Otto-von-Guericke-University Magdeburg

Faculty of Computer Science

Institute for Intelligent Cooperating Systems

Bachelor Thesis

Comparison of Real-Time Plane Detection Algorithms on Intel RealSense

Author:

Lukas Petermann

Examiner:
Prof. Frank Ortmeier

2nd Examiner:
M.Sc. Marco Filax

Supervisor:
M.Sc. Maximilian Klockmann

December 5, 2022

Petermann, Lukas
Comparison of Real-Time Plane Detection on Intel RealSense
Bachelor Thesis, Otto-von-Guericke-University Magdeburg, 2022.

Contents

List of Figures **5**

1 Introduction **6**
- 1.1 Real-Time Plane Detection . . . 6
- 1.2 Intel RealSense . . . 7

2 Background **8**
- 2.1 SLAM . . . 8
- 2.2 Intel Realsense . . . 9
- 2.3 Data Formats . . . 10
  - 2.3.1 Common Input Types . . . 10
  - 2.3.2 Common Plane Formats . . . 11
- 2.4 Plane Detection . . . 11
- 2.5 Plane Detection Algorithms . . . 12
  - 2.5.1 Robust Statistics approach for Plane Detection . . . 13
  - 2.5.2 Oriented Point Sampling . . . 14
  - 2.5.3 3D-KHT . . . 15
  - 2.5.4 Octree-based Region Growing . . . 16
  - 2.5.5 Probabilistic Agglomerative Hierarchical Clustering . . . 17
  - 2.5.6 Fast Cylinder and Plane Extraction . . . 18
  - 2.5.7 Plane Extraction using Spherical Convex Hulls . . . 20
  - 2.5.8 Depth Kernel-based Hough Transform . . . 21
  - 2.5.9 Depth Dependent Flood Fill . . . 22
  - 2.5.10 PlaneNet . . . 23
  - 2.5.11 PlaneRecNet . . . 24
  - 2.5.12 PlaneRCNN . . . 26
- 2.6 Datasets . . . 27
  - 2.6.1 2D-3D-S . . . 27
- 2.7 Evaluation Metrics . . . 28

3 Concept **30**
- 3.1 Selection of Plane Detection Algorithms . . . 31
  - 3.1.1 Criteria . . . 31
  - 3.1.2 Plane Detection Algorithms . . . 32

3.2 Datasets . . . 34
3.3 Definition Real-Time . . . 36
3.4 Summary . . . 36

4 Implementation **38**
4.1 System Setup . . . 38
4.2 Plane Detection Algorithms . . . 38
4.2.1 RSPD & OPS . . . 38
4.2.2 3D-KHT . . . 39
4.2.3 OBRG . . . 39
4.3 2D-3D-S . . . 39
4.4 FIN Dataset . . . 41

5 Evaluation **43**
5.1 Protocol . . . 43
5.1.1 Metrics . . . 43
5.1.2 Parameterization of Algorithms . . . 44
5.2 Results . . . 47
5.2.1 2D-3D-S . . . 47
5.2.2 FIN . . . 50
5.2.3 Comparison . . . 53
5.3 Summary . . . 53

6 Conclusion and Future Work **55**
6.0.1 Summary . . . 55
6.0.2 Fazit . . . 56

Bibliography **58**

List of Figures

2.1 RTAB-MAP Block Diagram 9
2.2 Hough Transform Accumulators 15
2.3 PlaneRecNet Architecture 24
2.4 PlaneRCNN Architecture 26

3.1 AR/VR System Overview 30
3.2 Dynamic Datasets 35

4.2 Dynamic Ground Truth Generation 42

5.1 Time per Cloud size 2D-3D-S 49
5.2 Time Results Auditorium 52

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:

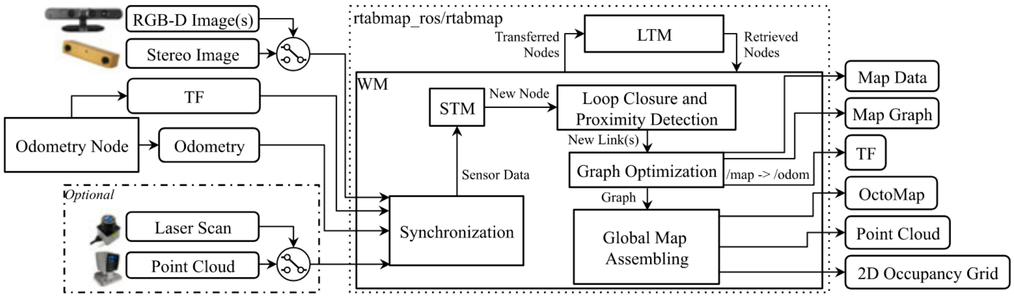


Figure 2.1: Block diagram of RTAB-MAP's main node. Taken from [14, Figure 1]

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 [1]) adaptation is also provided in Intel's Github repository [2].

[1] https://www.ros.org/
[2] 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 Data Formats

2.3.1 Common Input Types

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)
- image:
  - Depth (DI)
  - RGB (RGBI)

The differences between these input types are summarized in Table 2.2. 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., $width = |c|$ and $height = 1$. In contrast, the memory layout of an organized point cloud is a 2D matrix, where the width and height depend on the resolution of the used sensor. 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. When considering images, a differentiation between RGB images and depth images has to be made. Depth images are inherently similar to organized point clouds, given their resolution and two-dimensional structure. The primary difference is that the matrix stores distances to the sensor instead of 3D coordinates. RGB-images are, like Depth images, stored as a 2D Matrix but instead of distances or coordinates, the matrix stores values that describe the color of a pixel. Usually, this means that each pixel stores a triple that describes a specific color by the amount of included red, green, and blue.

Input Type	Value Types	Memory Layout
OPC	3D Coordinates	2D Matrix
UPC	3D Coordinates	1D Array
DI	Distances to Sensor	2D Matrix
RGBI	RGB	2D Matrix

Table 2.2: Possible inputs for plane detection algorithms.

2.3.2 Common Plane Formats

TODO

2.4 Plane Detection

introduction The field of plane detection has been around for decades. Most methods of detecting planar regions in unorganized point clouds 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, θ), with r being the orthogonal distance origin to the plane and θ 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 θ, depending on the number of inliers within a specific *Hough Space* (r_i, θ_i). The map that connects the votes to each θ 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 (ρ, θ, ϕ), with ρ being the orthogonal distance from the origin to the plane, θ being the azimuthal angle, and ϕ 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, 32, 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[32]. 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.

2.5 Plane Detection Algorithms

The following subsections detail the necessary background knowledge of all the plane detection algorithms mentioned in this paper. Aside from the functionality of a method, we give relevant higher-level information. This includes the separation of an algorithms process in pre-processing, plane detection, and post-processing, as well as the expected input format and the proposed output format.

2.5.1 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.*

pre-processing build octree and calculate normals for all points

plane detection

- build octree
- estimate normals

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 θ_d (see Eq. 2.1) *and*
  - The angle ϕ between the normals vectors of x and p is less than a threshold θ_a (see Eq. 2.2).

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

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

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.

2.5.2 Oriented Point Sampling

Oriented Point Sampling (OPS) [27] accepts an unorganized point cloud as input.

pre-processing

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.

plane detection

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 θ_N.

post-processing

- merge smaller planes if they pass a coplanarity test
- re-estimate normals after merge

2.5.3 3D-KHT

Limberger and Oliveira[15] propose a hough transform-based plane detection method, which accepts unorganized point clouds as input [15].

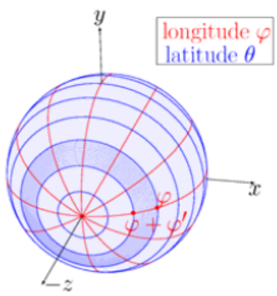
pre-processing

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.

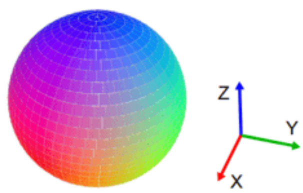
Each leaf inside the octree continues subdividing until the points inside a leaf node are considered approximately coplanar, or the number of points is less than a predefined threshold. The authors recommend this threshold to value 30 for large point clouds. After the approximately coplanar nodes are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

plane detection

This plane π can, in polar coordinates, be uniquely described by a triple (ρ, θ, ϕ). 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.



(a)



(b)

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 (ρ, θ, ϕ)), 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.

2.5.4 Octree-based Region Growing

Octree-Based Region Growing (OBRG) [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.

2.5.5 Probabilistic Agglomerative Hierarchical Clustering

Probabilistic Agglomerative Hierarchical Clustering (PEAC) [8] is a plane detection algorithm that takes an organized point cloud as input.

preprocessing

1. init graph:
   a) init nodes:
      - downsample resolution
   b) init edges
2. clean up graph by removing certain types of nodes and corresponding edges
   - nodes with high MSE (non-planar nodes)
   - missing data (due to sensor, e.g. behind windows, too far away)
   - depth discontinuities (e.g. monitor in front of wall)
   - boundary nodes

plane detection AHC:

1. build min heap for finding node with minimum MSE
2. exhaustively:
   a) find the node with the lowest MSE v
   b) merge with the best-fitting neighbor: $v \vee u_{best} \rightarrow u_{merge}$
   c) if the MSE of u_{best} exceeds a threshold T_{MSE}:
      - extract v from graph
   d) else:
      - insert edge (v, u_{best})
      - insert the merged node back into the graph

post-processing Refinement:

1. focus on three types of artifacts:
   a) sawtooth
   b) unused points
   c) over segmentation

   d) all of them are at boundaries
2. erode border nodes
3. perform pixel wise region growing, dabei alle 4 nachbarn betrachten
4. re-apply AHC on newly formed nodes

2.5.6 Fast Cylinder and Plane Extraction

Fast Cylinder and Plane Extraction (CAPE) [21] implements an algorithm that detects planes and cylinders in organized point clouds to improve the performance of visual odometry.

pre-processing

1. planar cell fitting
   - also used in PEAC
   - non-planar:
     - NaN
     - depth discontinuities
   - then: plane fitting with PCA(-¿normal) and eigenvalue (-¿MSE)
   - only the first two moments of 3d points needed, könig-huygen formula can be used to calc the cov matrix
2. build histogram of normals
   - convert normals of cell to polar and azimuthal angles
   - use those values for assorting to bins and building a histogram

plane detection

1. cell region growing
   - given histogram and grid of planar cells, (copy of G:) list of rem. cells R, Set of segments S
   - exhaustively, until R is empty:
     a) C = get cells from the most frequent bin
     b) if the number of cells is smaller than a threshold k_1: break
     c) choose seed: cell with minimum MSE out of C

d) grow seed

e) remove resulting region from R and remove cells in R from histogram

f) if $|R| < k_2$ continue

g) add R to S

return S

2. plane (+cylinder) fitting (for all regions:)
   - fit plane to region [A]
   - if ratio of second largest eigenvalue to smallest large enough :save grid segmentation and plane params
   - else: check if surface is invariant in one direction (property of open cylinders)
     - PCA on [n1,...,nn] stacked matrix of [N,-N] (normal of the cell)
     - only continue if ratio of first eigenvalue to last eigenvalue is larger than a threshold
     - use center and normal to projekt cell to plane
     - use ransac to fit a cylinder (das geht weil cylinder ebenen sind mit inf radius)
     - cylinder fitting returns set of set of segments
     - foreach subsegment:
     - fit plane (siehe [A]), and depending on the MSE, add to either plane set or cylinder set
3. plane region merging

post-processing boundary refinement:

1. erode cell boundaries with 3x3 kernel, 4er nbhood (wie in PEAC)
2. cells that are valid for refinement are given by the difference between the eroded segment and the segment to expand
3. distance between segment plane model to each point within the valid cells is calculated
4. pixel within the cells are added to the segment if the MSE is less than the model MSE*konstant (im paper= 9)

2.5.7 Plane Extraction using Spherical Convex Hulls

As the name suggests, Plane Extraction using Spherical Convex Hulls (SCH-RG) employs spherical convex hulls to detect planes in organized point clouds.

pre-processing

- bilateral filter to denoise
- generate normal map
- divide image into 20x20 pathes
- calc MSE for all patches, discard those with higher than threshold
- sort patches according to patch plane roughness
- patch centers = seed points

plane detection

- exhaustively: perform SCH-RG on seed points
- if a seed is associated to an extracted plane, remove it from SPL
- SCH-RG:
  1. given: patch T and seed s, clustered point set P
  2. add seed to P
  3. start a SCH from seed
  4. create neighborhood queue for seed
  5. exhaustively: get point from Q
  6. discard if distance to origin larger than threshold
  7. if p inside of the SCH, add to P and add its neighbors to Q
  8. else:
     - if point not in the CPR: discard
     - else: expand SCH with point, add point to P and add its neighbors to Q
- returns clustered point set

post-processing

- dilation of planes to eliminate holes caused by noise (not further detailed)

2.5.8 Depth Kernel-based Hough Transform

Depth Kernel-based Hough Transform (D-KHT) [28] is a Hough transform-based plane detection algorithm that accepts depth images as input.

pre-processing clustering:

- quadtree subdivision
  - whole image is root
  - each node with more than threshold s_{ms} samples: coplanarity test with PCA
  - all nodes with less than s_{ms} inliers are ignored in the voting process
  -
- Summed area tables (SAT) computation for PCA efficiency
  - the covariance and mean values can be calculated in constant time that way
  - pre-compute (9) SATs for each node

plane detection given set of approx coplanar point clusters in leaf node

- the best fitting plane is described by its mean point and a unit normal vector (which is the associated eigenvector to the least eigenvalue of the cov matrix)
- the gaussian kernel that weights votes from the cluster in the parameter space of the plane normal
- calc cov matrix of the gaussian kernel can be computed with first order error propagation **mathe**
- cluster votes in spherical accumulator map A
  - $\theta \in [-\pi, +\pi], \phi \in [0, \pi], \rho \in [0, \rho_{high}]$
  - ρ_{high} is the farthest point from camera to a sample
  - the amount of samples within rho and are defined by params N_ρ, N_ϕ
- voting:
  - for given kernel that is associated with a cluster:
  - increment bins of A that fall within two standart deviations of the mean **formel**
  - starting at specific bin, neighbors are flood filled with **mathe** als termination condition

- after voting:
  - compute convolution of accumulator with six-connected filter (smoothing of map)
  - hill climbing to detect peaks: plane is detected if the bin of a mean plane is a local maxima

post processing given set of parameters corresponding to detected planes

- sort planes by relevance
- relevance is defined by the summation of the weights associated with the clusters that contributed to the peak

The normal vector and the distance ρ define the normal equation of the plane supporting a subset of 3d point entries.

2.5.9 Depth Dependent Flood Fill

Depth Dependent Flood Fill (DDFF) is a region growing-based algorithm that detects planes in organized point clouds. Der algorithmus die weiterentwicklung des vorgängers [22]. **pre processing** None

plane detection seed selection:

- seed size parameter σ_{seed} defines cube around possible seed point
- check for distance discontinuities **formeln für beide**
- check for curvature discontinuities

region growing:

- perform region growth over spans of horizontal pixels
- member ship criteria:
  1. perpendicular distance of point to plane defined by seed point and its normal
  2. euclidian distance function to ensure pixel neighbors are also 3d neighbors
  3. assure the pixel is not 0 or nan

plane merging: region growing yields a set of oversegmented planes repeat the following until no planes are merged during an iteration, usually no more than 5 iteration is enough

- build neighborhood graph defined by euclidian distance of edge pixels

- iterate BFS through graph
- merge if two planes fullfil a condition **formel**
- if merged, recalculate centroid, normals and update graph accordingly

post-processing filling gaps:

- two passes:
  1. scan top to bottom
  2. bottom to top
- in both passes, identify horizontal strips of unmarked pixels that have the same neighbor either above or below and mark them with that neighbors label

2.5.10 PlaneNet

PlaneNet [16] ist a deep learning-based approach to piece-wise planar reconstruction of a scene from a single RBB-Image.

hier würde sich wahrscheinlich die grafik anbieten

- built upon Dilated Residual Networks (DNR)
- followed by three predicion tasks
  1. plane parameters
     - global average pooling, reduce feature map size to 1x1
     - fully connected layer to produce Kx3 plane parameters. (K=10 in experiments)
     - loss function: order agnostic, based on chamfer distance metric for regressed plane parameters
  2. segmentation mask
     - start: pyramid pooling module
     - then: convolution layer. produces K+1 channel likelihood maps for planar and non-planar surfaces
     - finally: Dense conditional random field (DCRF), based on fast inference. meanfield iterations: 5/10 (training/testing)
     - loss function: softmax cross entropy to supervise segmentation training (pixel level)

3. non-planar depthmap
   - start: pyramid pooling module
   - followed by convolution layer, produces a 1-channel depthmap
   - loss function: sum of squared depth differences between GT and predicted plane or non-planar depthmap (weighted by probabilities)

2.5.11 PlaneRecNet

PlaneRecNet [31] is a deep learning-based approach for piece-wise plane detection and reconstruction that takes RGB images as input.

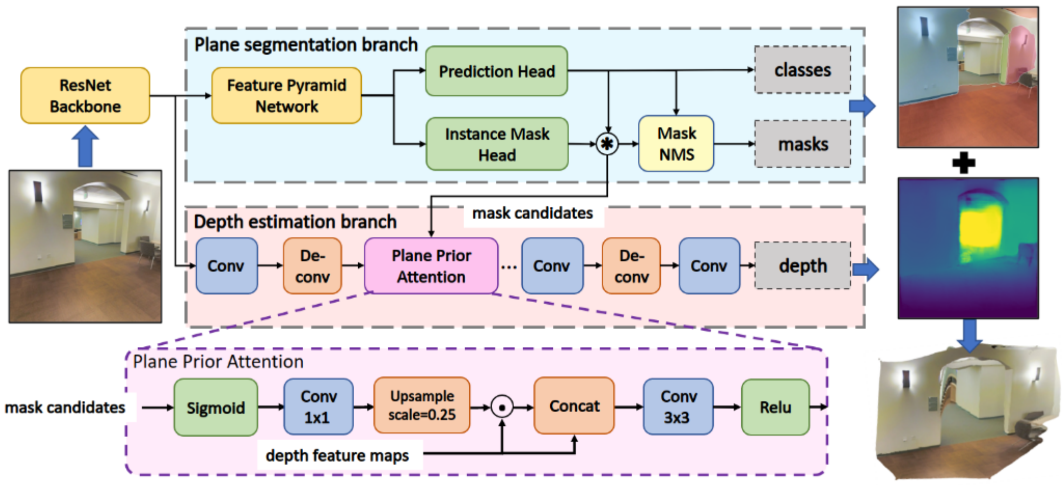


Figure 2.3: The PlaneRecNet Architecture. Taken from the respective paper [31]. **die grafik gibts evtl nochmal in scharf im repo**

approach: focus on high quality global depth estimation so that plane params can be computed by PCA or RANSAC with the plane segmentation and depth prediction.

- ResNet backbone output as input for the following
- two seperate branches
  - plane segmentation branch
    * in general: leight weight configuration of SOLOV2 [30] adapted for planar segmentation

    * more specific: Feature pyramid network, followed by a prediction head and an instance mask head
    * returns classes (obt. from pred. head)
    * and masks obt. from non maximum suppresion of pred head and inst. mask head
  - depth estimation branch
    * repeated convolution and de-convolution
    * after the first iteration: plane prior attention module with mask candidates as input which are obtained from dynamic convolution between the prediction head and instance mask head from the plane segmentation branch
    * finally: returns depth
  - plane prior attention module:
    1. sigmoid function
    2. 1x1 convolution
    3. upscale(=0.25) with interpolation
    4. multiplication and concatenation of mask candidates with depth feature maps
    5. concatenation of monocular depth estimation and depth feature maps
    6. 3x3 convolution
    7. relu activation
- i guess classes+masks are combined into planes
- which is combined with the depth for a reconstructed scene

2.5.12 PlaneRCNN

RCNN [17] is a deep neural architecture that detects planar regions and reconstructs a piecewise planar depth map from RGB-Images.

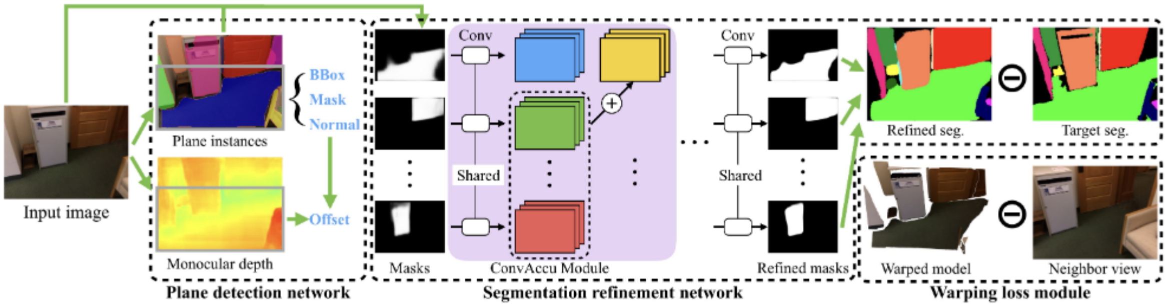


Figure 2.4: The PlaneRCNN Architecture. Taken from the respective paper [17]. **die grafik gibts evtl nochmal in scharf im repo**

3 main components:

1. plane detection network
   - predicts 3d plane parameters and segmentation mask
   - built upon Mask-R-CNN. here: "planar" as object instance
   - maskrcnn detects instances and estimates segmentation masks
   - then, estimate plane normal and offset d:
     a) predict normal per planar instance
     b) estimate depth map
     c) algebraically calculate plane offset **formel?**
2. segmentation refinement network
   - takes the planar regions from 1. and optimizes the masks
   - uses convAccu: combines convolution layer and an accumulation scheme
   - aggregates information from all masks
   - at the end, refined plane masks are concatenated and compared against target masks with a cross entropy loss
3. warping loss module (only during training)

- uses a second angle on the same scene to ensure consistency
- nearby view is 20 frames ahead during training
- calculate nearby view by warping every pixel in the nearby view to the current one given the camera intrinsics
- loss function is a simple absolute error between the current view and the warped nearby view.

2.6 Datasets

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[3]	UPC	Y	N	planes
TUM [26]	DI	Y	Y	trajectory

Table 2.3: Popular Datasets. The *GT*(Ground Truth) column specifies what the ground truth of each dataset represents.

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 $7 \cdot 10^6$ Furthermore, the average amount of points per scene is $\sim 10^6$ with an average file size of $\sim 34mb$.

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

[3] https://shop.leica-geosystems.com/de/leica-blk/blk360/dataset-downloads

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.

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).

$$Precision = \frac{|\{\text{correct values}\} \cap \{\text{obtained values}\}|}{|\{\text{obtained values}\}|} \tag{2.3}$$

$$Recall = \frac{|\{\text{correct values}\} \cap \{\text{obtained values}\}|}{|\{\text{correct values}\}|} \tag{2.4}$$

$$\textit{f1-score} = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{2.5}$$

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.

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

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

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

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

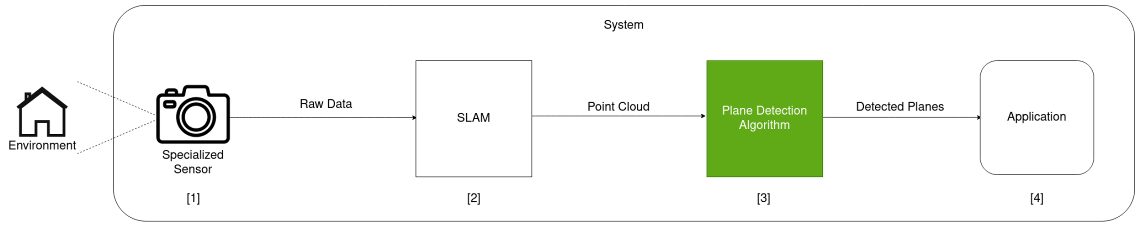


Figure 3.1: The procedure of the plane detection process. The specialized sensor records data ([1]), which is passed to a SLAM algorithm ([2]). After map assembly, a point cloud is handed to a plane detection algorithm ([3]). The detected planes are given to a use-case-specific application ([4]).

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.

We detail the different types of input in Section 2.3.

To reiterate, 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)

- image:
  - Depth (DI)
  - RGB (RGBI)

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 [31]	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**

RSPD, 3D-KHT and OBRG construct an octree (OC) during their pre-processing phase. Additionally, RSPD and OBRG perform an initial estimation of normals (NE). OPS estimates the normal vectors for a randomly chosen sample set of points. During post-processing, 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. RSPD and 3D-KHT do not perform post-processing. The pre-and post-processing steps are summarized in Table 3.2.

	RSPD	OPS	3D-KHT	OBRG
Pre	NE	NE	OC	OC + NE
Post	/	Merge	/	Refinement

Table 3.2: Pre-processing and post-processing steps of the plane detection algorithms. RSPD and 3D-KHT do not have any post-processing steps.

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 2.3).

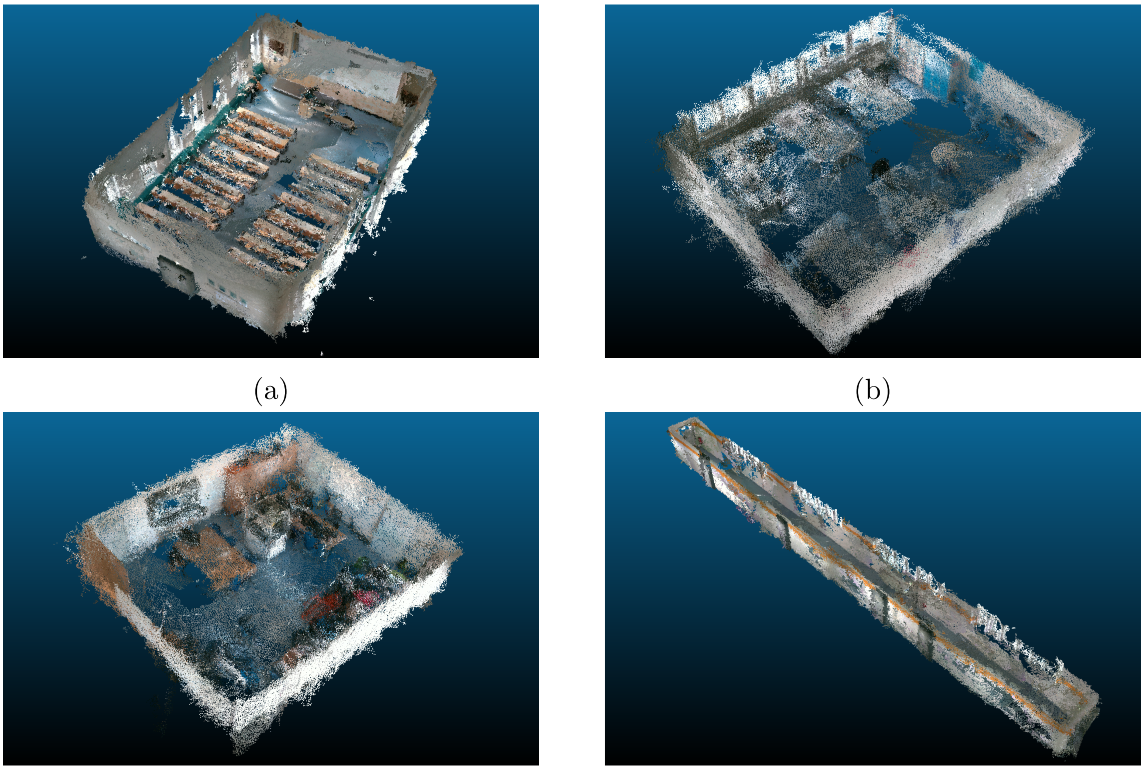
In Subsection 3.1.2, we determined unorganized point clouds as the type of input. Furthermore, we focus on plane detection in real environments in this work. Because most datasets do not conform to these two requirements, only *2D-3D-S* and *Leica* remain. Since we, additionally, want to detect planes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



(a) (b) (c) (d)

Figure 3.2: The recorded point clouds for each scene type: (a) auditorium, (b) conference room, (c) office and (d) hallway. The ceilings have been manually removed for visualization purposes but remain in the dataset for the experiments.

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[1] and OPS[2] 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*[3] and the C++ API of the Point-Cloud Library[4], *libpcl-dev*.

[1] https://github.com/abnerrjo/PlaneDetection
[2] https://github.com/victor-amblard/OrientedPointSampling
[3] https://eigen.tuxfamily.org/index.php
[4] 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[5]. Since the laptop we use does not run Windows, we use *cmake-converter*[6] 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 obtain the following parameter values through empiric experiments on multiple 2D-3D-S scenes:

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

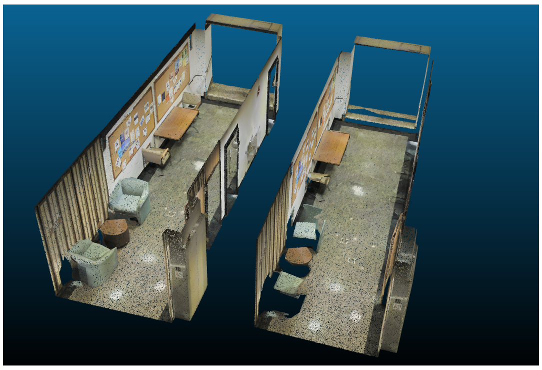
To determine a region's planarity, we calculate the number of points that fit the best fitting plane within a predefined threshold. If that number supersedes the proposed 70%-90%, depending on the expected noise, the region is considered planar [29, Section 3.4].

4.3 2D-3D-S

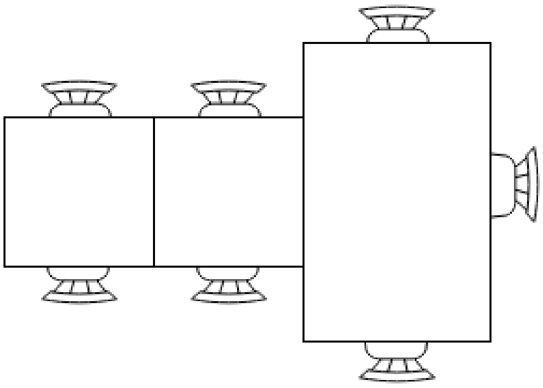
The 2D-3D-S dataset provides a ground truth in form of annotated point clouds corresponding to 13 object classes. Since these annotated objects are not always planar, we cannot use them for the evaluation of plane detection algorithms. Thus, we create a ground truth that focuses on planar structures. We use the open-source 3D point cloud

[5]https://www.inf.ufrgs.br/ oliveira/pubs_files/HT3D/HT3D_page.html
[6]https://cmakeconverter.readthedocs.io/

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
							139/272	**3410**

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

Reiterating Section 3.2, we select four scene types of the 2D-3D-S dataset for the recording of the self-created FIN dataset. Namely, these scene types are *auditorium, conference room, hallway,* and *office*. As the name of the dataset suggests, we record these scene types inside the *Faculty of INformatik* (FIN) at Otto-von-Guericke-University in Magdeburg. Running *realsense-ros* and holding our cameras, we walk through the corresponding parts of the building while scanning to the best of our ability. We save each incremental map update to a file for later usage.

Given the differences in spatial dimension, the recordings of each scene also differ in length and size. The auditorium scene has a total of 296 individual time frames, the conference room scene has 113, the hallway has a total of 174, and the office has 125 time frames. The FIN dataset, thereby, has a total of 708 time frames.

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. By creating a ground truth for only the last frame of each scene, we substantially reduce the time invested in this task. 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.

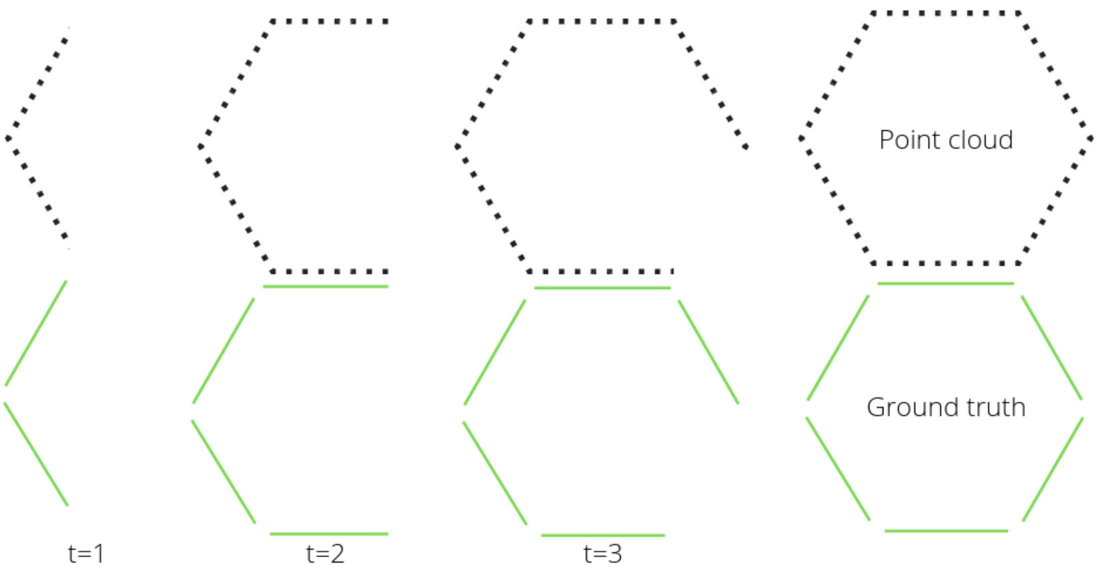


Figure 4.2: Dynamic ground truth generation. All planes that are included in *Ground Truth* are cropped depending on the available point cloud at each time t

5 Evaluation

In this chapter, the previously selected algorithms are uniformly compared. We present the results.

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. We conduct both experiments on an AMD Ryzen 7 5800H CPU with 16GB of RAM. 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. Note that we use the detected planes for the calculation, however, we are not using them directly as a measure of performance due to the subjective nature of manual ground truth segmentation. We calculate the F1-score, as it is the harmonic mean of the precision and the recall. In practice, this means that an algorithm has to yield sufficient *precision* and *recall* results to score a sufficient *F1-score*. The F1-score, thereby, is the primary metric for the quantitative comparison of algorithms in this work. However, we report the *precision* and the *recall* for thorough analysis.

Time Measurements In addition to the accuracy, we are interested in the real-time applicability of an algorithm. We introduce two definitions of *real-time* in Section 3.3. To reiterate, we consider an algorithm to be *totally real-time* (RT_{tot}) applicable if its total runtime is smaller than $1s$. Furthermore, we consider an algorithm to achieve *real-time plane detection* (RT_{calc}) if the total calculation time excluding the pre-processing step runs faster than $1s$. Having two definitions of real-time, it is helpful to introduce two metrics of computation time.

When recording a real environment, the point clouds grow incrementally with each map update. Therefore, the average calculation times alone are of limited significance, as we expect shorter times for the start of a recording and longer times for the most recent. The first metric is, therefore, the relationship between the point cloud size and the calculation time.

Following the separation of algorithms into phases, we accordingly divide the calculation into pre-processing, plane calculation, and post-processing. The respective calculation times are in the following, referred to as t_{pre}, t_{calc}, and t_{post}. This separation allows for an in-depth analysis. Furthermore, it allows us to determine whether an algorithm is RT_{calc}. To determine whether an algorithm is RT_{tot}, we consider the total computation time t_{tot}, which is the sum of the individual times:

$$t_{tot} = t_{pre} + t_{calc} + t_{post} \tag{5.1}$$

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. Furthermore, we determine an appropriate parameterization for the FIN dataset through empirical experiments. These experiments include multiple tests of different combinations of values on all scenes of the FIN dataset, the ranges of which are specified in the respective paragraph of each algorithm.

5.1.2.1 RSPD

die abkürzungen werden sicherlich im BG erklärt.

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

Table 5.1: 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[1]. These parameters include the maximum octree level l_O, the minimum number of samples per leaf node ε, the Maximum Outlier Ratio MOR per plane, the size of the nearest neighborhood k, the Maximum Normal Deviation MND, and the Maximum Distance to Plane MDP. 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. **reducing how much? - test it!**

FIN Multiple experiments with different values had been conducted. The individual ranges are: $l_O \in [8, 12], \varepsilon \in [15, 150], MOR \in [15, 40], k \in [30, 90], MND \in [50, 70]$, and $MDP \in [0.15, 0.4]$. Through these experiments, we were not able to find a configuration that yields considerably better results than the default parameterization. Therefore, no parameter modifications for the FIN dataset are made.

5.1.2.2 OPS

ja, die parameter werden im background *sicherlich* erklärt

Experiment	α_s	KNN	θ_h	θ_N	p
2D-3D-S	3%	30	0.05	100	0.99
FIN	3%	**90**	**0.35**	100	0.99

Table 5.2: 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 5.2. We use a sampling rate α_s of 3% and a neighborhood KNN of 30 for the estimation of normal vectors. Additionally, we use a distance threshold θ_h of 0.05(m). Furthermore, we set the inlier threshold θ_N to 100 and the probability for adaptively determining RANSAC iterations p to 0.99, as proposed in [27, Section 4A].

[1]https://github.com/abnerrjo/PlaneDetection

FIN We ran experiments on the FIN dataset with different parameter values in the respective ranges: $\alpha \in [0.3\%, 6\%]$, $KNN \in [30, 150]$, $\theta_h \in [0.05, 0.5]$, $\theta_N \in [50, 1000]$, and $p \in [0.90, 1.0]$. We choose the configuration, that shows a balance between speed and accuracy, namely the following. 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 θ_h because an increase in sensor noise ultimately thickens the recorded planes. Both modifications are highlighted in bold in the second row of Table 5.2.

5.1.2.3 3D-KHT

Experiment	ϕ_{num}	ρ_{num}	s_{level}	s_{ps}	d_{max}	s_α	s_β
2D-3D-S	30	200	2	0.002	0.08	18	6
FIN	30	**100**	2	0.002	**0.1**	**8**	6

Table 5.3: Parameter configuration of 3D-KHT used for the experiments.

2D-3D-S The parameter configuration is shown in Table 5.3. We use an accumulator discretization of 30 and 200 for ϕ and ρ, 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 s_{ps}, 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 the plane isotropy tolerance s_β to 6, as proposed in [15, Section 3.1]. In contrast, using a plane thickness tolerance s_α value of 18 seemed to yield better results than the proposed 25. This was determined by a small set of experiments with s_α values ranging between 15 and 33.

FIN For the FIN experiment, we modify the values of ρ_{num}, d_{max} and s_α to accommodate for the higher levels of noise. These values are obtained by performing tests with different parameterizations. Precisely, we used the following parameter ranges combinations of values therein: $\phi_{num} \in [20, 100]$, $\rho_{num} \in [50, 600]$, $s_{level} \in [1, 5]$, $\theta_N \in [50, 1000]$, and $d_{max} \in [0.05, 2.0], s_\alpha \in [5, 23]$, and $s_\beta \in [4, 8]$. Reducing ρ_{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_α to allow for slightly thicker, e.g. noisier, planes to be detected. The modification of parameters is highlighted in bold in Table 5.3.

5.1.2.4 OBRG

Experiment	l_{max}	θ_{res}	θ_d	θ_{ang}	θ_M	θ_p
2D-3D-S	5	0.08	0.08	0.18	5000	90%
FIN	5	**0.22**	**0.2**	**0.2**	5000	**70%**

Table 5.4: Parameter configuration of OBRG used for the experiments.

2D-3D-S The used configurations for the experiments are shown in Table 5.4. The parameterization was determined during the implementation process. Due to the low level of noise, we assign a very small tolerance to the residual thresholdθ_{res} and the distance threshold θ_d. Additionally, we assign a high planarity threshold value of $\theta_p = 90\%$.

FIN We performed tests with different parameterizations, individually ranging as follows. $l_{max} \in [4\%, 7\%]$, $\theta_{res} \in [0.05, 0.3]$, $\theta_d \in [0.05, 0.3]$, $\theta_{ang} \in [0.13, 0.3], \theta_M \in [200, 6000]$, and $\theta_p \in [70, 90]$. Due to higher levels of noise, and thus, thicker walls, we increase θ_{res}, θ_d, and the angular divergence threshold θ_{ang}. According to [29, Section 3.4], the planarity threshold θ_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 5.4.

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.

5.2.1 2D-3D-S

We ran RSPD, OPS, 3D-KHT, and OBRG on 139 scenes of the 2D-3D-S dataset. Subsequently, for each scene, the precision, recall, and F1-score of each algorithm were calculated. The computation times were measured and divided into pre-processing, plane detection, and post-processing. Table 5.5 shows the average of the computed results for each algorithm. The rightmost column gives the total computation time t_{tot}. The largest values of the accuracy and the smallest average values of the times are indicated in bold. It is to be noted that no lowest value of t_{post} is indicated since RSPD and

3D-KHT have no post-processing steps and, therefore, "per default", spend less time in this step.

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

Table 5.5: 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}). The rightmost column shows the average total calculation time s_{tot}. Note, that the absence of post-processing steps is denoted as "/". All times are measured in seconds.

Accuracy RSPD has the overall highest accuracy with a precision of ~85%, a recall of ~90%, and an F1-score of ~87%. OPS supersedes RSPD with a precision value of ~89% but scores significantly lower recall and F1-score values. 3D-KHT yields precision, recall and F1-score results in the range of ~71% and ~79%. OBRG has a high precision value of ~81%, however its recall and F1-score values are the lowest out of all algorithms with ~67% and ~71%, respectively.

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. According to these values, no algorithm achieves *total real-time* applicability. RSPD and 3D-KHT are very close to falling under the threshold of $1s$ for $real - timeplanedetection$.

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.

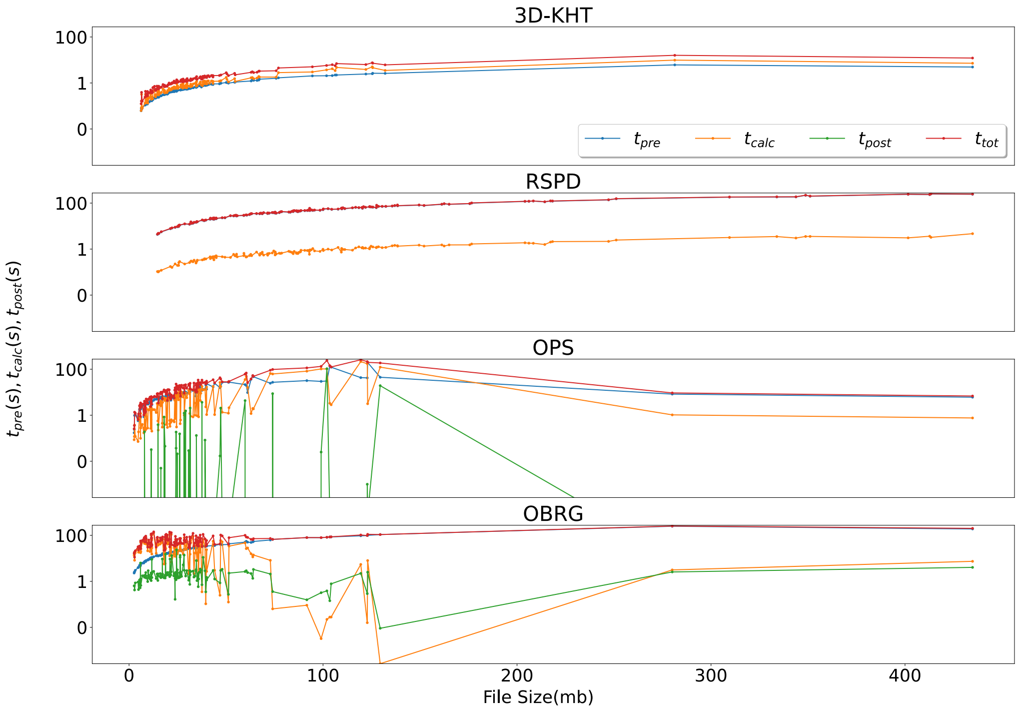


Figure 5.1: The time spent in pre-processing t_{pre}, plane detection t_{calc}, post-processing t_{post}, and the total calculation time t_{tot} per point cloud file size of the 2D-3D-S dataset. Note, that both y-axes are scaled logarithmically to the base of ten.

The computation times of 3D-KHT do not seem to strongly relate to the size of the point cloud. Both the duration of the pre-processing and the duration of the plane detection initially grow linearly but do not show a large growth even with large jumps in point cloud size. The difference in calculation times between $100mb$ and $>200mb$ is hardly noticeable.

The computation times of RPSD show a similar relation, with the difference that the pre-processing of RSPD takes significantly longer. As with 3D-KHT, the duration of plane detection seems to have an upper limit.

In general, the pre-processing time of OPS has a linear growth depending on the size of the point cloud. The duration of plane detection also shows a linear relationship, with the difference that there is a certain level of fluctuation. The post-processing times are negligible for the most part, given the values below 10^0. The irregularity of the spikes gives reason to assume that it primarily depends on the structure of the recorded environment rather than size alone.

The duration of the pre-processing of OBRG also shows a linear relationship with respect to the point cloud size. The average high t_{pre} values from Table 5.5 are also reflected here since most values are above $\sim 10^1$. The t_{calc} values of OBRG do not appear to be dependent on the size of the point cloud, as the computation times tend to decrease with growing point clouds. A possible reason could be the fixed value of the octree levels l_{max}. An adaptive determination of octree size would likely lead to a more balanced curve. Since the post-processing phase operates on the plane detection phase data, the curve behaves similarly, supporting the sub-optimal parameterization argument.

Summary 2D-3D-S Experiment OPS has the highest value in Precision, while RSPD achieves the highest Accuracy and F1 score. 3D-KHT has the lowest total computation time t_{tot} with an average of $1.74s$. With $>60s$, RSPD and OBRG have the largest t_{tot} values among the algorithms, with t_{pre} accounting for the majority for RSPD.

Adhering to the definitions of real-time, no algorithm achieves real-time applicability in this experiment.

Figure 5.1 shows that the runtimes of 3D-KHT are the smallest. RSPD also has low t_{calc} values but consistently spends the longest time in pre-processing. Moreover, the pre-processing times of all algorithms seem to be generally dependent on the size of the point cloud. The plane detection runtimes of OPS and OBRG fluctuate. However, OPS fluctuates less than OBRG. The post-processing runtimes of OPS are negligible overall. The t_{post} values of OBRG are stable at $2.61s$ on average, except for medium cloud sizes, see table 5.5.

5.2.2 FIN

Each of the total 708 time frames of the FIN data set was processed by each algorithm. Subsequently, we evaluated each time frame separately, i.e., by calculating the precision, the recall, and the F1 score. Additionally, we measured the computation times of each time frame for all algorithms.

The average results over all time steps of all scenes of the FIN experiment are presented in Table 5.6. The highest values are written in bold for precision, recall, and the f1-score, as are the lowest times of each calculation step.

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

Table 5.6: 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}). The rightmost column shows the average total calculation time s_{tot}. Note, that the absence of post-processing steps is denoted as "/". All times are measured in seconds.

Accuracy OPS has the highest average precision of the algorithms, with almost 70%. RSPD, on the other hand, achieved the highest values for recall and the F1-score. 3D-KHT and OBRG achieved a similar precision, but for Recall and F1 score, however, 3D-KHT has higher values than OBRG by approx. 13% and approx. 17%, respectively. RSPD thus achieves the highest overall accuracy, while OBRG achieves the overall lowest.

Average Time RSPD needs the most time in pre-processing among the algorithms. In contrast, RSPD has the shortest time spent during plane detection, with 0.19 seconds. Overall, 3D-KHT needs the shortest time for the complete computation t_{tot} of a time step with an average of 0.43 seconds. OPS achieves comparatively average times with about 6 seconds, and OBRG takes the longest overall to compute a time step with more than 20 seconds. RSPD, and 3D-KHT achieve *real-time plane detection*. Additionally, 3D-KHT also achieves *total real-time* applicability, with an average of $ttot = 0.43$. When excluding the pre-processing time, OPS almost achieves *real-time plane detection* with a value of $1.02s$, thereby taking $0.02s$ too long.

Relationship of Time and Size We want to consider the relationship between the size of the point cloud and the corresponding calculation time. In Figure 5.2, we compare the processing times of each time step to the file size of the *auditorium* scene of the FIN data set.

Note, that we created similar graphs for the other scenes of the FIN dataset. However, they do not introduce any new information in argumentation and are presented in the appendix **$REF** for completeness and spatial reasons. The auditorium scene was selected because it represents the longest recording, and thus contains the most data.

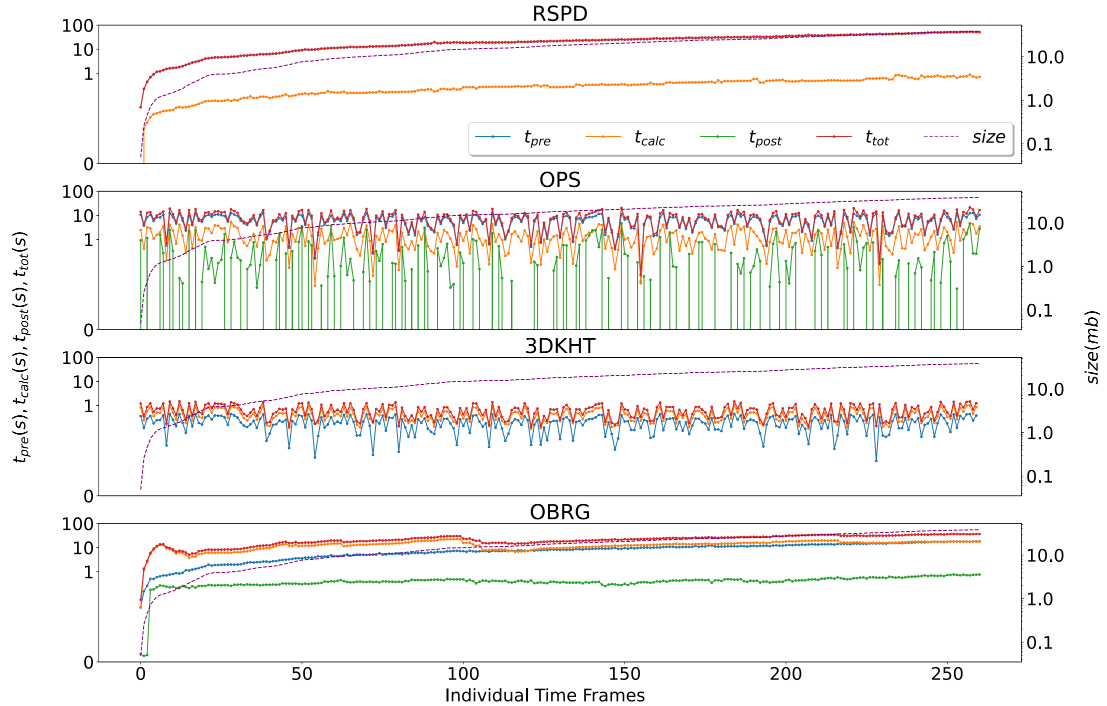


Figure 5.2: Time spent in pre-processing t_{pre}, plane detection t_{calc}, and post-processing t_{post} of the hallway scene and cloud sizes $size$ of each time frame. Note, that both y-axes are scaled logarithmically to the base of ten

The pre-processing times of RSPD and OBRG are noticeably proportional to the point cloud size. In contrast, for OPS and 3D-KHT, the pre-processing times seem to correlate with the plane detection times since both show similar spikes. The plane detection steps of OPS and 3D-KHT do not seem to depend on the point cloud size, as both seem to be limited by an upper bound. The pre-processing and plane detection times of OBRG grow rapidly in the beginning but afterward, show a linear growth in relation to the cloud size. The post-processing times of OPS fluctuate between 0 and the duration of plane detection. After the spike in the beginning, the t_{post} values of OBRG seem to be consistent.

Summary FIN Experiment OPS has the highest average precision, and RSPD has the largest percentages in recall and F1 score. Additionally, RSPD has the lowest t_{calc} value among the algorithms, with an average of $0.19s$ per time frame. In contrast, RSPD has the longest pre-processing time with $14.36s$ on average. The algorithm with the shortest pre-processing and the shortest total time is 3D-KHT with $t_{pre} = 0.14$ and $t_{sum} = 0.43$, respectively.

In general, the calculation times of RSPD and OBRG seem to depend on the point cloud size. However, the time RPSD spends in pre-processing is significantly higher than in its plane detection step. In contrast, the pre-processing and plane detection times of OBRG seem to converge at the end. The calculation times of OPS and 3D-KHT seem to be independent of the point cloud size and consistently stay under an upper bound. However, 3D-KHT has a smaller upper bound than OPS.

5.2.3 Comparison

When comparing Table 5.5 and Table 5.6, 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.

Observing Figure 5.2 and Figure 5.1, common features are noticeable:

The curve shape of RSPD is very similar for both experiments, linearly dependent on the point cloud size, and the t_{pre} accounts for 99% of the total calculation time t_{tot}. In both experiments, the post-processing time of OPS fluctuates. The post-processing of OBRG seems to be mostly stable around a small value ($\sim 10^{-1}$). 3D-KHT scores very low processing times in both experiments.

However, there are also notable differences. Whereas the calculation times of 3D-KHT seem to be proportional to the point cloud size in the 2D-3D-S experiment, they show no such relation during the FIN experiment. It is, however, noteworthy that the cloud sizes widely differ between the experiments, as the maximum size of the FIN experiment ($\sim 30mb$) is very small, compared to largest scene of the 2D-3D-S dataset ($>400mb$). Nonetheless, since Figure 5.1 portrays a proportionality, even for smaller clouds, the reason for different curves is likely the difference of parameterization.

5.3 Summary

OBRG does not achieve the highest or lowest values in any experiment. OPS has the highest precision value in both experiments. Due to Recall and the F1-score, RSPD has the highest overall accuracy in both experiments. The pre-processing of RSPD takes the longest time of all algorithms. In contrast, RSPD has the smallest average t_{calc} value in the FIN experiment. 3D-KHT has the lowest total computation time in both experiments, and in the 2D-3D-S experiment, it outperforms the plane detection time of RSPD.

6 Conclusion and Future Work

6.0.1 Summary

Use case, scenario, real-world application, current problem Viele AR/VR anwendungen benutzen plane detection.. Echtzeit ist wichtig.. Es werden spezielle Sensoren benötigt, ein SLAM algo und ein geeigneter plane detection algo.. Die Auswahl des besten plane detection algorithmus ist nicht trivial..

Daher das thema dieser arbeit Wir vergleichen algorithmen mit fokus auf die echtzeitfähigkeit wir haben 2 definitionen für echtzeit angegeben.. Wir beschränken uns in dieser arbeit auf plane detection mit intel realsense technologie das beinhaltet 2 kameras und die dazugehörige software (+SLAM)

algorithmen Es wurde die aktuelle literatur gelesen und eine liste aus algorithmen erstellt. Es wurden metriken zum objektiven und oberflächlichen Vergleich der algorithmen angegeben basierend auf den metriken wurden vier algorithmen für einen vergleich ausgewählt

Testscenario / datensatz Da die ausgewählten algorithmen nicht auf dem selben Datensatz getestet wurden haben wir geeignete ausgesucht. Wir haben uns für den 2D-3D-S Datensatz entschieden.. motiviert durch die abwesenheit eines datensatzes mit temporaler komponente haben wir einen eigenen datensatz erstellt

Experimentaufbau Da wir zwei datensätze haben haben wir auch zwei experimente Zunächst wurden bewertungsmetriken der datensätze eingeführt: precision, recall, f1, dazu wurden die berechnungszeiten genau gemessen und es wird unterschieden in preprocessing, plane detection und post-processing. diese experimente wurden seperat durchgeführt und ausgewertet im anschluss darauf wurden die ergebnisse beider Experimente verglichen

Ergebnisse der Experimente die ergebnisse lassen sich zusammenfassen in: 3D-KHT ist der insgesamt schnellste RSPD ist der insgesamt präziseste

wir haben zwei definitionen von echtzeit gemäß dieser definitionen und anhand der ergebnisse aus dem letzten kapitel gilt: (in einem realwelt environment)

- 3D-KHT $\in RT_{tot}$, da $t_{tot} < 1$
- RSPD $\in RT_{calc}$, da t_{calc}¡ 1

6.0.2 Fazit

OBRG ist nicht echtzeitfähig.

Da das pre-processing von RSPD viel zu lange dauert ist RSPD insgesamt nicht total echtzeitfähig, die echtzeitfähigkeit beschränkt sich somit auf die plane detection phase.

OPS erzielt im FIN experiment so grade werte die für eine plane detection echtzeit sprechen würden. Auch hier dauert das pre-processing zu lange um von einer totalen echtzeit sprechen zu können.

3D-KHT ist im FIN experiment deutlich unter der Schranke von einer sekunde, damit in einem realen enviroment total echtzeitfähig.

Limitationen der ergebnisse Ein Faktor der langsamen laufzeit von OBRG ist definitiv die implementation. Python ist aus diversen Gründen magnituden langsamer als C++ und es wurde bei der anfertigung der implementation die präzision priorosiert da durch die wahl der programmiersprache sowieso nicht von echtzeitfähigkeit ausgegangen wurde.

3D-KHT ist der schnellste algorithmus, lässt aber leider in sachen accuracy viele prozente liegen.

Algo x ist der beste, mögl verbesserungen Das einzige was RPSD von einer eindeutigen dominanz trennt ist die Dauer des preprocessings. In RPSD's pre-processing phase werden die normalen der Punktwolke berechnet. Es ist möglich dass die normalen noch vor der plane detection phase aus Figure 3.1 berechnet werden. Zb kann sogar die intel realsense technologie die normalen exportieren. Dies würde nach angemessenen anpassungen des algorithmuses die Berechnungszeiten auf t_{calc} reduzieren. Somit würde RSPD ebenfalls total echtzeitfähig sein.

Diese anpassung des experiments würde einen bias gegenüber RSPD darstellen, weswegen wir das unterlassen haben und als weitere aufgabe nach dem abschluss dieser arbeit ansehen.

Ebenso interessant sind die Laufzeiten einer optimierten implementierung von OBRG in C++.

Die parametrisierung von 3D-KHT ist definitiv die komplizierteste aus den algorithmen und hat daher das größte optimierungspotential. Das ist in dem fall ein doppelschneidiges schwert, da zuerst eine optimale/ausreichende parametrisierung gefunden werden muss, bevor der algorithmus das volle potential entfaltet.

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.

[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 Labbé and François Michaud. “RTAB-Map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation: LABBÉ 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.

[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.

[21] Pedro F. Proença 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, Marceli 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] Xinlong Wang et al. “Solov2: Dynamic and fast instance segmentation”. In: *Advances in Neural information processing systems* 33 (2020), pp. 17721–17732.

[31] 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`.

[32] 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)