repository <sup>2</sup>.

<sup>1</sup><https://www.ros.org/>

<sup>2</sup><https://github.com/IntelRealSense/realsense-ros>

	Image	Type	max. Resolution	D-FOV	Shutter	Price	max. FPS
D455	Stereo	RGB-D	1280x720	111°	global	419\$	90
T265	Stereo	Tracking	848 x 800	163°	global	199\$	30

Table 2.1: Intel RealSense T265 and D455 camera specifications. More information and the complete Datasheets can be found on <https://www.intelrealsense.com/>.

## 2.3 Plane Detection

**introduction** The field of plane detection has been around for decades. Most methods of detecting planar regions are based on one of three main categories [15, 1]:

- Hough Transform (HT)
- RANSAC (RC)
- Region Growing (RG)

### Hough Transform

The original motivation behind the Hough transform was detecting lines in images [10]. All points are sequentially processed via a voting procedure to detect the best fitting line over a set of 2d points. Multiple lines with different orientations are fit through each given point  $p$ . Because a line in slope-intercept form parallel to the y-axis would lead to an infinite slope, the Hesse normal form is chosen as the primary line representation[7].

In Hesse normal form, an individual line can be parameterized with a pair  $(r, \theta)$ , with  $r$  being the orthogonal distance origin to the plane and  $\theta$  being the angle between the x-axis and the line that connects the origin to the closest point on the line. This pair is also called a *Hough Space* in this context. Votes are cast on the corresponding value of  $\theta$ , depending on the number of inliers within a specific *Hough Space*  $(r_i, \theta_i)$ . The map that connects the votes to each  $\theta$  is called an *accumulator*. Finally, the best fitting line is determined by the number of votes it received.

In the context of plane detection in 3D point clouds, a plane would be uniquely identified by the triple  $(\rho, \theta, \phi)$ , with  $\rho$  being the orthogonal distance from the origin to the plane,  $\theta$  being the azimuthal angle, and  $\phi$  being the inclination. Since more parameters are needed to describe a plane in 3D, the accumulator must be adapted. Therefore, a three-dimensional accumulator is used, whereas the specific shape has been discussed [4].

## RANSAC

RANSAC (RAndom SAmple Consensus) has been researched for decades. While many use cases revolve around image processing, it is also heavily employed in many plane detection algorithms[27, 31, 3]. RANSAC is an iterative process. Each iteration randomly samples a certain amount of data points and fits a mathematical model through them. The level of outliers determines the quality of the obtained model and preserves the best overall model.

Within the context of plane detection in 3D point clouds, an approach could involve random sampling of 3 points, fitting a plane through them, and counting the number of points within a certain range of the plane[31]. The model, in that case, could be a cartesian plane equation.

## Region Growing

Region Growing methods are often used in the field of image or point cloud segmentation [21, 29]. RG-based segmentation methods aim to grow a set of disjoint regions from an initial selection of seed points. The regions increase in size by inserting neighboring values based on an inclusion criterion. The quality of the resulting regions depends on the choice of seed points, e.g., a very noisy seed point could decrease overall quality [18]. In the context of this work, a criterion for region growth could be the distance or curvature between a region and its adjacent data points.

## 2.4 Plane Detection Algorithms

This section describes four algorithms that are used for the evaluation during this work.

### 2.4.1 Robust Statistics approach for Plane Detection

Robust Statistics approach for Plane Detection (RSPD) [1] is based on region growing. After taking an unorganized point cloud as input, the procedure is divided into three phases; *Split, Grow and Merge*.

**Split** The authors propose to use an octree to recursively subdivide the point cloud. The subdivision is repeated until every leaf node contains less than 0.1% of the total amount of points. This is followed by a planarity test, during which the octree is traversed bottom-up. If all eight children of a node  $n$  are leaf nodes and fail the planarity test,  $n$  replaces its children and becomes a leaf node of its own. This procedure is repeated until the root of the octree is reached.

**Grow** In preparation for the growth phase, a neighborhood graph (NG) over the entire point cloud is created. Every node of NG represents one point and an edge between two nodes exists only if a k-nearest-neighbor search detects both points being in the same neighborhood.

The graph construction is subsequently followed by a breadth-first-search, during which a point  $x$  is inserted into a planar patch  $p$  if it satisfies the following conditions:

- $x$  is not included in any patch *and*
- $x$  satisfies the inlier conditions for  $p$ :
  - The distance  $d$  of  $x$  to  $p$  is smaller than a threshold  $\theta_d$  (see Eq. 2.1) *and*
  - The angle  $\phi$  between the normals vectors of  $x$  and  $p$  is less than a threshold  $\theta_a$  (see Eq. 2.2).

$$d = |(x - p.\text{center}) \cdot p.\text{normal}| < \theta_d \quad (2.1)$$

$$\phi = \text{acos}(|x.\text{normal}, p.\text{normal}|) < \theta_a \quad (2.2)$$

**Merge** In the last phase, the previously grown patches are merged. Two planar patches  $P_1$  and  $P_2$  can be merged, if the following conditions are met:

- The octree nodes of  $P_1$  and  $P_2$  are adjacent,
- $P_1.n$  and  $P_2.n$  have a divergence within a tolerance range *and*
- at least one inlier of  $P_1$  satisfies the inlier conditions(see Eq. 2.1+2.2) from  $P_2$  and vice versa.

This phase returns all maximally merged planar patches, i.e. the final planes.

### 2.4.2 Oriented Point Sampling

Oriented Point Sampling (OPS) [27] accepts an unorganized point cloud as input. First, a sample of points is uniformly selected. The normal vectors of these points are estimated using SVD and the  $k$  nearest neighbors, which had been obtained using a k-d tree. An inverse distance weight function is employed to prioritize neighboring points that are closer to the sample of which the normal vector is currently being estimated.

After normal estimation, one-point-RANSAC is used to find the largest plane. Usual RANSAC implementations sample three points to fit a plane. However, OPS fits a plane with only one sample point and its normal vector. Once a plane with the most inliers is obtained, its normal vector is re-estimated using SVD on all inliers, and all inliers are removed from the point cloud. This process is repeated until the number of remaining points falls below a predefined threshold  $\theta_N$ .

### 2.4.3 3D-KHT

Limberger and Oliveira[15] propose a hough transform-based plane detection method, which accepts unorganized point clouds as input [15]. The point cloud is spatially subdivided. The authors propose the usage of octrees over k-d trees because the k-d tree lacks efficiency in creation and manipulation. Furthermore, the octree succeeds in capturing the shapes inside the point cloud, while the k-d tree does not.

Each leaf inside the octree continues subdividing until the points inside a leaf node are considered approximately coplanar, or the number of points is less than a predefined threshold. The authors recommend this threshold to value 30 for large point clouds. After the approximately coplanar nodes are refined by removing outliers, a plane is fit through the remaining points.

This plane  $\pi$  can, in polar coordinates, be uniquely described by a triple  $(\rho, \theta, \phi)$ . Inspired by Borrmann et al.[4], an accumulator ball (Fig. 2.2b) is used for the voting procedure because the cells in polar regions are smaller (and therefore contain fewer normal vectors) in three-dimensional accumulator arrays, as portrayed in Figure 2.2a.

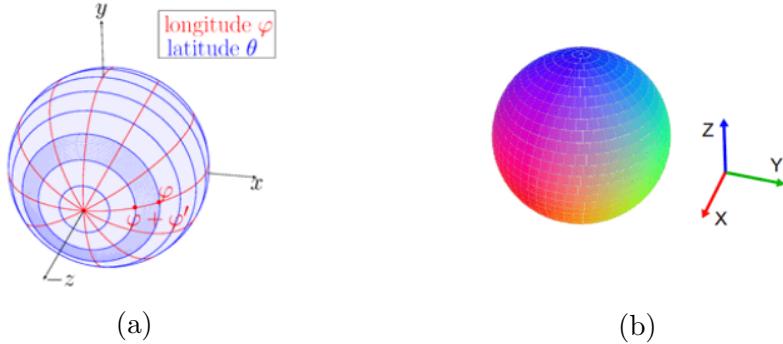


Figure 2.2: Accumulator array (a), taken from [4, Figure 3]. Accumulator ball(b) used in 3D-KHT, taken from [15, Figure 5].

During the voting procedure, votes are not cast for each data point but rather on previously calculated approximately coplanar clusters. When casting a vote on a given cluster  $c_i$  with its plane (represented by  $(\rho, \theta, \phi)$ ), the corresponding entry in the accumulator ball is updated. With this update, its neighboring clusters also receive a vote determined by the uncertainty value of  $c_i$ . Due to the non-discrete values of uncertainty, the votes are floating-point values as well.

All Peaks within the accumulator ball are detected in the last step. Because the votes tend to be sparsely distributed [15, Section 3.4], an auxiliary array  $A$  is used to memorize the entries inside the accumulator that are set. When an accumulator index is assigned a value for the first time, it is also added to  $A$ . Therefore, it is only necessary to iterate the auxiliary array to find peaks inside the accumulator. Furthermore, an intermediary smoothing step is performed by merging adjacent peaks inside the accumulator and storing them in  $A$ . Then,  $A$  is sorted in descending order. If a cell  $c$  in the accumulator has not yet been visited during iteration,  $c$  is considered a peak. In addition,  $c$  and its 26 neighboring cells are tagged as *visited*. That way, the most dominant plane, i.e., the one with the most votes, is detected first. Finally, the detected planes are sorted by the number of different clusters that voted for them.

#### 2.4.4 OBRG

OBRG (Octree-Based Region Growing [29]) is also a method that employs region growing.

First, an unorganized point cloud is recursively subdivided using an octree. An octree node  $n$  repeatedly subdivides itself into eight children until the level of  $n$  supersedes a predefined maximum subdivision value or if the amount of contained points in  $n$  is less than a predefined minimum of included points. Saliency features are calculated for every

leaf node in preparation for the region growing step. A normal vector is obtained by performing a principle component analysis (PCA) on the points inside each leaf node. The best-fitting plane of each leaf is defined by the mean normal vector and its center point. A residual value is obtained by taking the RMS of the distance of all included points to the plane.

For the region growing phase, all leaf nodes are selected as individual seed points. Starting from the seed with the lowest residual value, which relates to a low amount of noise, a neighboring leaf node  $n$  is inserted into the region if  $n$  does not belong to any region and the angular divergence between both normal vectors is smaller than a predefined threshold.

Lastly, a refinement step is employed. Fast refinement (FR) is performed on regions that succeed in a planarity test, i.e., 70%-90% of included points fit the best plane. FR is leaf-based, and all previously unallocated neighboring nodes that satisfy an inlier criterion are added to the region. General refinement (GR) is performed on regions that are considered non-planar. In contrast to the fast refinement, GR is point based. Therefore, points from neighboring and previously unallocated leaf nodes are considered and inserted into the region if they, too, satisfy the inlier criterion. The refinement process returns a complete set of planar regions.

## **2.4.5 PEAC - Probabilistic Agglomerative Hierarchical Clustering**

## **2.4.6 CAPE - Fast Cylinder and Plane Extraction**

## **2.4.7 SCH-RG - Plane Extraction using Spherical Convex Hulls**

## **2.4.8 D-KHT - Hough Transform for Real-Time Plane Detection**

## **2.4.9 DDFF - Depth Dependent Flood Fill**

## **2.4.10 PlaneNet**

## **2.4.11 PlaneRecNet**

## **2.4.12 PlaneRCNN**

## **2.5 Data Formats**

### **2.5.1 Common Input Types**

The fundamental difference between UPC and OPC is their format. Each point cloud has a *width* and a *height* parameter. An unorganized point cloud  $c$  is generally equal to an unordered 1D array of 3D coordinates, i.e.,  $\text{width} = |c|$  and  $\text{height} = 1$ . In contrast, the memory layout of an organized point cloud is a 2D array, where the width and height depend on the resolution of the used sensor. Taking the maximum resolution of the T265(see Table 2.1) as an example, the OPC would have a *width* and *height* of 1280 and 720, respectively. Intuitively, the value at index (0,0) would be in the top-left corner, and the value at index (*width*,*height*) would be in the bottom-right corner. Depth images are inherently similar to organized point clouds, given their resolution and two-dimensional structure. The primary difference is that the values stored in the array are distances to the sensor instead of 3D coordinates.

Dataset	Input Format	Real	Indoor	GT
SegComp [12]	DI	N	/	planes
2D-3D-S [2]	UPC	Y	Y	objects
NYU V2 [24]	DI	Y	Y	classes
Kinect [20]	OPC	Y	Y	planes
ICL-NUIM [9]	DI	Y	Y	trajectory
SYNBEP [23]	OPC	N	/	planes
ARCO [11]	OPC	Y	Y	/
SUN [25]	DI	Y	Y	objects
Leica <sup>3</sup>	UPC	Y	N	planes
TUM [26]	DI	Y	Y	trajectory

Table 2.2: Popular Datasets. The *GT*(Ground Truth) column specifies what the ground truth of each dataset represents.

### 2.5.2 Common Plane Formats

## 2.6 Datasets

### 2.6.1 2D-3D-S

2D-3D-S was recorded in three different buildings and divided into six distinct areas, including 272 different scenes. A detailed statistic of the included scene types can be found in Table 4.1. An individual scene has a complete unstructured point cloud and a list of annotated files representing semantically different objects that can be found therein. The dataset includes a wide range of point cloud sizes, with a minimum of  $8 \cdot 10^4$  and a maximum of  $7 \cdot 10^6$

diese 2 **sätze sind dann jetzt wahrscheinlich misplaced** Furthermore, one could argue that an uneven distribution of scene types introduces a particular bias. While it is true that the distribution is quite uneven, the dataset nevertheless reflects a realistic distribution of scene types since it is not realistic if a building contains only lecture halls. Inversely, it is appropriate to assume that an office complex contains a substantial amount of hallways needed to connect all offices.

---

<sup>3</sup><https://shop.leica-geosystems.com/de/leica-blk/blk360/dataset-downloads>

## 2.7 Evaluation Metrics

Wenn man Sachen segmentiert oder Muster erkennen möchte usw. benutzt man oft zur Evaluierung Metriken, die die Qualität der benutzten Methode beschreiben. Usual metrics are *precision*, *recall* and the *f1-score*. In general, *precision* describes how many of the results are relevant, i.e., the percentage of correctly calculated values (see Eq. 2.3). *Recall* describes the ratio of relevant results to all relevant data, i.e. the likelihood of a result being relevant (see Eq. 2.4). Lastly, the *f1-score* is the harmonic mean of the former two metrics (see Eq. 2.5).

$$Precision = \frac{|\{\text{correct values}\} \cap \{\text{obtained values}\}|}{|\{\text{obtained values}\}|} \quad (2.3)$$

$$Recall = \frac{|\{\text{correct values}\} \cap \{\text{obtained values}\}|}{|\{\text{correct values}\}|} \quad (2.4)$$

$$f1\text{-score} = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \quad (2.5)$$

In the context of this work, we calculate *precision*, *recall* and the *f1-score* as follows. Required are the original point cloud  $PC$ , the corresponding list of ground truth planes  $GT$  and the planes obtained from a plane detection algorithm  $A$ . First, we regularize the  $PC$  to reduce complexity and to avoid proximity bias, because of the inverse relationship between distance to sensor and cloud density. This regularization is obtained through voxelization of the point cloud. With this voxel grid, we can now calculate corresponding sets of voxels for each list of points that represent a plane. In the next step, we compare our planes from  $GT$  with  $A$  to obtain a list of corresponding pairs of ground truth and found planes. A ground truth plane  $gt_i$  is marked as *detected* if any plane from the list of found planes achieves a minimum voxel overlap of 50%. With this list of correspondences, we calculate *precision*, *recall* and the *f1-score*.

For a given ground truth plane  $gt_j$  and a corresponding detected plane  $a_k$  we can sort a given voxel  $v_i$  into the categories *True Positive(TP)*, *False Positive(FP)* and *False Negative(FN)* as follows.

$$v_i \in gt_j \wedge v_i \in a_k \Rightarrow v_i \in TP$$

$$v_i \in gt_j \wedge v_i \notin a_k \Rightarrow v_i \in FN$$

$$v_i \notin gt_j \wedge v_i \in a_k \Rightarrow v_i \in FP$$

With those four rules, we can calculate the precision, recall and F1 score like this:

$$Precision = \frac{|TP|}{|TP| + |FP|}$$

$$Recall = \frac{|TP|}{|TP| + |FN|}$$

## 3 Concept

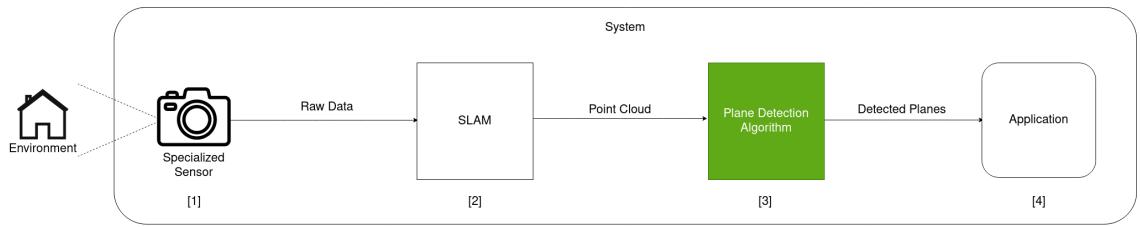


Figure 3.1: The procedure of the plane detection process. The specialized sensor records data ([1]), which is passed to a SLAM algorithm ([2]). After map assembly, a point cloud is handed to a plane detection algorithm ([3]). The detected planes are given to a use-case-specific application ([4]).

Many AR and VR Systems integrate plane detection into their software, some use it only to calculate the ground floor while others use plane detection to build a smaller model of the environment. Figure 3.1 shows a generic block diagram of such a VR/AR system including plane detection. In general, the environment is continuously recorded by a specialized sensor which is usually a camera([1]). A SLAM algorithm then integrates the new data into its already existing map([2]). The map, in form of a point cloud, is subsequently passed to a plane detection algorithm([3]). The algorithm performs the necessary steps to detect all planes inside the current map and passes the planes to the application([4]). The application would then further process those planes, e.g., by creating a live visualization of them or by assisting the movement of visually impaired people [5].

To remove any noticeable lag in the application, the plane detection step has to run under a temporal restriction, henceforth referred to as *real-time*. We state a more precise definition in Section 3.3.

When creating such a AR/VR system, the choice of plane detection algorithm is naturally of great importance. The problem is that most published algorithms are not inherently comparable. Often different datasets or metrics are used, which precludes comparison by quantification. Alternatively, algorithms are not comparable by internal functionality because many methods require different inputs, and the format of the

planes differs accordingly. All in all, selecting a single 'best' algorithm is impossible solely based on the metrics presented in their respective work.

To answer the question of which algorithm is best and whether it is real-time capable, we make a unified comparison of plane detection algorithms. To perform this evaluation, we need the following:

1. Appropriate plane detection algorithms,
2. a useful dataset *and*
3. a definition of *real-time*.

The following sections are dedicated to them.

## 3.1 Selection of Plane Detection Algorithms

Since most algorithms differ in certain aspects, it is not possible to compare them all uniformly. Furthermore, not all algorithms have the same motivation and therefore focus on different things. For example, evaluating an algorithm in a scenario it has not been designed for would not yield meaningful results. It is, therefore, necessary to first define objective criteria to superficially determine which algorithm seems to be relevant for the context of this work.

### 3.1.1 Criteria

In the following paragraphs, we define and outline appropriate criteria for the objective assessment of plane detection algorithms.

**Type of Input** The first criterion is the type of input expected by a plane detection algorithm. Allowing vastly different input types is likely to render the evaluation more complicated, if not impossible because an equivalent transformation between two input types is not always possible. Usually, the data representation of the recorded environment falls into one of three categories:

- *unorganized or unstructured point cloud* (UPC)
- *organized or structured point cloud* (OPC)
- *(depth-) image* (D-/I)

OPC and UPC both describe point clouds in the cartesian coordinate system. The primary difference is that the 3D coordinates inside an organized point cloud are saved in a 2D grid, while the unorganized cloud resembles an unsorted 1D array. Like OPC, depth images are a 2D grid of values. However, in contrast to the 3D coordinates of an OPC, the data points of depth images are the distances to the sensor.

**Detected Plane Format** Which specific representation the detected planes take the form of is also essential. If no uniform output type can be determined, consequently, no uniform metric for comparison can also be found.

Often the found planes are saved as a list of 3D points, henceforth referred to as inliers, which were assigned to a plane. Another often-plane output format is the cartesian equation of a plane described by a normal vector  $n$  and a vector  $d$ .

In methods that work on image data, found planes are often described by an image mask or pixels that belong together.

Finally, some methods use plane detection as a means to an end, e.g., for reconstructing a scene.

### 3.1.2 Plane Detection Algorithms

A list of state-of-the-art algorithms is compiled through comprehensive research of the current literature on plane detection (see Table 3.1). [explanation of table](#)

Im folgenden werden aus den zuvor aufgestellten kriterien die für diese arbeit sinnvollsten werte(?"ich nehme UPC aus UPC, OPC, DI... idk wie ich das nennen soll) ausgewählt und anhand dessen unpassende algorithmen von der evaluierung ausgeschlossen.

Plane Detection Algorithm	Input Data	Plane Format
RSPD [1]	UPC	inliers
OPS [27]	UPC	inliers
3DKHT [15]	UPC	inliers
OBRG [29]	UPC	inliers
PEAC [8]	OPC	inliers
CAPE [21]	OPC	n, d
SCH-RG [19]	OPC	inliers
D-KHT [28]	DI	inliers
DDFF [22]	DI	inliers
PlaneNet [16]	I	n, d
PlaneRecNet [30]	I	reconstructed scene
PlaneRCNN [17]	I	n, d

Table 3.1: Plane Detection Algorithms

Addressing the criterion of input type, we are only interested in performing plane detection in complete environments. Because unorganized point clouds are not limited in their dimension, they are more suitable for capturing entire environments. We hereby consider organized point clouds or images inappropriate because they do not offer a complete view on a scene. We, therefore, exclude *PEAC*, *CAPE*, *SCH-RG*, *D-KHT*, *DDFF*, *PlaneNet*, *PlaneRecNet* and *PlaneRCNN* from our evaluation.

Secondly, the detected planes need to be in the same format because, even for the same plane, different representations could very well lead to different results. Assume a plane in cartesian form and a plane represented by its inliers. The calculated metrics may differ significantly because the plane in cartesian form is infinitely dense. In contrast, the plane described by its inliers allows for holes and non-rectangular shapes, e.g., doorways or a round table. We thereby determine *inliers* as the preferred plane format and exclude all methods which do not comply, namely *CAPE*, *PlaneNet*, *PlaneRecNet*, and *PlaneRCNN*.

Finally, we end up with, and thus include, the following plane detection algorithms in our evaluation:

- RSPD
- OPS
- 3D-KHT
- OBRG

## 3.2 Datasets

As mentioned at the beginning of this chapter, we also need an appropriate dataset for the evaluation. Through extensive research of current literature, we compiled a list of popular datasets (see Table 2.2).

In Subsection 3.1.2, we determined unorganized point clouds as the type of input. Furthermore, we focus on plane detection in real environments in this work. Because most datasets do not conform to these two requirements, only *2D-3D-S* and *Leica* remain. Since we, additionally, want to detect planes in indoor environments, *Leica* also ceases to be an option. Thus, we choose 2D-3D-S as the dataset for the evaluation.

beispiel bilder von 2d3ds ?

Nonetheless, we cannot use the provided ground truth of 2D-3D-S because it represents the segmented scene on the level of objects, rather than planes in the scene. Consequently, we create a ground truth that aligns with the aim of this work, i.e., planes. We outline the details thereof in Section 4.3.

Lastly, 2D-3D-S does not inherit any temporal component, i.e., the unorganized point clouds do not grow incrementally over time. To the best of our knowledge, there exists no dataset that meets the above criteria and, additionally, provides a plane-focused ground truth. Thus, we record an incrementally growing dataset in the Faculty of Computer Science (FIN) at Otto-von-Guericke University Magdeburg.

To perform a thorough comparison between the FIN and 2D-3D-S, and, subsequently, between the static and the dynamic dataset, we record a scene for each of the following scene types:

- office
- conference room
- auditorium
- hallway

We focus on these four scene types because they are the most common in a real indoor environment. The recorded point clouds can be seen in Figure 3.2.

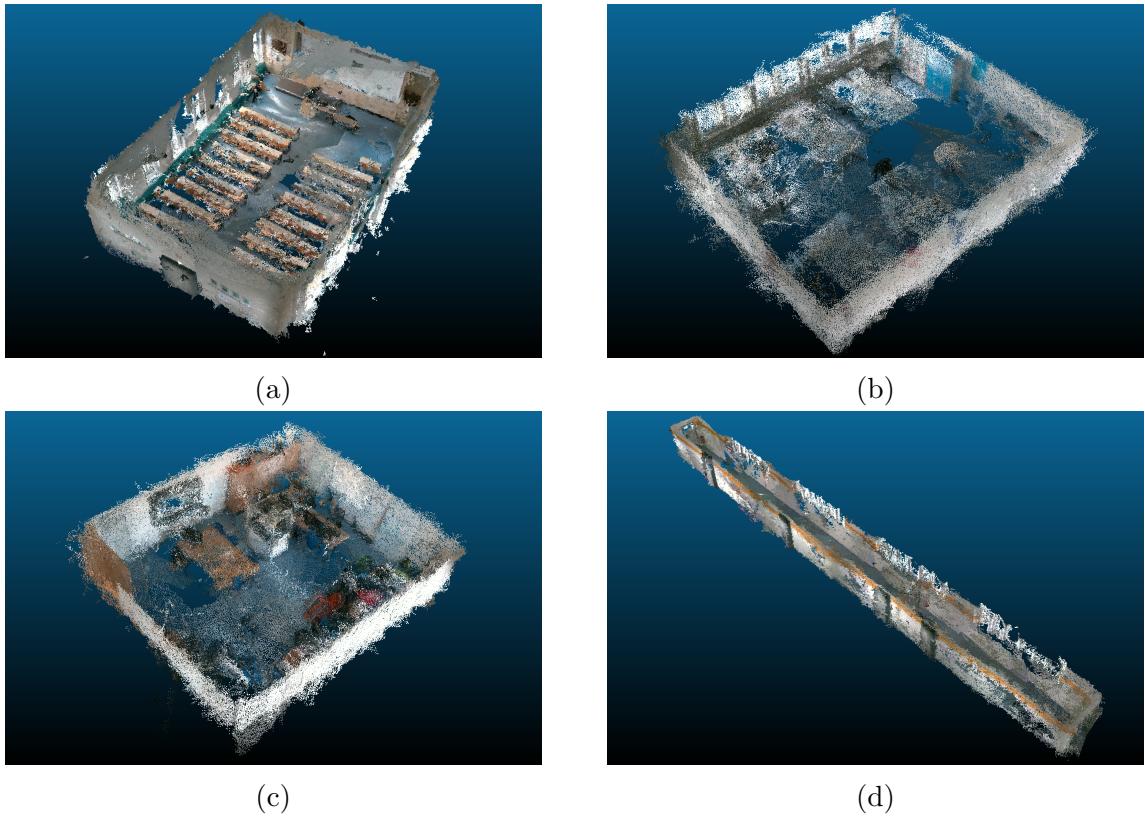


Figure 3.2: The recordings for each scene type: (a) auditorium, (b) conference room, (c) office and (d) hallway.

Lastly, since this is a novel dataset and thus has no ground truth, we create a ground truth. The details thereof are explained in Section 4.4.

### 3.3 Definition Real-Time

Finally, to determine whether or not an algorithm runs in real-time, we must first define the meaning of real-time.

Nicht alle Algorithmen funktionieren auf die gleiche Art und Weise. So kommt es, dass die Algorithmen verschiedene pre-und/oder post-processing Schritte implementiert haben. Da manche dieser pre-processing Schritte durch den SLAM algorithmus oder die Kamera Sensorik ersetzt werden können, werden wir daher im Folgenden zwei Definitionen von *real-time* präsentieren.

Generally, we have to consider possible hardware limitations, data flow, and how often it is needed to perform calculations, e.g., how quickly the SLAM algorithm updates the map (Figure 3.1, [2]) or how frequent new planes are needed (Figure 3.1, [4]).

The recorded raw data is not directly sent to the plane detection algorithm but instead given to RTAB-MAP, which then performs calculations to update and publish the map. Therefore, the upper limit is the frequency of how often RTAB-MAP publishes those updates, which by default is once per second.

**Total Real-Time** According to this upper limit of RTAB-MAP, we consider an algorithm *totally Real-Time* applicable, if it achieves an average frame rate of minimum 1, e.g., the total processing time of an algorithm lies under one second. In the rest of this work, we will refer to *total Real-Time* as  $RT_{tot}$ .

**Real-Time Plane Calculation** Being a subset of *totally Real-Time applicability*, *Real-Time Plane Calculation* determines the real-time applicability if the processing time of an algorithm *excluding* pre-processing lies under the aforementioned upper bound of 1s. We will henceforth refer to *Real-Time Plane Calculation* as  $RT_{calc}$ .

## 3.4 Summary

Many applications have constraints in the form of a temporal component. Augmented or Virtual Reality applications that include plane detection are no exception. In addition to time constraints, good quality is usually tightly coupled to expensive sensors. The selection of the best plane detection algorithm, however, is non-trivial. We selected possible algorithms and presented two definitions of *real-time* namely  $RT_{tot}$  and  $RT_{calc}$ . To evaluate to what extent it is possible to perform precise plane detection with a real-time constraint on off-the-shelf hardware, we compare the selected algorithms on the 2D-3D-S dataset and a self-recorded dataset.

# 4 Implementation

This chapter provides the implementation details of the outlined concept of the previous chapter.

## 4.1 System Setup

It is necessary to perform all experiments on the same machine to ensure a consistent comparison. We implement all algorithms and further architecture on a Lenovo IdeaPad 5 Pro, which runs Linux Ubuntu 20.04.5. The laptop has an AMD Ryzen 7 5800H CPU and 16 GB of RAM.

We install the most recent ROS distribution, *ROS Noetic Ninjemys*, as well as *realsense-ros* with all additional dependencies.

## 4.2 Plane Detection Algorithms

### 4.2.1 RSPD & OPS

We implement RSPD<sup>1</sup> and OPS<sup>2</sup> using their respective open source implementations on GitHub. Note that, while the implementation of RSPD is provided by the author, we could not determine whether the user who uploaded his implementation of OPS is affiliated with Sun and Mordohai[27]. Both methods are implemented in C++ and depend on the C++ linear algebra library *Eigen*<sup>3</sup> and the C++ API of the Point-Cloud Library<sup>4</sup>, *libpcl-dev*.

---

<sup>1</sup><https://github.com/abnerjo/PlaneDetection>

<sup>2</sup><https://github.com/victor-amblard/OrientedPointSampling>

<sup>3</sup><https://eigen.tuxfamily.org/index.php>

<sup>4</sup><https://pointclouds.org/>

## 4.2.2 3D-KHT

The authors of 3D-KHT, provide an implementation, in form of a Visual Studio project, on their website<sup>5</sup>. Since the laptop we use does not run Windows, we use *cmake-converter*<sup>6</sup> to convert the solution to a CMake project we can build using *make*.

## 4.2.3 OBRG

To our knowledge, no open-source implementation is available for the algorithm. We, therefore, use our own implementation.

We implement the algorithm using python. We choose to write our own octree implementation for spatial subdivision of our point cloud, since public libraries like *open3d* are limited in terms of leaf node functionality. The subdivision is followed by calculating the saliency features using *open3d*'s normal estimation function. We follow the pseudocode as stated in [29, Algorithm 1]. We modify the insertion into the set of regions by adding a containment check, to avoid redundancy of regions. By reducing the number of regions (incl. redundancies), we also reduce the total calculation time. Since the exact values of all thresholds have not been specified, we empirically select as follows:

- $\theta_r = 0.08$
- $\theta_{ang} = 0.3$
- $\theta_d = 0.08$
- $\theta_M = 5000$

To determine a region's planarity, we calculate the number of points that fit the best fitting plane within a predefined threshold. If that number supersedes the proposed 70%-90%, depending on the expected noise, the region is considered planar [29, Section 3.4].

It is worth noting that the choice of implementation using python is inferior considering calculation time when compared with an equivalent implementation in C++. Writing an optimized implementation in C++ would, therefore, go beyond the scope of this work, as the optimization of a single method is not our focus.

---

<sup>5</sup>[https://www.inf.ufrgs.br/~oliveira/pubs\\_files/HT3D/HT3D\\_page.html](https://www.inf.ufrgs.br/~oliveira/pubs_files/HT3D/HT3D_page.html)

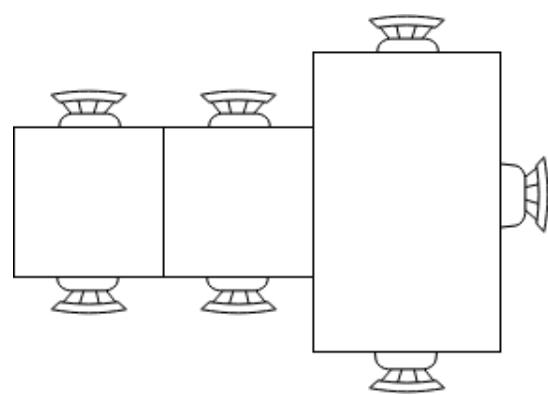
<sup>6</sup><https://cmakeconverter.readthedocs.io/>

## 4.3 2D-3D-S

The 2D-3D-S dataset provides a ground truth in form of annotated point clouds corresponding to 13 object classes. Since these annotated objects are not always planar, we cannot use them for the evaluation of plane detection algorithms. Thus, we create a ground truth that focuses on planar structures. We use the open-source 3D point cloud and mesh processing software *CloudCompare* to visualize a scene and manually segment included planes. Because we cannot assume all walls to be planar or that, e.g., the tops or three adjacent tables always form the same number of planes (see Figure 4.1b), we have to view each point cloud and segment the included planes manually. An exemplary before-and after-segmentation is shown below in Figure 4.1a.



(a) Ground Truth Segmentation of a hallway in CloudCompare. Shown is the input cloud on the left and segmented planes on the right. Both are cropped and without ceilings for visualization purposes.



(b) The provided ground truth considers these tables to be three separate objects. Within the context of plane detection, the three table tops would form exactly one plane.

The manual segmentation process is very time consuming, not only because of the large amount of data but also due to the following recurring questions.

1. "Which level of curvature separates planar from non-planar?",
2. "How sparse can a cloud be to still be considered a plane?" *and*
3. "When does a plane need splitting(or multiple planes merging)?"

All these questions slow down the segmentation process. On average, the segmentation of a scene took 8-10 minutes, which, for all 272 scenes, would equate to a total of 36-45 hours. To reduce the time spent in segmentation, we perform an initial analysis of the scenes in a given area and omit scenes that show no noticeable difference compared to others. This analysis reduces the number of segmented scenes to slightly more than half of the total, reducing the time to 18-23 hours.

The results of the manual segmentation process are documented in Table 4.1.

Scene Categories	Area_1	Area_2	Area_3	Area_4	Area_5	Area_6	TOTAL	Planes
auditorium	-	2/2	-	-	-	-	2/2	70
conference room	2/2	1/1	1 / 1	3/3	3/3	1/1	11/11	375
copy room	1/1	-	-	-	-	1/1	2/2	45
hallway	8/8	12/12	6/6	14/14	1/15	6/6	48/61	977
lobby	-	-	-	2 / 2	1/1	-	3/3	207
lounge	-	-	2/2	-	-	1/1	3/3	101
office	16/31	5/14	10/10	9/22	4/42	3/37	48/156	1116
open space	-	-	-	-	-	1 / 1	1/1	10
pantry	1/1	-	-	-	/1	1/1	3/3	73
storage	-	9/9	2 / 2	4/4	4/4	-	19/19	222
WC	1/1	2/2	2/2	4/4	4/2	-	11/11	214
							<b>139/272</b>	<b>3410</b>

Table 4.1: 2D-3D-S statistics. Shown are the number of scenes per category and for how many we created a ground truth ( $\#GT/\#Total$ ). Furthermore, the rightmost column reports the number of segmented planes per scene category.

## 4.4 FIN Dataset

Running *realsense-ros* and holding our cameras, we walk through the aforementioned parts of the building while scanning to the best of our ability. We save each incremental map update to a file for later usage.

Since no ground truth exists for a novel dataset like this, we create a set of ground truth planes  $gt_{end}$  for only the most recent update of each scene, e.g., for the entire recording. To prepare for the evaluation of a map  $m_t$  at a given time  $t$ , we crop all planes in  $gt_{end}$  by removing all points that are not present in  $m_t$ , as shown in Figure 4.2. We speed up this expensive process by employing a KD-Tree neighbor search with a small search radius since we only need to know whether a certain point is present or not.

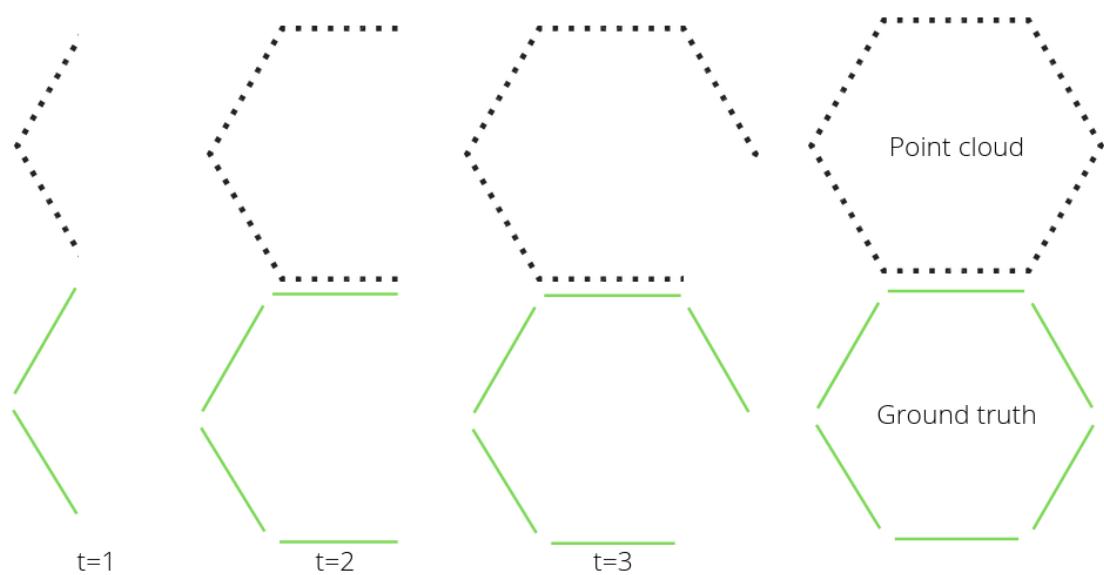


Figure 4.2: Dynamic ground truth generation. All planes that are included in *Ground Truth* are cropped depending on the available point cloud at each time  $t$

# 5 Evaluation

In diesem kapitel werden zuvor ausgewählte algorithmen einheitlich verglichen und die resultierenden ergebnisse ausgewertet.

## 5.1 Protocol

This work aims to determine which plane detection algorithm is the most suitable for an AR/VR system. For this decision, we uniformly compare the algorithms selected in Chapter 3. We split the comparison into two experiments conducted on different datasets: the 2D-3D-S and the self-created FIN dataset. Since both datasets are fundamentally different, we will perform the experiments and the analysis separately and then compare the results. First, we present the metrics used for comparison, followed by an outline of the used configurations of parameters for each experiment.

### 5.1.1 Metrics

**Accuracy** To quantify the accuracy of the plane detection algorithms, we use the detected planes and the created ground truth to calculate the three following metrics: Precision, Recall, and the F1-score. The procedure of calculation is taken from [1, Section 4] and detailed in Section 2.7.

hier noch mehr ins detail gehen? (antwort) \_\_\_\_\_

**Time Measurements** In addition to the accuracy, we are interested in the real-time applicability of an algorithm. We present two definitions of *real-time* in Section 3.3. Da wir zwei Definitionen von echtzeit haben, ist es sinnvoll, auch zwei Metriken der Berechnungszeiten einzuführen. Dazu, um präzise Aussagen über die Laufzeiten der algorithmen treffen zu können, teilen wir die Berechnung in pre-processing, plane calculation, und post-processing auf. Die jeweiligen Berechnungszeiten werden im Folgenden als  $t_{pre}$ ,  $t_{calc}$  und  $t_{post}$  bezeichnet. Neben präzisen Aussagen ermöglicht uns diese Aufteilung auch zu bestimmen, ob ein Algorithmus  $RT_{calc}$  ist. Um zu bestimmen, ob ein

Algorithmus  $RT_{tot}$  ist, werden wir dazu die totale Berechnungszeit  $t_{tot}$  angeben, welche sich aus der Summe der individuellen Zeiten ergibt:

$$t_{tot} = t_{pre} + t_{calc} + t_{post} \quad (5.1)$$

**das hier eher in den BG oder?** RSPD and OPS perform an initial estimation of normals, while 3D-KHT and OBRG construct an octree during their pre-processing phase. Note, that the octree construction of OBRG includes a local estimation of normals on the leaf level. OPS merges smaller planes if they pass a coplanarity test and then re-estimates the normals of the resulting plane. In the post-processing step, OBRG refines the borders of detected planes by inserting previously unallocated regions. The pre-and post-processing steps are summarized in Table 5.1.

**#detected planes als metrik einführen? ist imho wenig aussagekräftig da die GT erstellung so subjektiv ist**

	RSPD	OPS	3D-KHT	OBRG
Pre	Normal est.	Normal est.	Octree constr.	Octree constr.
Post	/	Merge	/	Refinement

Table 5.1: Pre-processing and post-processing steps of the plane detection algorithms. RSPD and 3D-KHT do not have any post-processing steps.

### 5.1.2 Parameterization of Algorithms

Because the datasets inherit different amounts of noise, it is necessary to modify the algorithms accordingly. We thereby modify the algorithms' parameterization to achieve more noise robustness. In the following, the parameterizations of the algorithms with respect to the two experiments are outlined. Therein, we refer to the parameterization of the 2D-3D-S experiment as the default configuration. All deviations from this default configuration necessary for the FIN experiment are determined empirically.

#### 5.1.2.1 RSPD

**die abkürzungen werden sicherlich im BG erklärt.**

Experiment	$l_O$	$\varepsilon$	MOR	$k$	MND	MDP
2D-3D-S	10	30	25%	30	60°	0.258
FIN	10	30	25%	30	60°	0.258

Table 5.2: Parameter configuration of RSPD used for the experiments.

**2D-3D-S** For the 2D-3D-S experiment, we use the parameters of the provided implementation. These parameters include the maximum octree level  $l_O$ , the minimum number of samples per leaf node  $\varepsilon$ , the maximum percentage of outliers per plane  $\theta_{outlier}$ , and the size of the nearest neighborhood  $k$ . Note that while  $k = 50$  is used in the respective paper [1, Section 3.3], we use  $k = 30$  because, in our experience, it produces sufficient results while reducing the pre-processing time.

**FIN** Da bei der erstellung von rspd besonders auf noise resistenz geachtet wurde, passen wir keinen parameter für das FIN experiment an. Es wurden diverse anpassungen getestet, keine davon haben jedoch die ergebnisse verbessert.

### 5.1.2.2 OPS

ja, die parameter werden im background *sicherlich* erklärt

Experiment	$\alpha_s$	$KNN$	$\theta_h$	$\theta_N$	$p$
2D-3D-S	3%	30	0.05	100	0.99
FIN	3%	<b>90</b>	<b>0.35</b>	100	0.99

Table 5.3: Parameter configuration of OPS used for the experiments.

**2D-3D-S** The parameter configuration used for the 2D-3D-S experiment is shown in the first row of Table 5.3. We use a sampling rate  $\alpha_s$  of 3% and a neighborhood  $KNN$  of 30 for the estimation of normal vectors. Additionally, we use a distance threshold  $\theta_h$  of 0.05(m). Furthermore, we set the inlier threshold  $\theta_N$  to 100 and the probability for adaptively determining RANSAC iterations  $p$  to 0.99, as proposed in [27, Section 4A].

**FIN** For the FIN experiment, we increase  $KNN$  to 90, as larger neighborhood sizes increase the accuracy of normal estimation and, consequently, the overall accuracy of a method. Furthermore, we increase the tolerated plane thickness  $\theta_h$  because an increase in sensor noise ultimately thickens the recorded planes. Both modifications are highlighted in bold in the second row of Table 5.3.

### 5.1.2.3 3D-KHT

Experiment	$\phi_{num}$	$\rho_{num}$	$s_{level}$	$s_{ps}$	$d_{max}$	$s_\alpha$	$s_\beta$
2D-3D-S	30	200	2	0.002	0.08	18	6
FIN	30	<b>100</b>	2	0.002	<b>0.1</b>	<b>8</b>	6

Table 5.4: Parameter configuration of 3D-KHT used for the experiments.

**2D-3D-S** The parameter configuration is shown in Table 5.4. We use an accumulator discretization of 30 and 200 for  $\phi$  and  $\rho$ , respectively. Starting to check for planarity at an octree level  $s_{level}$  of 2 seems to yield the best results. Limberger and Oliveira [15] propose a minimum of 30 samples per cluster, however, we use 0.2% of the total point cloud due to the wide ranges of point cloud sizes in the dataset (see Subsection 2.6.1). Lastly, we set  $s_\beta$  to 6, as proposed in [15, Section 3.1]. In contrast, using a  $s_\alpha$  value of 18 seemed to yield better results than the proposed 25.

**FIN** For the FIN experiment, we modify the values of  $\rho_{num}$ ,  $d_{max}$  and  $s_\alpha$  to accommodate for the higher levels of noise. Reducing  $\rho_{num}$  should decrease the accuracy, however, it seems to yield better results in a high-noise environment like the FIN dataset. We increase  $d_{max}$  and decrease  $s_\alpha$  to allow for slightly thicker, e.g. noisier, planes to be detected. The modification of parameters is highlighted in bold in Table 5.4.

### 5.1.2.4 OBRG

Experiment	$l_{max}$	$\theta_{res}$	$\theta_d$	$\theta_{ang}$	$\theta_M$	$\theta_p$
2D-3D-S	5	0.08	0.08	0.18	5000	90%
FIN	5	<b>0.22</b>	<b>0.2</b>	<b>0.2</b>	5000	<b>70%</b>

Table 5.5: Parameter configuration of OBRG used for the experiments.

**2D-3D-S** The used configurations for the experiments are shown in Table 5.5. Due to the low level of noise, we assign a very small tolerance to  $\theta_{res}$  and  $\theta_d$ . Additionally, we assign a high planarity threshold value of  $\theta_p = 90\%$ .

**FIN** Due to higher levels of noise, and thus, thicker walls, we increase the residual threshold  $\theta_{res}$ , the distance threshold  $\theta_d$ , and the angular divergence threshold  $\theta_{ang}$ . According to [29, Section 3.4], the planarity threshold  $\theta_p$  should be chosen between 70% and 90% depending on the noise level. As the expected noise level of the FIN dataset is much higher than the noise of the 2D-3D-S dataset, we reduce this threshold to 70%. The used parameters for the FIN experiment are summarized in the second row of Table 5.5.

## 5.2 Results

This section deals with the results of the experiments. The individual results of both experiments are presented and analyzed. Lastly, the results are compared.

**this still seems poorly placed** In einer echten Umgebung wachsen die Punktwolken inkrementell über Zeit. Die Angabe von Durchschnittswerten der Berechnungszeiten ist daher nur bedingt aussagekräftig, da diese bei anfänglichen Zeitschritten kürzer sein werden, als am Ende der Aufnahme. Wir werden daher neben Durchschnittswerten der Berechnungszeiten auch den Zusammenhang zwischen der Größe der Punktwolke und der Dauer der Berechnung betrachten.

### 5.2.1 2D-3D-S

Wir haben RSPD, OPS, 3D-KHT und OBRG auf 139 Scenen des 2D-3D-S Datensatzes laufen lassen. Im Anschluss wurden zu jeder Scene die Präzision, der Recall und der F1-Score von jedem Algorithmus berechnet. Dazu wurden die Berechnungszeiten gemessen und in pre-processing, plane detection und post-processing aufgeteilt. Die rechteste Spalte gibt die totale Berechnungszeit  $t_{tot}$  an. Table 5.6 zeigt die average der berechneten Ergebnisse zu jedem Algorithmus. Durch dicke Schrift werden die besten durchschnittswerte hervor gehoben. Für precision, recall und f1 sind die größten Werte dick geschrieben. Für alle zeitlichen Ergebnisse ist der geringste Wert durch dicke Schrift hervor gehoben. Dazu sei anzumerken, dass kein niedrigster Wert von  $t_{post}$  angedeutet wird, da RSPD und 3D-KHT keine post-processing schritte haben und daher "per default" weniger zeit in diesem Schritt verbringen.

Algorithm	Precision	Recall	F1-Score	$t_{pre}$	$t_{calc}$	$t_{post}$	$t_{tot}$
RSPD	84.80%	<b>89.79%</b>	<b>86.84%</b>	62.65	1.04	/	63.69
OPS	<b>88.98%</b>	70.45%	77.68%	13.12	10.97	1.01	25.10
3DKHT	71.40%	78.32%	75.19%	<b>0.71</b>	<b>1.03</b>	/	<b>1.74</b>
OBRG	81.38%	66.77%	71.00%	28.07	34.29	2.61	62.97

Table 5.6: Average results of each algorithm over the 2D-3D-S dataset. The right half of the inner columns shows the average time spent in pre-processing ( $t_{pre}$ ), the average time spent in the plane detection ( $t_{calc}$ ), and the average time spent in post-processing steps ( $t_{post}$ ). Note, that the absence of post-processing steps is denoted as "/". All times are measured in seconds.

**Accuracy** Mit einer Präzision von ca. 85%, einem Recall von ca. 90% und einem F1-Score von ca. 87% ist RSPD der insgesamt präziseste Algorithmus. OPS übertrifft RSPD zwar in der Precision mit einem Wert von ca. 89%, erzielt dafür aber deutlich geringere Recall und F1-Score Ergebnisse. 3D-KHT erzielt Ergebnisse mit einer Precision, Recall und F1-Score im Bereich zwischen ca. 71% und ca. 79%. OBRG erreicht zwar eine hohe Präzision mit ca 81%, erzielt aber schlussendlich die niedrigsten Recall und F1-Score Werte unter den Algorithmen mit 67%, bzw. 71%.

**Average Time** 3D-KHT scores the lowest processing times. With an average of 0.71 seconds spent in pre-processing, and an average of 1.03 seconds spent in plane detection, 3D-KHT only needs an average total of 1.74 seconds to process an entire point cloud. RSPD is the only algorithm that scores similar  $t_{calc}$  values, with an average of 1.04 seconds. However, RSPD's time spent in pre-processing is the highest among the algorithms. With an average  $t_{tot}$  of ca. 25 seconds and ca. 63 seconds, OPS and OBRG, respectively, run dramatically slower than the other two algorithms.

**Relationship of Time and Size** As stated in Paragraph 5.1.1, it is useful to consider the relationship between the computation times and the amount of processed data. For each scene of the 2D-3D-S dataset, the pairs of processing times and point cloud file sizes are presented in Figure 5.1. Note, that the most scenes have a file size of under 50mb.

Die pre-processing Zeiten von RSPD, OPS und OBRG scheinen ein lineares Verhältnis zu der File size zu haben. 3D-KHT hat als einziger Algorithmus ein logarithmisches Wachstum bei den pre-processing zeiten.

Der Verlauf von  $t_{calc}$  bei 3D-KHT und RSPD ist sehr ähnlich, was die durchschnittswerte aus Tabelle 5.6 bestätigt. Beide Laufzeiten scheinen ein sehr geringes, lineares Wachstum in Abhängigkeit zur Dateigröße zu haben.

The plane detection step of OPS seems to be somewhat dependent on the file size. However, the distinct processing times seem to fluctuate more in comparison to 3D-KHT and RPSD, which can be seen by the vertical broadness of the yellow graph. The  $t_{calc}$  values of OBRG also show fluctuations, however, the range is larger compared to OPS. Overall, the time OBRG spent in plane detection does not seem to depend directly on the size of the processed point cloud.

3D-KHT and RSPD do not have post-processing steps, hence the absence in their respective subfigures. The post-processing times of OPS seem to be primarily negligible, given the narrow spikes. On the other hand, the post-processing times of OBRG seems to be somewhat stable around  $10^0$ s, with a few exceptions, e.g.  $t_{post} = 10^{-2}$ s at a file size of ca. 130. **wenn log scale und graph grade, sag ich dann dass das wachstum logarithmisch oder linear ist?**

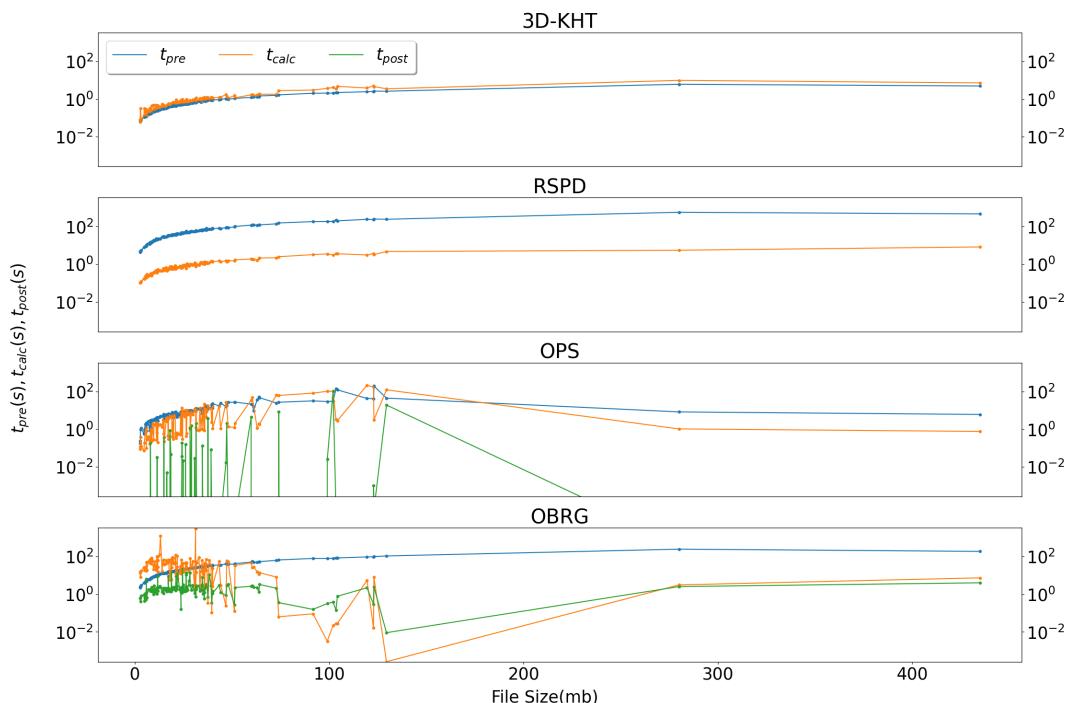


Figure 5.1: Time spent in pre-processing ( $t_{pre}$ ), plane detection ( $t_{calc}$ ), and post-processing ( $t_{post}$ ) per point cloud file size of the 2D-3D-S dataset. Note, that the y-axis is scaled logarithmically to the base of ten.

**Summary 2D-3D-S Experiment** OPS hat den größten Wert in Precision, während RSPD die höchste Accuracy und den größten F1-Score erzielt. 3D-KHT hat mit durchschnittlich  $1.74s$  die geringste total Berechnungszeit  $t_{tot}$ . RSPD und OBRG haben mit mehr als 60 sekunden die größten  $t_{tot}$  Werte unter den Algorithmen, wobei  $t_{pre}$  bei RSPD den Großteil ausmacht.

Figure 5.1 zeigt, dass die Laufzeiten von 3D-KHT am geringsten sind. RSPD hat ebenfalls geringe  $t_{calc}$  Werte, verbringt jedoch die durchweg längste Zeit im pre-processing. Ferner scheinen die pre-processing Zeiten aller algorithmen im allgemeinen von der Größe der Punktwolke abhängig zu sein. Die plane detection Laufzeiten von OPS und OBRG fluctuieren, OPS weniger als OBRG. Die post-processing Laufzeiten von OPS sind insgesamt zu vernachlässigen, bei OBRG, mit Ausnahme von Ausreisern bei mittleren Dateigrößen, stabil bei durchschnittlich  $2.61s$ , siehe Tabelle 5.6.

## 5.2.2 FIN

Jeder der insgesamt 732 Zeitschritte des FIN Datensatzes wurde von jedem algorithmus verarbeitet. Im Anschluss wird jeder Zeitschritt separat ausgewertet, also die Präzision, der Recall und der F1-Score berechnet. Dazu wurden, wie beim 2D-3D-S Experiment die Berechnungszeiten der Algorithmen für jeden Zeitschritt gemessen. Die durchschnittlichen Ergebnisse über alle Zeitschritte aller Scenen des FIN Experiments werden in Table 5.7 präsentiert. Für precision, recall und den f1-score sind die höchsten Werte dick geschrieben. Die niedrigsten Zeiten jedes Berechnungsschritts sind ebenfalls durch dicke Schrift gehighlighted.

Algorithm	Precision	Recall	F1-Score	$t_{pre}$	$t_{calc}$	$t_{post}$	$t_{tot}$
RSPD	57.30%	<b>60.75%</b>	<b>58.70%</b>	14.36	<b>0.19</b>	/	14.55
OPS	<b>69.38%</b>	29.23%	39.43%	4.61	0.89	0.13	5.63
3DKHT	49.76%	44.40%	46.48%	<b>0.14</b>	0.29	/	<b>0.43</b>
OBRG	49.23%	27.42%	33.94%	6.03	14.70	0.35	21.08

Table 5.7: Average Results of the FIN experiment. Shown are the average values of all scenes and time frames, sorted by algorithm. The right half of the columns shows the average time spent in pre-processing ( $t_{pre}$ ), the average time spent in the plane detection itself ( $t_{calc}$ ), and the average time spent in post-processing steps ( $t_{post}$ ). Note, that the absence of post-processing steps is denoted as "/".

**Accuracy** OPS hat mit nahezu 70% die höchste Durchschnittspräzision der Algorithmen. RSPD hat dafür mit ca. 60% die jeweils höchsten Werte für Recall und F1-Score erzielt. 3D-KHT und OBRG erzielen ähnliche Ergebnisse der precision, bei Re-

call und F1-Score hat 3D-KHT jedoch um ca. 13%, bzw. ca. 17% höhere Werte als OBRG.

**Average Time** RSPD braucht unter den algorithmen die meiste Zeit im pre-processing. Im kontrast dazu hat RSPD mit 0.19 sekunden die kürzeste Zeit, die in der plane detection verbracht wird. Insgesamt braucht 3D-KHT mit durchschnittlich 0.43 sekunden jedoch am kürzesten für die komplette Berechnung  $t_{tot}$  eines Zeitschritts. OPS erzielt mit ca. 6 sekunden vergleichsweise durchschnittliche Zeiten und OBRG braucht mit mehr als 20 sekunden insgesamt am längsten für die Berechnung eines Zeitschritts.

**Relationship of Time and Size** Wie auch in Paragraph 5.2.1 möchten wir den Zusammenhang zwischen der Menge an Daten und den Zeiten der Berechnung von ihnen betrachten. In Figure 5.2 werden die Dauer der Berechnung jedes Zeitschritts der Dateigröße von der *auditorium* Scene des FIN Datensatzes gegenüber gestellt. **Dazu sei gesagt, dass vergleichbare grafiken für die anderen Scenen des FIN Datensatzes erstellt wurden. Sie stellen argumentativ keine neuen informationen dar und werden vollständigkeitshalber und aus platzgründen im anhang \$REF präsentiert. Es wurde sich hier für die Auditorium Scene entschieden, da diese die längste Aufnahme darstellt, und damit die meisten Daten beinhaltet.**

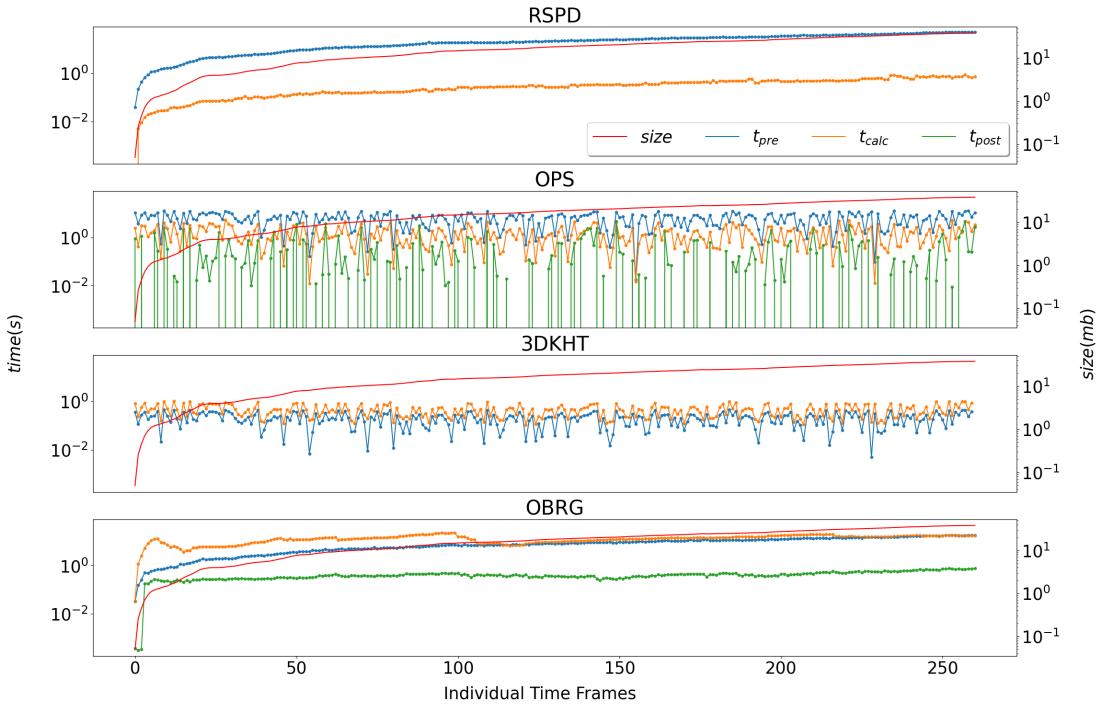


Figure 5.2: Time spent in pre-processing (blue), plane detection (yellow), and post-processing (green) of the hallway scene and cloud sizes (red) of each time step. Note, that both y-axes are scaled logarithmically to the base of ten

Bei RSPD und OBRG kann man direkt sehen, dass die pre-processing zeiten  $t_{pre}$  proportional zur größe der punktwolke sind. Im kontrast dazu scheinen bei OPS und 3D-KHT die pre-processing zeiten mehr mit den Zeiten der plane detection  $t_{calc}$  in Verbindung zu stehen. Bei RSPD, OPS und 3D-KHT scheint  $t_{calc}$  unabhängig von der Größe der Punktwolke zu sein und befindet sich stets unter einem gewissen Wert. Einzig auffallend ist das rapide Wachstum von OBRG im zeitraum vom Start bis ca. 25. Finally, the time spent in post-processing of OPS and OBRG is negligible as both times seem to stay at a constant value.

**Summary FIN Experiment** OPS hat die höchste Durchschnittspräzision, RSPD hat die größten Prozente in Recall und F1-Score. Auch in der plane detection hat RSPD mit durchschnittlichen  $0.19s$  den geringsten  $t_{calc}$  Wert unter den Algorithmen. Im kontrast dazu hat RSPD den längste Dauer des pre-processing mit  $14.36s$  im Durchschnitt. Der algoritmus mit der Kürzesten pre-processing, sowie totalen Zeit ist 3D-KHT mit  $t_{pre} = 0.14$ , bzw.  $t_{sum} = 0.43$ .

Die Berechnungszeiten von RSPD scheinen proportional zur Größe der Punktwolke, das gleiche gilt für OBRG. Dabei wächst bei RSPD  $t_{pre}$  deutlich schneller als  $t_{calc}$ . Im

kontrast dazu scheinen die pre-processing und plane detection Zeiten von OBRG am Ende der Aufnahme sich bei einen gemeinsamen Wert zu treffen. Die Berechnungszeiten von OPS scheinen unabhängig von der Größe der Punktwolke zu sein, dafür aber eher voneinander. Wie OPS scheint das Wachstum der Punktwolke keinen Einfluss auf die Berechnungszeiten von 3D-KHT zu haben, die totale Berechnungszeit einer Punktwolke ist konstant unter einer sekunde.

### 5.2.3 Comparison

When comparing Table 5.6 and Table 5.7, a pattern emerges: OPS has the highest precision value, RSPD yields the highest Recall and F1-Score, and 3D-KHT has the lowest average total processing time.

Wenn man sich Figure 5.2 and Figure 5.1 ansieht, fallen auch gemeinsamenkeiten auf: Der Kurvenverlauf von RSPD ist in beiden Experimenten praktisch identisch, ist proportional zur Größe der Punktwolke und die pre-processing Zeit stellt 99% der totalen Berechnungszeit dar. Eine andere gemeinsamkeit ist, dass die post-processing Zeiten von OPS sehr fluktuiert. Bei OBRG scheint die Dauer des post-processings bis zu einer Punktwolken Größe von ca. 80mb vergleichsweise stabil um einen geringen Wert zu sein.

Es gibt aber auch Unterschiede. Im Vergleich zum 2D-3D-S Experiment verhalten sich die Laufzeiten von 3D-KHT beim FIN experiment nicht proportional zur Größe der Punktwolke. Dazu sei gesagt,dass die maximale Größe im FIN experiment bei ca. 30mb liegt, was natürlich deutlich kleiner als das maximum des 2D-3D-S Datensatzes ( $> 400mb$ ) ist. Da in Figure 5.1 aber auch für kleinere Wolken eine proportionalität zu erkennen ist, wird der grund für die verschiedenen Verläufe wahrscheinlich die unterschiedliche Parametrisierung sein.

## 5.3 Summary

RSPD hat in beiden Experimenten insgesamt die höchsten Accuracy Werte. Dazu hat RSPD den größten  $t_{pre}$  Wert in beiden Experimenten und den kleinsten  $t_{calc}$  Durschnittswert im FIN Experiment. 3D-KHT hat in beiden Experiment die geringste totale Berechnungszeit und übertrifft im 2D-3D-S Experiment die plane detection Zeit von RSPD.

# **6 Conclusion and Future Work**

## **Summary**

**Use case, scenario, real-world application, current problem**

**Daher das thema dieser arbeit**

**algorithmen**

**Testscenario / datensatz**

**Experimentaufbau**

**Ergebnisse der experimente**

## **Fazit**

**Limitationen der ergebnisse**

**Algo x ist der beste, mögl verbesserungen**

# Bibliography

- [1] Abner M. C. Araújo and Manuel M. Oliveira. “A robust statistics approach for plane detection in unorganized point clouds”. en. In: *Pattern Recognition* 100 (Apr. 2020), p. 107115. ISSN: 00313203. DOI: 10.1016/j.patcog.2019.107115.
- [2] Iro Armeni et al. “3D Semantic Parsing of Large-Scale Indoor Spaces”. In: *Proceedings of the IEEE International Conference on Computer Vision and Pattern Recognition*. 2016.
- [3] Ramy Ashraf and Nawal Ahmed. “FRANSAC: Fast RANdom Sample Consensus for 3D Plane Segmentation”. en. In: *International Journal of Computer Applications* 167.13 (June 2017), pp. 30–36. ISSN: 09758887. DOI: 10.5120/ijca2017914558.
- [4] Dorit Borrmann et al. “The 3D Hough Transform for plane detection in point clouds: A review and a new accumulator design”. en. In: *3D Research* 2.2 (June 2011), p. 3. ISSN: 2092-6731. DOI: 10.1007/3DRes.02(2011)3.
- [5] Aparicio Carranza et al. “Plane Detection Based Object Recognition for Augmented Reality”. en. In: May 2021. DOI: 10.11159/cdsr21.305. URL: [https://avestia.com/CDSR2021\\_Proceedings/files/paper/CDSR\\_305.pdf](https://avestia.com/CDSR2021_Proceedings/files/paper/CDSR_305.pdf).
- [6] Davison. “Real-time simultaneous localisation and mapping with a single camera”. In: *Proceedings Ninth IEEE International Conference on Computer Vision*. 2003, 1403–1410 vol.2. DOI: 10.1109/ICCV.2003.1238654.
- [7] Richard O. Duda and Peter E. Hart. “Use of the Hough Transformation to Detect Lines and Curves in Pictures”. In: *Commun. ACM* 15.1 (Jan. 1972), pp. 11–15. ISSN: 0001-0782. DOI: 10.1145/361237.361242. URL: <https://doi.org/10.1145/361237.361242>.
- [8] Chen Feng, Yuichi Taguchi, and Vineet R. Kamat. “Fast plane extraction in organized point clouds using agglomerative hierarchical clustering”. en. In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. Hong Kong, China: IEEE, May 2014, pp. 6218–6225. ISBN: 978-1-4799-3685-4. DOI: 10.1109/ICRA.2014.6907776. URL: <http://ieeexplore.ieee.org/document/6907776/>.
- [9] A. Handa et al. “A Benchmark for RGB-D Visual Odometry, 3D Reconstruction and SLAM”. In: *IEEE Intl. Conf. on Robotics and Automation, ICRA*. Hong Kong, China, May 2014.

- [10] Peter E. Hart. “How the Hough transform was invented [DSP History]”. In: *IEEE Signal Processing Magazine* 26.6 (2009), pp. 18–22. DOI: 10.1109/MSP.2009.934181.
- [11] Alejandro Hidalgo-Paniagua et al. “A Comparative Study of Parallel RANSAC Implementations in 3D Space”. en. In: *International Journal of Parallel Programming* 43.5 (Oct. 2015), pp. 703–720. ISSN: 0885-7458, 1573-7640. DOI: 10.1007/s10766-014-0316-7.
- [12] Adam Hoover et al. “An Experimental Comparison of Range Image Segmentation Algorithms”. In: *IEEE Trans. Pattern Anal. Mach. Intell.* 18 (July 1996), pp. 673–689. DOI: 10.1109/34.506791.
- [13] Christian Kerl, Jurgen Sturm, and Daniel Cremers. “Dense visual SLAM for RGB-D cameras”. en. In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Tokyo: IEEE, Nov. 2013, pp. 2100–2106. ISBN: 978-1-4673-6358-7. DOI: 10.1109/IROS.2013.6696650. URL: <http://ieeexplore.ieee.org/document/6696650/>.
- [14] Mathieu Labb  and Fran ois Michaud. “RTAB-Map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation: LABB  and MICHAUD”. en. In: *Journal of Field Robotics* 36.2 (Mar. 2019), pp. 416–446. ISSN: 15564959. DOI: 10.1002/rob.21831.
- [15] Frederico A. Limberger and Manuel M. Oliveira. “Real-time detection of planar regions in unorganized point clouds”. en. In: *Pattern Recognition* 48.6 (June 2015), pp. 2043–2053. ISSN: 00313203. DOI: 10.1016/j.patcog.2014.12.020. URL: [https://www.inf.ufrgs.br/~oliveira/pubs\\_files/HT3D/HT3D\\_page.html](https://www.inf.ufrgs.br/~oliveira/pubs_files/HT3D/HT3D_page.html).
- [16] Chen Liu et al. “PlaneNet: Piece-Wise Planar Reconstruction from a Single RGB Image”. en. In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. Salt Lake City, UT: IEEE, June 2018, pp. 2579–2588. ISBN: 978-1-5386-6420-9. DOI: 10.1109/CVPR.2018.00273. URL: <https://ieeexplore.ieee.org/document/8578371/>.
- [17] Chen Liu et al. “PlaneRCNN: 3D Plane Detection and Reconstruction From a Single Image”. en. In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Long Beach, CA, USA: IEEE, June 2019, pp. 4445–4454. ISBN: 978-1-72813-293-8. DOI: 10.1109/CVPR.2019.00458. URL: <https://ieeexplore.ieee.org/document/8953257/>.
- [18] Aminah Abdul Malek et al. “Seed point selection for seed-based region growing in segmenting microcalcifications”. en. In: *2012 International Conference on Statistics in Science, Business and Engineering (ICSSBE)*. Langkawi, Kedah, Malaysia: IEEE, Sept. 2012, pp. 1–5. ISBN: 978-1-4673-1582-1. DOI: 10.1109/ICSSBE.2012.6396580. URL: <http://ieeexplore.ieee.org/document/6396580/>.

- [19] Hannes Mols, Kailai Li, and Uwe D. Hanebeck. “Highly Parallelizable Plane Extraction for Organized Point Clouds Using Spherical Convex Hulls”. en. In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. Paris, France: IEEE, May 2020, pp. 7920–7926. ISBN: 978-1-72817-395-5. DOI: 10.1109/ICRA40945.2020.9197139. URL: <https://ieeexplore.ieee.org/document/9197139/>.
- [20] Bastian Oehler et al. “Efficient Multi-resolution Plane Segmentation of 3D Point Clouds”. en. In: *Intelligent Robotics and Applications*. Ed. by Sabina Jeschke, Honghai Liu, and Daniel Schilberg. Vol. 7102. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 145–156. ISBN: 978-3-642-25488-8. DOI: 10.1007/978-3-642-25489-5\_15. URL: [http://link.springer.com/10.1007/978-3-642-25489-5\\_15](http://link.springer.com/10.1007/978-3-642-25489-5_15).
- [21] Pedro F. Proen  a and Yang Gao. “Fast Cylinder and Plane Extraction from Depth Cameras for Visual Odometry”. en. In: arXiv:1803.02380 (July 2018). number: arXiv:1803.02380 arXiv:1803.02380 [cs]. URL: <http://arxiv.org/abs/1803.02380>.
- [22] Arindam Roychoudhury, Marcelli Missura, and Maren Bennewitz. “Plane Segmentation Using Depth-Dependent Flood Fill”. en. In: *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Prague, Czech Republic: IEEE, Sept. 2021, pp. 2210–2216. ISBN: 978-1-66541-714-3. DOI: 10.1109/IROS51168.2021.9635930. URL: <https://ieeexplore.ieee.org/document/9635930/>.
- [23] Alexander Schaefer et al. “A Maximum Likelihood Approach to Extract Finite Planes from 3-D Laser Scans”. In: *2019 IEEE/RSJ International Conference on Robotics and Automation (ICRA)*. IEEE. May 2019. URL: <http://ais.informatik.uni-freiburg.de/publications/papers/schaefer19icra.pdf>.
- [24] Nathan Silberman et al. “Indoor Segmentation and Support Inference from RGBD Images”. In: *Computer Vision – ECCV 2012*. Ed. by Andrew Fitzgibbon et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 746–760. ISBN: 978-3-642-33715-4.
- [25] Shuran Song, Samuel P. Lichtenberg, and Jianxiong Xiao. “SUN RGB-D: A RGB-D scene understanding benchmark suite”. In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 567–576. DOI: 10.1109/CVPR.2015.7298655.
- [26] J. Sturm et al. “A Benchmark for the Evaluation of RGB-D SLAM Systems”. In: *Proc. of the International Conference on Intelligent Robot Systems (IROS)*. Oct. 2012.
- [27] Bo Sun and Philippos Mordohai. “Oriented Point Sampling for Plane Detection in Unorganized Point Clouds”. en. In: arXiv:1905.02553 (May 2019). arXiv:1905.02553 [cs]. URL: <https://github.com/victor-amblard/OrientedPointSampling>.

- [28] Eduardo Vera et al. “Hough Transform for real-time plane detection in depth images”. en. In: *Pattern Recognition Letters* 103 (Feb. 2018), pp. 8–15. ISSN: 01678655. DOI: 10.1016/j.patrec.2017.12.027.
- [29] Anh-Vu Vo et al. “Octree-based region growing for point cloud segmentation”. en. In: *ISPRS Journal of Photogrammetry and Remote Sensing* 104 (June 2015), pp. 88–100. ISSN: 09242716. DOI: 10.1016/j.isprsjprs.2015.01.011.
- [30] Yaxu Xie et al. “PlaneRecNet: Multi-Task Learning with Cross-Task Consistency for Piece-Wise Plane Detection and Reconstruction from a Single RGB Image”. en. In: arXiv:2110.11219 (Jan. 2022). number: arXiv:2110.11219 arXiv:2110.11219 [cs]. URL: <http://arxiv.org/abs/2110.11219>.
- [31] Michael Ying Yang and Wolfgang Forstner. “Plane Detection in Point Cloud Data”. en. In: *Proceedings of the 2nd int conf on machine control guidance* 1 (Feb. 2010), p. 16.

# **Declaration of Academic Integrity**

I hereby declare that I have written the present work myself and did not use any sources or tools other than the ones indicated.

Datum: .....  
(Signature)