FACULTY OF
COMPUTER SCIENCE

Otto-von-Guericke-University Magdeburg

Faculty of Computer Science

Institute for Intelligent Cooperating Systems

Bachelor's 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

November 29, 2022

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

Abstract

Planar structures account for a significant portion of indoor man-made environments. With advances in the field of Augmented Reality (AR), the automatic detection of planar surfaces has become essential for recent AR applications. Often, these applications operate under a strict temporal constriction, also referred to as *real-time*. Naturally, this time restriction applies to the integrated plane detection algorithm as well. The technology that provides *real-time* plane detection already exists. However, for different reasons, these devices are often not suitable for the average consumer. This motivates the utilization of consumer off-the-shelf hardware. Additionally, an appropriate plane detection algorithm is needed. Decades of research yield a wide variety of different approaches. As these methods are predominantly evaluated scientifically, the real-world applicability poses an open question. Moreover, the inherent incomparability of most plane detection algorithms renders a selection non-trivial.

This work evaluates the real-world applicability of *real-time* plane detection algorithms. After considering current state-of-the-art plane detection algorithms, we select four algorithms, namely RSPD [2], OPS [58], 3D-KHT [33], and OBRG [60]. In a similar approach, we select the 2D-3D-S dataset and compose the novel FIN dataset. We introduce a definition of *real-time* and perform experiments on both datasets. Subsequently, we compare the respective results. The results show that 3D-KHT is the only *real-time* applicable plane detection algorithm in a realistic environment.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Man-made environments usually consist, to a large extent, of planar structures. As the *Manhattan-world* assumption dictates, the general alignment of most urban scenes, both indoors and outdoors, is based on a three-dimensional cartesian grid [10]. Thus, the assumption indicates that urban scenes are primarily comprised of planar surfaces which lie orthogonal to each other.

Due to this large amount of planes in everyday environments, automatic detection of these planes is growing in relevance: Plane detection is an essential component in numerous Augmented or Virtual Reality (AR/VR) applications and systems [27, 56]. Therein, planes are vital for general scene understanding as they lay the foundation for scene reconstruction [1], object recognition [44, 47], video games [9], and are even used in applications that support people with disabilities, e.g. visual impairments [37, 53].

Many of these applications operate under specific time constraints: The application could navigate a visually impaired person into a nearby wall if the plane detection is delayed, just as a noticeable lag in a video game caused by plane detection could spoil the user experience. Strict temporal constraints are often broadly referred to as *real-time*. The definition of *real-time* usually derives from the frequency of new sensor updates [12]. Since the process of plane detection is often an integral part of these systems [62, 11, 28], these constraints also apply there.

Real-time plane detection is already possible, though expensive hardware is often needed as a sensor's price increases with its precision and included functionality. For instance, AR devices like the *Microsoft HoloLens 2* and imaging laser scanners like the *Leica BLK360* produce very precise representations of the surrounding environment. Moreover, the *HoloLens 2* can perform plane detection as part of its *Scene Understanding Software Development Kit*[1]. Therein, a recorded environment is represented as a dense

[1] https://learn.microsoft.com/en-us/windows/mixed-reality/design/scene-understanding

triangle mesh in which planes are detected and subsequently assigned a plane category, e.g., walls, ceilings, and floors. Nevertheless, the affordability of the *BLK360* is questionable with a starting price of ∼$19.000. While the starting price of the *HoloLens 2* may be affordable at ∼$3.500, due to the software being closed source, we still consider the *HoloLens 2* non-practical.

Through this lack of affordability and practicability of high-end sensors, the usage of consumer off-the-shelf hardware is gaining interest. With AR development platforms like *ARKit*[2] or *ARCore*[3], it is generally possible to perform AR *plane detection* on mobile phones[4]. However, the choice of development environment and the used sensor is arbitrary. Being representatives for consumer off-the-shelf hardware, we use the *Intel RealSense* cameras T265 and D455 (see Section 2.2) in this work. The cameras have a combined starting price of ∼$600 and an open source software development kit, namely *librealsense*[5], is provided.

In addition to the used sensors, selecting an appropriate plane detection algorithm is important as well. Decades of research on plane detection yielded numerous algorithms, and many are reported to be *real-time* applicable by the respective authors [33, 49, 64, 67, 17, 30]. While generally achieving the same goal, i.e., the detection of planar structures, notable differences in methodology exist: We can usually differentiate between algorithms by their type of input, the format of detected planes, additional hardware requirements, and core algorithm (see Section 2.4). Furthermore, most algorithms have been evaluated using different datasets and metrics. It is, therefore, impossible to assess the real-world applicability of most algorithms through the reported findings because the corresponding authors evaluated them in a scientific context and environment.

1.1 Goals

This thesis deals with a uniform comparison of plane detection algorithms. Through this comparison, we aim to evaluate the applicability of *real-time* plane detection on consumer off-the-shelf hardware such as the *Intel RealSense* cameras. The answer to this question will consequently determine which algorithm is most suitable. This work focuses on plane detection in complete environments, e.g. an entire room instead of just parts thereof. Furthermore, we restrict ourselves to plane detection in indoor environments as Augmented Reality is usually performed indoors.

[2]https://developer.apple.com/augmented-reality/arkit/
[3]https://developers.google.com/ar
[4]Supported devices of ARCore: https://developers.google.com/ar/devices#google_play_devices
[5]https://github.com/IntelRealSense/librealsense

1.2 Structure

The following chapter presents the basics or background information necessary for this work. In Chapter 3, our concept of achieving the goals mentioned above is detailed. Therein, we prepare the evaluation by selecting suitable algorithms and datasets. A definition of *real-time* closes the chapter. Chapter 4 specifies the implementation details for the concept in the previous chapter. We outline the general system setup, the necessary steps included in the implementation of each algorithm, and the dataset modifications needed to conduct quantitative experiments. The uniform comparison of the selected algorithms is conducted in Chapter 5. Moreover, the results thereof are presented and analyzed. Based on the obtained results, we conclude in Chapter 6. Lastly, the limitations thereof are considered, and this work closes with the prospects of future research.

Chapter 2

Background

In this chapter, we present relevant literature needed to completely understand the proposed concept of chapter 3. The structure is inspired by the system overview pictured in Figure 3.1: First, we explain different formats of data, separated by input and output. Then, we provide details about the used sensors, followed by a general introduction to SLAM algorithms, and a subsequent outline of the SLAM algorithm used in this work (see Subsection 2.3). Thereafter, we detail the plane detection algorithms that resulted from our review of the current literature after introducing three core functionalities of plane detection. Lastly, we outline the list of datasets we took into consideration, and finish with an explanation of metrics used in the evaluation of Chapter 5.

2.1 Data Formats

In the following chapters, we often refer to different data representations. This section provides detailed information about these data representations while differentiating between the input and output of a plane detection algorithm. In this work, note that an algorithm's input is equivalent to the output of the sensors or the SLAM algorithm.

2.1.1 Common Input Types

As described in Chapter 1, specialized sensors are needed to record the environment. Through recording, the environment is represented through different kinds of data. Usually, this data representation of the recorded environment falls into one of the following categories:

- unorganized or unstructured point cloud (UPC)
- organized or structured point cloud (OPC)
- Depth-Image (DI)
- RGB-Image (RGBI)

The input types are compared in Table 2.1. Both the organized and the unorganized point cloud store 3D coordinates. The fundamental difference between UPC and OPC is their format. For instance, in the popular *point-cloud-library* data format[1], each point cloud has a width and a height. The width of a UPC is the number of included points, while the height equals 1. In contrast, the width and height of an OPC depend on the resolution of the recording camera. For instance, if a camera has a resolution of 640x480, the resulting organized point cloud would have a width and height of 640 and 480, respectively. Another distinguishing factor is that the organized point cloud can only represent the environment from one specific viewport at a time, which can lead to point occlusion, e.g., the wall behind a monitor. Moreover, an OPC can include missing or invalid data due to the limitations of the sensor. In contrast, UPCs are neither limited to a specific viewport nor do they allow for missing points.

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.

Table 2.1: Possible inputs for plane detection algorithms columns dedicated to the stored data, the memory layout, and whether the input type allows for invalid values.

Input Type	**Value Types**	**Memory Layout**	**0 or NaN**
OPC	3D Coordinates	2D Matrix	N
UPC	3D Coordinates	1D Array	Y
DI	Distances to Sensor	2D Matrix	Y
RGBI	RGB	2D Matrix	Y

It is worth noting that, in the literature, the terms "depth image", "depth map" and "organized point cloud" are used interchangeably.

[1] http://pointclouds.org/documentation/tutorials/pcd_file_format

Furthermore, some methods are specifically designed to accept *Light Detection And Ranging* (LiDAR) point clouds. Since the underlying data is, essentially, an unorganized point cloud, we do not differentiate between these two types of data.

2.1.2 Common Output Formats

The format of the detected planes has to conform to the specific application. For instance, an architectural remodeling application would require the output to be able to contain non-rectangular shapes and holes. An algorithm that returns a set of planes represented by their convex hull would not be appropriate because it would fail to correctly represent planes like the top of a corner desk or a wall with a doorway.

Being related to the input format, the output of plane detection algorithms can be clustered as follows:

- The cartesian plane equation parameterized by its normal vector n and/or its distance to the origin d, *and*
- 2D inliers (2D-IN), *or*
- 3D inliers (3D-IN).

A common output format is a plane parameterized by its normal vector n and its distance to the origin d. This representation describes a plane as infinitely dense and unbound, if no further information is provided, e.g., extents in certain directions.

We differentiate between the plane inliers according to the way, individual values are accessed. 3D inliers are a set of 3D coordinates that represent a plane in an unorganized point cloud. Like the point cloud itself, the inliers are accessed through their 1D index within the set. In contrast, 2D inliers are a set of indices that correspond to values in an organized point cloud, depth image, or RGB image. Moreover, some methods return a set of segmentation masks, which we also refer to as 2D inliers. Note that we include organized point clouds in the 2D case of inliers even though the values inside are three-dimensional, as the access thereof is still grid-like.

2.2 Intel Realsense

In this work, we use both 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 main motivation for using RGB-D cameras is depth perception. The primary differences and similarities between the T265 and the D455 are reported in Table 2.2. With two fisheye lenses and two imaging sensors,

the T265 and D455, respectively, are considered stereo cameras. Moreover, the D455 has additional RGB and infrared (IR) sensors. The combination of IR and two imaging sensors is used to produce depth images. With a diagonal field of view (D-FOV) of 163°, the T265 has a much wider D-FOV than the D455, which only has a D-FOV of 111°. The maximum frames per second (FPS) of 90 of the D455 are determined by the individual FPS of both its imaging sensors, and its IR sensor. It is worth noting that the FPS decreases with increasing sensor resolution and that 90 FPS is only reached with a resolution of 640x480. In contrast, the T265 has a maximum FPS of 30. Furthermore, both cameras have an integrated Inertial Measurement Unit (IMU) which is used to gather information like monentum, orientation, and acceleration.

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 Robot Operating System[2] (ROS) adaptation is also provided in Intel's Github repository[3].

Table 2.2: Intel RealSense T265 and D455 camera specifications. The resolution and FPS columns report the respective maximum values. More information and the complete Datasheets can be found at https://www.intelrealsense.com/.

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

2.3 Simultaneous Localization and Mapping

Simultaneous Localization And Mapping (SLAM) 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. It 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 simultaneously attempts to accurately estimate its position based on the updated map (*Localization*). The new information about its position is processed during the next map update. This functionality of building a comprehensive map of the recorded environment is also employed by many modern AR systems to gain spatial awareness.

Over decades of research, varieties of different (combinations of) sensors have been employed to solve this problem more accurately and efficiently. Internal odometry sensors

[2]https://www.ros.org/
[3]https://github.com/IntelRealSense/realsense-ros

alone can be unreliable if the robot moves over uneven or slippery surfaces. Visual sensors are, therefore, integrated into many SLAM methods, making them a Visual-SLAM (VSLAM) method. Therein, the methods can be further divided into *visual-only, visual-intertial*, and *RGB-D* SLAM [38].

Visual-only SLAM algorithms are solely based on the input of a camera. Popular VSLAM algorithms include *ORB-SLAM2* [41], $MonoSLAM$ [13], and DSO-SLAM [16].

Visual-intertial SLAM methods integrate additional sensor information from an *Inertial Measurement Unit* (IMU). The IMU provides information about the rotation, acceleration, and magnetic field around the device. These types of information generally aid the estimations based on visual input. However, the IMU data needs additional estimations, which increases the SLAM's complexity. *ORBSLAM3* [8] and *Robust Visual Inertial Odometry* (ROVIO) [6] are *visual-intertial*-based SLAM methods.

Lastly, RGB-D SLAM methods combine a monocular RGB camera and a depth sensor. This combination removes additional calculations initially needed to obtain depth information. *Dense Visual Odometry* (DVO) [29] and $RGBDSLAMv2$ [15] are instances of popular RGB-D SLAM methods.

Real-Time Appearance-Based Mapping

Real-Time Appearance-Based Mapping (RTAB-MAP) is a VSLAM system with an optional IMU input, making it a *visual-intertial* SLAM. RTAB-MAP's general workflow is shown in Figure 2.1. All these inputs are combined during a synchronization step and 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.

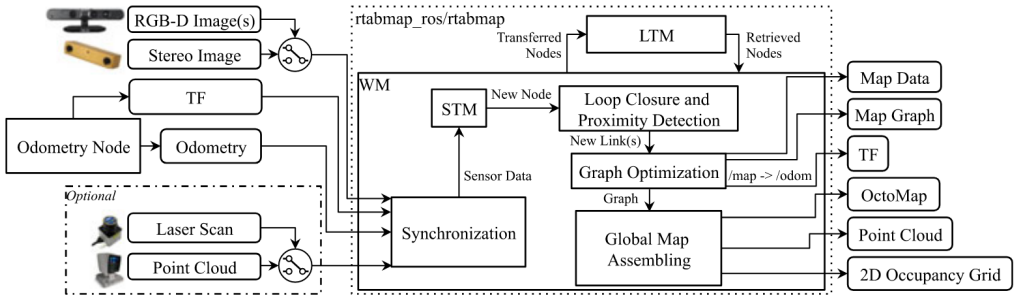


Figure 2.1: Block diagram of RTAB-MAP's system architecture. Taken from [32, Fig. 1]

2.4 Core Algorithms

The field of plane detection has been around for decades. Most methods of detecting planar regions are based on one of three main categories [33, 2]:

- Hough Transform,
- RANSAC, *and*
- Region Growing.

These categories of algorithms are fundamentally different. Hence, it is helpful to explain the difference. In the following subsections, we detail each one by providing general information and a subsequent explanation in the context of plane detection.

2.4.1 Hough Transform

The original motivation behind the Hough transform was detecting lines in images [20]. 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[14].

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 tuple is also called a *Hough Space* in this context.

Votes are then cast on the corresponding value of θ, depending on the number of inliers within a specific *Hough Space* (r, θ). 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. The performence of different shapes thereof had been discussed by Borrmann et al. [7].

2.4.2 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[58, 65, 4]. 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[65]. The model, in that case, could be a cartesian plane equation.

2.4.3 Region Growing

Region Growing methods are often used in the field of image or point cloud segmentation [46, 60]. Region growing-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 [39]. 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, e.g., 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) [2] is based on region growing. After taking an unorganized point cloud as input, the procedure is divided into three phases: *Split, Grow and Merge* (see Figure 2.2).

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 a threshold ε corresponding to the minimum number of points per leaf. The authors propose a value of $\varepsilon = 0.1\%$ of the entire point cloud. Alternatively, the subdivision terminates if a node reaches the maximum depth level $l_O = 10$. Note that this parameter is not referenced in [2]. However, it is used in the official implementation[4]. The subdivision is followed by a planarity test [2, Section 3.2], during which the octree is traversed bottom-up. It is comprised of three individual tests:

1. A distance test,
2. a normal divergence test, *and*
3. an inlier to outlier ratio test.

These tests are influenced by corresponding parameters: The distance test is passed if the point-to-plane distance is smaller than the *Maximum Distance to Plane* (MDP). If the normals of a patch deviate less than the *Maximum Normal Deviation* (MND), the second test is passed. Lastly, a patch is discarded if its percentage of outliers exceeds the *Maximum Outlier Ratio* (MOR). If all eight children of a node n are leaf nodes that fail the planarity test, n replaces its children and becomes a leaf node itself. This procedure is repeated exhaustively or until the root of the octree is reached.

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

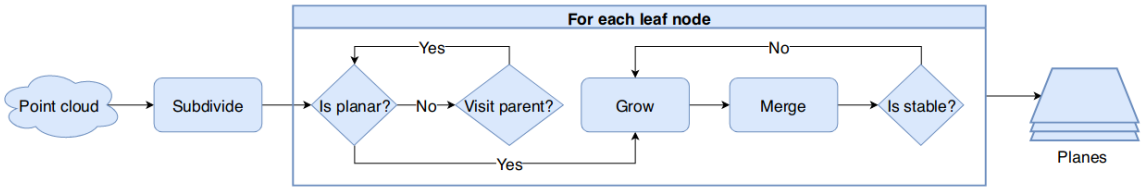


Figure 2.2: The RSPD plane detection pipeline. Taken from [2, Figure 2].

Grow In preparation for the growth phase, a neighborhood graph over the entire point cloud is created. Every node within the graph represents one point, and an edge between two nodes exists only if a nearest-neighbor search with a neighborhood size k detects both points being in the same neighborhood. The authors use a neighborhood size of $k = 50$.

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).

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

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

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,
- their normal vectors diverge less than a tolerance *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) [58] accepts an unorganized point cloud as input.

First, a sample of points is uniformly selected from the entire point cloud. Sample sizes of $\alpha_s \in [0.3\%, 3\%]$ are used in the Evaluation in [58, Table 3]. The normal vectors of these points are estimated using *singular value decomposition* (SVD) and the k-nearest-neighbors (KNN), which are obtained using a k-d tree. An inverse distance weight function is employed to prioritize neighboring points that are closer to the sample of which the normal vector is currently being estimated.

After normal estimation, one-point-RANSAC is performed. Usual RANSAC implementations sample three points to fit a plane [65, 5]. However, OPS fits a plane with only one sample point and its normal vector and counts the number of points where the distance to the plane is smaller than a *minimum-distance parameter* θ_h. Moreover, this number of inliers must exceed a *minimum plane size* threshold θ_N, for a plane to be accepted. 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. Furthermore, the number of needed iterations I is adaptively determined in each iteration, see Equation 2.3.

$$I = \frac{\log(1-p)}{\log(1-(1-e))}, \tag{2.3}$$

where e is the ratio of outliers left in the point cloud, and p is a tuneable parameter that corresponds to the likelihood that a random sample includes no outliers. This process is repeated until the number of remaining points falls below a predefined threshold θ_N. After termination, smaller detected planes are merged if they pass a coplanarity test. After a successful merging of planes, the normal of the resulting plane is calculated via singular value decomposition.

2.5.3 3D Kernel-based Hough Transform

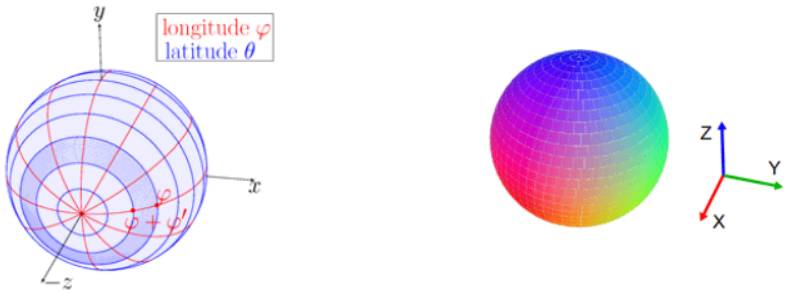
With the *3D Kernel-based Hough Transform* (3D-KHT), Limberger and Oliveira [33] propose a Hough transform-based plane detection method,that accepts unorganized point clouds as input.

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. The octree level, at which the algorithm starts to check for approximate coplanarity under nodes is adjusted through the corresponding parameter s_{level}.

Approximate coplanarity of a point cluster is evaluated based on its eigenvalues and two parameters s_α, s_β. Therein, s_α corresponds to the tolerance regarding noise, and s_β corresponds to the tolerance regarding the anisotropy of the included points. The authors report good results with $s_\alpha = 25$ and $s_\beta = 6$.

Each leaf inside the octree continues subdividing until the points inside a leaf node are considered approximately coplanar, or the number of points is smaller than a minimum-points threshold s_{ps}. The authors recommend $s_{ps} = 30$ for large point clouds. However, no definition of large in this context is given. After the approximately coplanar nodes are refined by removing outliers, a plane is fit through the remaining points.

This plane can, in polar coordinates, be uniquely described by a triple (ρ, θ, ϕ). Inspired by Borrmann et al. [7], an accumulator ball (Fig. 2.3b) 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 2.3a. Furthermore, the discretization of the accumulator is determined by tuneable parameters, namely ϕ_{num} and ρ_{num}. No additional parameter is employed for the discretization of ρ, as this is allocated as needed during the voting procedure. The authors use $\phi_{num} = 30$ and $\rho_{num} = 300$ during their evaluation.



(a) (b)

Figure 2.3: (a) Accumulator array taken from [7, Figure 3]. (b) Accumulator ball used in 3D-KHT, taken from [33, Figure 5].

During the voting procedure, votes are cast on previously calculated approximately coplanar clusters. When casting a vote on a given cluster c 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. 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. Furthermore, an intermediary smoothing step is performed by merging adjacent peaks inside the accumulator. 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) [60] employs region growing to detect planes in UPC.

First, an octree is used to recursively subdivie an unorganized point cloud. 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 root-mean-square (RMS) of the distance of all included points to the plane.

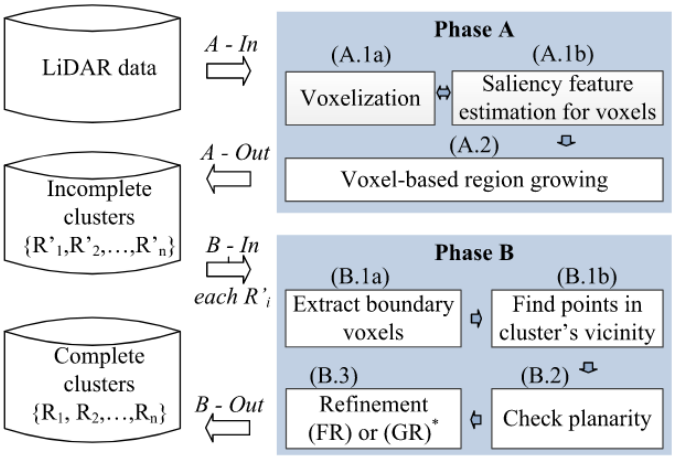


Figure 2.4: The OBRG plane detection pipeline. Taken from [60, Figure 1].

For the region growing phase, all leaf nodes are selected as individual seed points. Starting from the seed with the lowest residual value, 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 θ_{ang}. Values between 3.5 degree and 15 degree were used during the evaluation [60, Tables 1, 4, 7]. Furthermore,

if a leaf node is inserted into a region, it is also considered a starting point for a future iteration, if its residual value is lower than the corresponding threshold θ_{res}. Small values at a maximum of 0.05(m) are reported. If a region cannot be expanded, it is marked as detected, if its number of inliers exceeds a threshold θ_M.

Lastly, a refinement step is employed. For efficiency reasons, it is only performed on voxels at the edge of segments. Fast refinement (FR) is performed on regions that succeed in a planarity test, i.e., 70%-90% (θ_p) of included points fit the best plane [60, Section 3.4]. FR is leaf-based, and all previously unallocated neighboring nodes are added to the region if the point-to-plane distance is smaller than a distance threshold θ_d. 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) [17] is a plane detection algorithm that takes an organized point cloud as input. The agglomerative hierarchical clustering is based on the *line regression* algorithm[42, Section III.B]. The primary difference is that, instead of a double linked list, PEAC operates on a graph G.

First, the input organized point cloud is divided into non-overlapping nodes through the initialization of G. The nodes have a pre-determined height and width and thereby cluster a set of points. Then, G is refined by removing the following types of nodes and corresponding edges [17, Section III.A]:

1. Nodes that contain NaN or 0 values, e.g., missing data,
2. nodes that contain at least one point that is depth-discontinuous with its four neighbors,
3. non-planar nodes, *and*
4. nodes that share an edge with two different planes, e.g., the corners of walls.

During this refinement step, all points inside a node share a common plane normal.

The *agglomerative hierarchical clustering* step starts with the construction of a data structure that contains the nodes of G, sorted in ascending order by their MSE. The following steps are then repeated exhaustively. First, the node v that currently has the smallest MSE is merged with one of its neighbors u that minimizes the merged MSE MSE_{merge}. If the MSE_{merge} exceeds a threshold θ_{MSE}, then a plane is found and the

merged node is removed from G. Otherwise, the merged node is added back to G, joining the edges of v and u.

Lastly, the detected planes are refined. Due to the clustering of nodes that contain a set of points, certain types of artifacts can occur [17, Section IV]:

- Sawtooth,
- unused data points, *and*
- over-segmentation.

First, if not all neighbors of a node v belong to the plane of v, all points of v are added to a queue Q_{RG}. Then, region growing is performed on the points inside Q_{RG}. Therein, the 4-connected neighbors of the current seed are observed. A neighboring point n of a region is moved to other regions, if the distance thereto is shorter than the distance to its current region. Additionally, n is inserted into Q_{RG}. The region growing is followed by another *agglomerative hierarchical clustering* step.

2.5.6 Fast Cylinder and Plane Extraction

Fast Cylinder and Plane Extraction (CAPE) [46] is a region growing-based algorithm that detects planes and cylinders in organized point clouds. The authors propose this algorithm as an extension of their Probabilistic Visual Odometry Framework [45].

In preparation for the region growing step, the cloud is subdivided into patches of predefined size P, similar to the graph initialization of PEAC (see Section 2.5.5). A P value of 20x20 was used in their evaluation [46, Section V.A]. Then, the planarity of the patches is tested, wherein a patch is considered non-planar, if:

- The amount of NaN or 0 points exceeds a threshold, *or*
- the patch has depth discontinuities.

Therein, only the pixels on an axis-aligned cross through the center of the patch are checked for depth discontinuities. A plane is then fit through the patch inliers by performing principle component analysis. The plane's MSE corresponds to the lowest eigenvalue, and the plane's normal vector is its eigenvector. Patches are considered planar if the mean depth deviates less than a threshold θ_{depth} from the standart deviation of the points inside the patch.

The normal vectors are converted from cartesian to spherical coordinates, thereby being described by a polar angle, and an azimuth angle. A histogram H is built by assorting these normals into bins. This histogram is subsequently used for the region growing step. First, a set of patches C of the most frequent bin in H are obtained. The region growing terminates if $|C|$ is smaller than a threshold k_1. The patch with the smallest MSE out of

C is chosen as a seed s. In general, a 4-connected neighborhood is used in this approach, and a neighboring patch n is inserted into a region, if:

1. n is not assigned to any region,
2. the dot product of both patches normal vectors is less than a threshold T_N, *and*
3. the orthogonal distance of n's center to the region is less than $T_d(s) = l\sqrt{(1 - T_N^2)}$,

where l is the distance between the corner points of s. If no further growth is possible, a complete segment S is found. All included patches in S are removed from the list of remaining patches, and the bins of the histogram are updated accordingly. Lastly, if S exceeds the minimum plane size k_2, it is considered a plane candidate.

By comparing the ratio of the second largest eigenvalue to the smallest eigenvalue of the included points, the planarity of a region is evaluated. If this ratio exceeds a predetermined parameter θ_{plane}, the segment is considered a plane. Otherwise, additional steps are needed.

First, the surface of a segment is checked for invariance, which is done by a principle components analysis on the normal vectors. Therein, if their condition number exceeds a threshold θ_{cond}, the segment is comprised of a set of cylinders or planes [46, Section III.D].

The center points and normal vectors of all patches in the segment are then projected onto a plane. Then, plane and cylinder models are fit by performing RANSAC. If the MSE of the plane model is lower than the MSE of the corresponding cylinder model, the model is considered a plane, and a cylinder otherwise.

Lastly, the segments are undergone a refinement step. First, segments are eroded by removing boundary patches through the use of a 3x3 kernel. Next, the eroded segments are expanded by using a 3x3 8-neighbor kernel. The authors propose, that all patches valid for refinement are given by the difference between the expanded segment and the eroded segment. Lastly, each pixel p within the patches is added to the segment S if:

$$dist(p, S) \leq MSE_S \cdot k, \tag{2.4}$$

where k is a constant. The authors use $k = 9$ in their work.

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 through region

growing. The primary concept behind this method is that the planes are not parameterized, but rather represented as a set of geometric constraints in the spherical coordinate system.

First, a set of pre-processing steps is employed. The sensor noise is reduced through bilateral filtering. Subsequently, a normal map of the OPC is generated. To prepare for the region growing step, appropriate seeds are selected. Similar to PEAC and CAPE, the 2D point cloud is subdivided into equal-sized patches, and the patch centroids are choosen as seeds. A patch is considered planar if its MSE is smaller than a threshold.

Having obtained a set of seed points, the region growing step is employed. A queue is used for the retrieval of seeds and neighboring points. Each time a new seed is retrieved, all normals of the corresponding patch are gathered and transformed into the spherical domain. Starting from a seed s, neighboring points and their normals are sequentially retrieved. If a neighbor n is already assigned to a region or the depth difference to s exceeds a threshold T, it is discarded. If n is located inside of the spherical convex hull of the region of s, it is added to the region, and its neighbors are inserted into the queue. If n is not inside a convex hull, it is necessary to check whether n is outside of the so-called *cluster-permissible region* (CPR):

$$\mathrm{CPR}(p) = \{q \in \mathbb{S} | \angle(p, q) \leq 2\theta\}, \tag{2.5}$$

where p is a normal on the sphere, and θ is an angular threshold. Note that the CPR of multiple points, e.g., a region, is determined by the intersection of all individual points. If n is outside the CPR of the current region, it is discarded. Otherwise, the convex hull is updated with n, n is added into the set of points included in the region, and its neighbors are inserted into the queue. It is worth noting, that seed points that are associated with already detected planes are discarded upon retrieval.

Lastly, dilation is performed to eliminate holes in planes [40, Section III.B].

2.5.8 Depth Kernel-based Hough Transform

Depth Kernel-based Hough Transform (D-KHT) [59] is a Hough transform-based plane detection algorithm that accepts depth images as input.

The algorithm is performed in three stages: *Clustering, Voting,* and *Peak Detection.*

Clustering In the clustering step, a quadtree is constructed to spatially subdivide the image. A node recursively subdivides itself, until a minimum of included points s_{ms} is reached, or until the set of points is considered approximately coplanar. The latter is determined through a PCA. Pre-computing and subsequently referencing *Summed-Area-Tables* (SATs) is done to increase the efficiency of this step, leading to a constant-time calculation of the covariance matrix. If a non-planar node has less than s_{ms} points, these points are then considered non-relevant and ignored during the voting step.

Voting The clustering step returns a set of approximately coplanar point clusters. The best-fitting plane of each cluster is calculated. It is determined by the cluster's mean point, and a normal vector which is the eigenvector associated with the least eigenvalue of the covariance matrix. An accumulator to store the votes and a set of gaussian kernels corresponding to the coplanar clusters are necessary to perform the voting step. Given the mean point $\mu_{(x,y,z)}$ and the unit normal vector $\overrightarrow{n} = (n_x, n_y, n_z)^T$ of a cluster, the center of the gaussian kernel is calculated:

$$\mu_{(\rho,\phi,\theta)} = \begin{pmatrix} \mu_x \\ \mu_y \\ \mu_z \end{pmatrix} = \begin{pmatrix} n_x\mu_x + n_y\mu_y + n_z\mu_z \\ \cos^{-1}(n_z) \\ \tan^{-1}(\frac{n_y}{n_x}) \end{pmatrix} \tag{2.6}$$

The covariance matrix of the gaussian kernel can then be calculated through first-order error propagation:

$$\Sigma_{(\rho,\phi,\theta)} = J\Sigma_{(x,y,z)}J^T, \tag{2.7}$$

where J is the Jacobi-Matrix of Equation 2.6. Similar to [7], an unbiased spherical accumulator is used to store votes. The axes of the accumulator range are in the following ranges:

$$\theta \in [-\pi, +\pi], \tag{2.8}$$

$$\phi \in [0, \pi], \ and \tag{2.9}$$

$$\rho \in [0, \rho_{high}]. \tag{2.10}$$

Therein, ρ_{high} is the determined by the farthest distance between the camera and a point of the input image. The discretization of θ and ϕ are pre-determined by the user and should be based on the expected granularity of detected planes [59, Secion 3.3]. This likely relates to the expected amount of noise as well. During the voting itself, the accumulator bins are incremented if they are within two standard deviations of the kernel mean $\mu_{(\rho,\phi,\theta)}$.

Peak Detection The voting step fills the bins of the accumulator with votes. A smoothing of the bins is performed to avoid oversegmentation or the detection of multiple equivalent planes. Vera et al. [59, Section 3.5] compute a convolution of the accumulator, as well as a 6-connected filter with a central weight of 0.2002 and neighbor weights of 0.1333. Each peak represents the parameterization of a detected plane. These planes are then sorted by relevance, whereas the relevance of a plane is determined by a weighted sum of the clusters that voted for the corresponding peak. The cluster weights correspond to the number of pixels, rather than the number of samples.

All planes, parameterized by (ρ, ϕ, θ), are then transformed back into cartesian coordinates. Moreover, they are parameterized by a center point, which is defined by ρ, and a normal vector $\overrightarrow{n}$:

$$\overrightarrow{n} = \begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix} = \begin{pmatrix} \sin\phi\cos\theta \\ \sin\phi\sin\theta \\ \cos\phi \end{pmatrix} \tag{2.11}$$

2.5.9 Depth Dependent Flood Fill

Depth Dependent Flood Fill (DDFF) [49] is a region growing-based algorithm that detects planes in organized point clouds. The algorithm builds upon their previous method [50].

The algorithm starts by selecting appropriate seeds. The organized point cloud gets subdivided into cubes of a pre-determined size σ_{seed}. The points inside are then checked for depth and curvature discontinuities. If neither case holds, the center of the cube is marked as a seed for region growing.

The region growing is performed on point-level. Starting from a given seed s, a region is expanded by checking neighbors within the same horizontal span. Given a horizontal span of length n and its middle pixel p, the minimum plane size σ is calculated:

$$\sigma = \rho(p) = \kappa_\rho \cdot \delta(p)^2 + \gamma_\rho, \tag{2.12}$$

where $\delta(p)$ is the depth value of p, and κ_ρ and γ_ρ are tuneable parameters. A pixel n that is σ pixel away from the left or right border of the span is observed next. n and all pixels toward the border are added to the span of s if n and at least one of its vertical neighbors pass the following membership test:

1. The perpendicular distance of n to the plane defined by s and its normal is smaller than a threshold τ_{flood},

2. the euclidian distance between p and a neighboring pixel is smaller than a threshold τ_{point}, *and*
3. n is neither NaN, nor 0.

In a last refinement step, the set of planes is reduced by merging oversegmented planes. First, a neighborhood graph of the rough segments is constructed. Any segments that share a border are connected through a bidirectional edge. The graph is traversed in a breadth-first manner, merging two connected nodes, a, b, if they satisfy the following conditions:

1. the seed normals of a and b diverge less than a threshold τ_{angle}, *and*
2. the perpendicular distance between the segments is less than threshold τ_{merge}.

If a, and all the planes a merged with fail this membership test with b, the edge (ab) is removed from the graph. Note that the edge (ba) will be validated later, as b is subject to change through merges with other neighbors. After a successful merge, a new centroid and normal are obtained by adding both plane normals and centroids together, while the respective amount of inliers acts as a weight.

The BFS iteration is repeated, until no merge occurs during an iteration. The authors state that, typically, no more than five iterations are needed [49, Section III.E]. If this fails, the unidirectional edge (ab) is removed from the graph.

The flooding under the constraint of minimum plane size leaves gaps. Therefore, as a refinement step, a two-pass algorithm is used to fill them. In both passes, horizontal strips of non-assigned pixels are detected. If both vertical neighbors of an unmarked pixel share a label, the unmarked pixel is marked with this label.

2.5.10 PlaneNet

PlaneNet [36] is a deep learning-based approach to piece-wise planar reconstruction of a scene from a single RBB-Image.

The method implements a *Dilated Residual Networks* (DRNs) [68] as a precursor for a total of three output branches. These three branches predict

1. A set of plane parameters,
2. a set of corresponding segmentation masks, *and*
3. a non-planar depthmap,

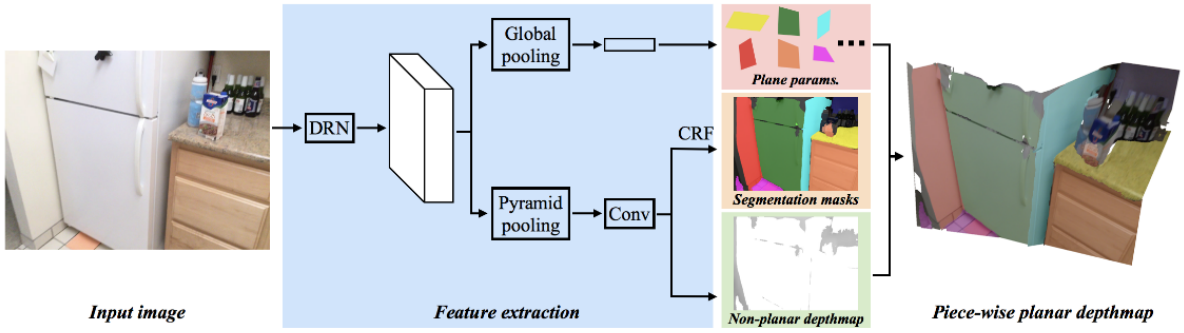


Figure 2.5: The PlaneNet Architecture. Taken from the respective paper [36, Figure 2].

as portrayed in Figure 2.5. The DRN is followed by a global pooling, or a pyramid pooling step, depending on the prediction branch. To obtain a set of plane parameters, the global pooling is followed by a fully connected layer, which produces K parameter triples. K corresponds to the number of expected planes within a scene. Liu et al. [36] use $K = 10$ during their experiments. A convolution layer is placed after the pyramid pooling the output of which is returned by the *non-planar depthmap* branch. An additional dense conditional random field (DCRF) is applied to obtain the segmentation masks.

2.5.11 PlaneRecNet

PlaneRecNet [63] is a deep learning-based approach for piece-wise plane detection and reconstruction that takes RGB images as input.

The entire architecture of PlaneRecNet is portrayed in Figure 2.6. The general approach of this method is to calculate plane parameters through PCA or RANSAC after providing a precisely estimated depth. This is achieved by implementing two separate branches, namely *Plane segmentation*, and *Depth estimation*. Both branches obtain their input from a common precursor which is a *ResNet* [22] backbone. The *Plane segmentation* branch is a lightweight configuration of SOLOV2 [61] that has been modified to fit the context of plane detection. The *Depth estimation* branch is a lightweight *Feature Pyramid Network* (FPN) [34]. Within the *Depth estimation* branch, a so-called *Plane Prior Attention* (PPA) module is used to introduce the mask candidates from the *Plane segmentation* branch into the depth estimation. The PPA module is based on the *Depth Attention Volume* [26].

The estimated depth values and the segmentation masks, and classes, are then combined into a piecewise planar scene reconstruction of the input image. No further refinement is implemented.

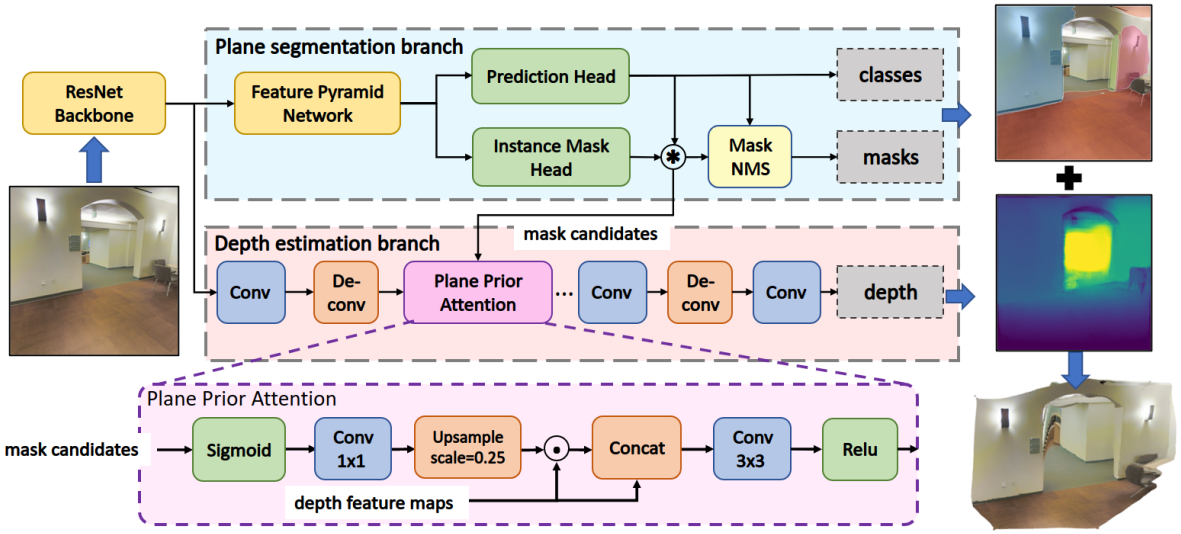


Figure 2.6: The PlaneRecNet Architecture. Taken from the respective paper [63, Figure 1]

.

2.5.12 PlaneRCNN

PlaneRCNN [35] is a deep neural architecture that detects planar regions and reconstructs a piecewise planar depth map from RGB-Images. The architecture of PlaneRCNN, as seen in Figure 2.7, consists of three primary modules:

1. The plane detection network,
2. a segmentation refinement network, *and*
3. a warping loss module.

The first module is built upon the semantic segmentation network MaskRCNN [21]. The module is modified to differentiate between "planar" and "non-planar" object instances. In contrast to *PlaneNet* (see Subsection 2.5.10), the number of planes to detect is not pre-determined. Moreover, plane parameters and segmentation masks are calculated.

The second module, namely the *segmentation refinement network*, refines the extracted segmentation masks obtained from the *plane detection network*. This is done by introducing non-locality into a U-Net [48] by combining a convolution layer with an accumulation of feature volumes. The authors name this particular step the *ConvAccu* module [35, Section 3.2].

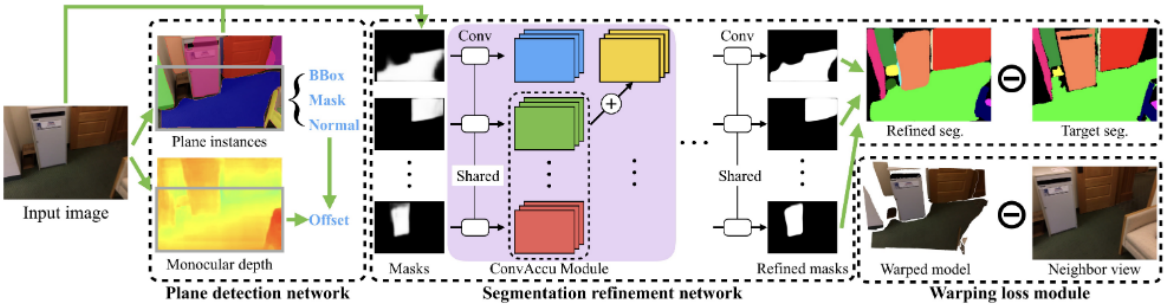


Figure 2.7: The PlaneRCNN Architecture. Taken from the respective paper [35, Figure 2].

The *warping loss module* implements a refinement step. This module uses a second angle of the same view to ensure consistency. A view from 20 frames ahead is projected into the current view to further refine the segmentation process. It is worth noting, that this is only performed during training to improve the accuracy.

2.6 Datasets

This section outlines a set of datasets popular in the evaluation of algorithms in the field of plane detection. Table 3.3 summarizes the key characteristics of each dataset. The individual datasets are outlined in the following subsections.

2.6.1 2D-3D-S

2D-3D-S [3] was recorded in three different buildings of the Stanford University and divided into six distinct areas, including 272 different scenes. During the recording, a 360° *Matterport*[5] camera was used. A detailed statistic of the included scene types can be found in Table 4.1. The scenes include a complete unstructured point cloud and a list of annotated files representings emantically different objects that can be found therein. Within the dataset, the point cloud sizes, i.e., number of points, range from $\sim 8 \cdot 10^4$ to $\sim 9 \cdot 10^6$, with an average of $\sim 1 \cdot 10^6$.

[5] https://matterport.com/de/cameras/360-cameras

2.6.2 Leica

The Leica[6] dataset is a collection of 23 scans of outdoor environments. Each scan consists of a dense unorganized point cloud which has been recorded by the *Leica BLK360* LiDAR Scanner[7]. The dataset is saved in *ASTM E57* [25], which is a general-purpose file format usually used for the storage of 3D data, e.g., point clouds, laser scans or images.

2.6.3 Kinect

The Kinect [43] dataset is a set of 30 organized point clouds. The recording was performed with a Microsoft Kinect camera with a 640x480 resolution inside . A ground truth that focuses on plane and cylinder segmentation is provided in addition to the point clouds.

2.6.4 SYNPEB

The SYNPEB dataset was introduced by Schaefer et al. [52] to improve upon the popular *SegComp* dataset. The synthetic dataset includes a 6x7x3m room with various geometric objects inside. Moreover, a total of 40 scans from different views within the room are provided as organized point clouds with a resolution of 500x500. Lastly, the provided ground truth represents a planar segmentation of the scene.

2.6.5 ARCO

The ARCO [23] dataset consists of 6 organized point clouds that have been recorded with a Microsoft Kinect camera. All scenes show real indoor spaces, including a living room, a kitchen, a saloon, a hallway, furniture, and an ordinary room. They are saved in the popular *.pcd*[8] file format. We are not aware of the existence of a ground truth.

[6] https://shop.leica-geosystems.com/de/leica-blk/blk360/dataset-downloads
[7] https://shop.leica-geosystems.com/de/leica-blk/blk360
[8] http://pointclouds.org/documentation/tutorials/pcd_file_format

2.6.6 SegComp

The SegComp [24] dataset includes a collection of over 400 depth images, as well as a corresponding ground truth. Additionally, a complete evaluation package is provided on the corresponding website[9]. The images are saved in their own file format, namely *.gt-seq*.

2.6.7 NYU V2

The NYU-V2 [54] dataset is an improvement of the preceding version, namely *NYU-V1*[10]. The dataset consists of 1449 RGB-D images, 464 scenes from 3 US cities, and more than $400,000$ unlabeled frames. A ground truth in form of labeled objects is provided, as well as a toolbox for arbitrary manipulation of data. Everything was recorded with the Microsoft Kinect camera.

2.6.8 ICL-NUIM

The ICL-NUIM [18] dataset was introduced for the benchmarking of RGB-D, VO and SLAM algorithms. Eight scenes are included, which are comprised of a surface ground truth, depth images, and the trajectory of the camera. The dataset includes a total of over 8000 images over these four scenes, whereas four have been recorded two indoor environments, namely a living room, and an office.

2.6.9 SUNRGB-D

The SUNRGB-D [55] dataset is used for 3D object detection. The data is split into a training set and a testing set. Over both sets, a total of over $13,000$ RGB-D images are included, with corresponding ground truths in form of 3D bounding boxes. Furthermore, a toolbox is provided. The dataset differentiates between 19 different objects, including, but not limited to: "wall", "door", "bookshelf", and "table". More information can be found on the related website[11] under "Other Materials".

[9] http://www.eng.usf.edu/cvprg/range/seg-comp/SegComp.html
[10] https://cs.nyu.edu/~silberman/datasets/nyu_depth_v1.html
[11] https://rgbd.cs.princeton.edu/challenge.html

2.6.10 TUM

The TUM [57] dataset was created for the evaluation of VO and Visual-SLAM systems. A Microsoft Kinect sensor with a resolution of 640x480 is used to record a total of 39 scenes in indoor environments. While the dataset is primarily focused on trajectories, organized point clouds are provided as well. A ground truth trajectory is provided for each scene. The dataset can be found on the project website[12].

2.7 Evaluation Metrics

In fields that revolve around the detection or the segmentation of particular objects, metrics that describe the quality of detection or segmentation, respectively, are helpful. Prevalently used metrics include the *Precision, Recall*, and the *F1-Score*. *Precision* generally describes how many of the results are relevant, i.e., the percentage of correctly calculated values(see Eq. 2.13). *Recall* describes the ratio of relevant results to all relevant data, i.e. the likelihood of a result being relevant(see Eq. 2.14). Lastly, the *F1-Score* is the harmonic mean of the former two metrics (see Eq. 2.15).

In the context of this work, we calculate *Precision, Recall* and the *F1-Score* based on voxel sets. 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 PC to reduce complexity and to avoid proximity bias, because of the inverse relationship between distance to sensor and cloud density. This regularization is obtained through voxelization of the point cloud. With this voxel grid, we can now calculate corresponding sets of voxels for each list of points that represent a plane. In the next step, we compare our planes from GT with A to obtain a list of corresponding pairs of ground truth and found planes. A ground truth plane gt_i is marked as detected if any plane from the list of found planes achieves a minimum voxel overlap of 50%. With this list of correspondences, we calculate *Precision, Recall* and the *F1-Score*.

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

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

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

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

With those rules, we can calculate the *Precision, Recall*, and *F1-Score*:

[12] https://vision.in.tum.de/data/datasets/rgbd-dataset/download

$$Precision = \frac{True\ Positive}{True\ Positive\ +\ False\ Positive} \tag{2.13}$$

$$Recall = \frac{True\ Positive}{True\ Positive\ +\ False\ Negative} \tag{2.14}$$

$$F1\text{-}Score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{2.15}$$

Chapter 3

Concept

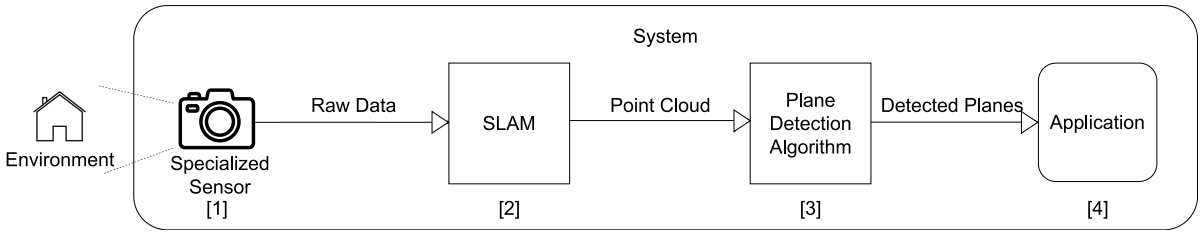


Figure 3.1: A general view on an AR/VR system's architecture. The specialized sensor records data ([1]), which is passed to a SLAM algorithm ([2]). After map assembly, a point cloud is handed to a plane detection algorithm ([3]). The detected planes are given to a use-case-specific application ([4]).

Many AR/VR Systems integrate plane detection into their software[1][2][3]. Figure 3.1 shows a generic block diagram of such a AR/VR system including plane detection. The environment is continuously recorded by a specialized sensor, which is usually a camera, a depth sensor, or a combination thereof ([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 then further processes those planes as required by the underlying use case.

To remove any noticeable delay in the application, the plane detection step has to run within a specific time limit. This temporal constraint is usually referred to as *real-time*.

[1] https://measurekit.com/
[2] https://www.locometric.com/roomscan
[3] https://www.housecraftapp.com/

Davison [12] observe that *real-time* is usually bound to the sensor's frequency. Motivated by this, we introduce a precise definition of *real-time* in Section 3.3.

When creating such an AR/VR system, the choice of plane detection algorithm is naturally of great importance. The problem is that most published algorithms are inherently incomparable. Often different datasets or metrics are used, which precludes a quantitative comparison. Moreover, algorithms are often incomparable by internal functionality due to differences in inputs and the format of the detected planes. Plane detection algorithms are, in some cases, developed to run on specific hardware [66, 40]. We can safely conclude that selecting a single 'best' algorithm, solely based on the results presented in their respective work, is impossible.

Motivated by Chapter 1, we aim to determine the *real-time* applicability in a realistic environment. To achieve this, we perform a uniform comparison of suitable plane detection algorithms. Therein, we especially pay attention to the accuracy and the calculation times. This comparison will thereby yield the algorithm that produces the best results as well. However, to perform this evaluation, we need the following:

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

The following sections are dedicated to these requirements.

3.1 Selection of Plane Detection Algorithms

Since most algorithms differ in certain aspects, it is not possible to perform a comparison that includes every single one. Furthermore, not all algorithms are created from the same motivation and focus on different things. Evaluating an algorithm in a scenario it has not been designed for is likely to yield meaningless results. It is, therefore, necessary to first define objective criteria upon which algorithms are judged to select suitable candidates for the remainder of this work.

3.1.1 Criteria

In the following paragraphs, we 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 inputs 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 2.1. To reiterate, the data representation of the recorded environment falls into one of four categories:

- unorganized or unstructured point cloud (UPC),
- organized or structured point cloud (OPC),
- Depth-Image (DI), *and*
- RGB-Image (RGBI),

whereas 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 point cloud resembles an unsorted 1D array. Moreover, the 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.

Depth images, like OPCs are arranged in a two-dimensional grid, while the included values are the distances to the sensor instead of 3D coordinates. The only difference between RGB and depth images is that the RGB image values are colored pixels.

Detected Plane Format It is essential to determine an appropriate representation of the detected planes. If no uniform output type can be determined, consequently, no uniform metric for comparison can also be found.

In this work, we differentiate between the following:

- 3D-inliers (3D-IN),
- 2D-inliers (2D-IN), *and*
- the plane's normal vector and distance to origin (n, d),

whereas the 3D-inliers of a plane are a set of 3D points in the format of an UPC. In contrast, 2D-inliers are two-dimensional representations of a plane. This includes all methods to describe a plane in a two-dimensional manner, e.g., sets of indices, pixels or segmentation masks. These plane formats generally correspond to two-dimensional input formats like organized point clouds or (depth-) images. Lastly, planes are often described mathematically over their normal vector n and distance to the origin d.

Hardware Requirements Another important aspect to consider is the hardware required by an algorithm. Most plane detection algorithms run on the CPU, some even implement some form of CPU parallelism, e.g., 3D-KHT uses OpenMP[4] to speed up the octree construction [33, Section 4]. However, some methods are implemented either completely or partially on the GPU. For instance, Hidalgo-Paniagua et al. [23] compare different implementations of the RANSAC algorithm, whereas three versions are processed entirely on the CPU, and the last one is offloaded to the GPU via CUDA[5].

To summarize, we differentiate between algorithms that run solely on the CPU and algorithms that, additionally, employ the machine's GPU.

Availability Lastly, to evenly compare a set of algorithms, all need to be implemented on the same machine to exclude the used hardware as a factor. Some authors provide a corresponding implementation to the paper in which they propose their novel plane detection technique. While other publications are limited to the paper, the level of detail regarding the implementation varies.

For further reference, we consider a plane detection algorithm to be *available* if an implementation is generally possible, i.e., the authors provide their implementation or outline the algorithm in a way that enables self-implementation or a corresponding implementation is available online.

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). The table shows the input type and the output format of all algorithms, as well as the required hardware, and the availability. Note that we consider all algorithms to be available. However, we are not aware of public implementations of OBRG and SCH-RG. However, the respective publications outline their methods in high detail, thereby guiding a self-implementation.

The final outputs of PlaneNet, PlaneRecNet and PlaneRCNN are piecewise-planar depth maps of the input image. Since modifying the architecture to return the segmentation masks and plane parameters would require minimal effort, we adjusted the output types in the table accordingly. Similarly, RSPD returns a set of planes parameterized by its normal vector n, distance to origin d, and two additional extents. Modifying the output to return inliers requires minimal effort as well.

[4]https://www.openmp.org/
[5]https://developer.nvidia.com/cuda-toolkit

Table 3.1: A list of Plane Detection Algorithms compiled by reviewing the current literature. The algorithms are clustered by their type of input. The Section column provides the placement within this work. We refer to Subsection 3.1.1 for details regarding the specific namings of different plane formats. In the Hardware column, note that GPU implies the usage of the CPU.

Plane Detection Algorithm	**Section**	**Input Data**	**Plane Format**	**Hardware**	**Available**
RSPD [2]	2.5.1	UPC	3D-IN, (n, d)	CPU	Y
OPS [58]	2.5.2	UPC	3D-IN	CPU	Y
3DKHT [33]	2.5.3	UPC	3D-IN	CPU	Y
OBRG [60]	2.5.4	UPC	3D-IN	CPU	Y
PEAC [17]	2.5.5	OPC	2D-IN	CPU	Y
CAPE [46]	2.5.6	OPC	(n, d)	CPU	Y
SCH-RG [40]	2.5.7	OPC	2D-IN	GPU	Y
D-KHT [59]	2.5.8	DI	2D-IN	CPU	Y
DDFF [49]	2.5.9	DI	2D-IN	CPU	Y
PlaneNet [36]	2.5.10	RGBI	2D-IN, (n, d)	GPU	Y
PlaneRecNet [63]	2.5.11	RGBI	2D-IN, (n)	GPU	Y
PlaneRCNN [35]	2.5.12	RGBI	2D-IN, (n, d)	GPU	Y

As mentioned above, we consider all presented algorithms available even if SCH-RG and OBRG do not seem to have an official implementation. Therefore, while necessary, the criterion of availability does not constrain the selection of algorithms in this case.

Integrating an external GPU into the system poses an additional cost factor. Moreover, the additional weight could have negative effects on the user experience, as AR/VR devices are usually handheld or head-worn. We exclude algorithms that require an external GPU, namely SCH-RG, PlaneNet, PlaneRecNet, and PlaneRCNN.

Addressing the criterion of input type, we are only interested in performing plane detection in complete environments. Each update published by RTAB-MAP is the union of new data and the current state of the recorded map. RTAB-MAP publishes this update in form of an unorganized point cloud (see Figure 2.1). To perform plane detection with an algorithm that expects an OPC as input, the UPC has to be transformed into an OPC. This transformation is not-trivial and involves the projection of 3D coordinates onto a sphere based on a set of sensor parameters. An exemplary implementation thereof is included in the lidar toolbox of MATLAB[6]. However, this transformation neglects the global structure of the environment, as it returns a two-dimensional representation of the environment. Therefore, we focus on unorganized point clouds in this work and exclude PEAC, CAPE, SCH-RG, D-KHT, DDFF, PlaneNet, PlaneRecNet and PlaneRCNN from our evaluation.

[6]https://de.mathworks.com/help/lidar/ug/unorgaized-to-organized-pointcloud-conversion.html

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 (n, d) and a plane represented by its (3D/2D) inliers. The calculated metrics may differ significantly because the plane in cartesian form is infinitely dense. Conversely, the plane described by its inliers allows for holes and non-rectangular shapes, e.g., doorways or a round table, respectively. Being able to represent planes of arbitrary shape is important for many applications. Moreover, only the 3D inliers (3D-IN) conform to the determined input type of UPC (see Section 2.1.2). We thereby determine 3D-IN as the preferred plane format and exclude all methods which do not comply, namely CAPE, PlaneNet, PlaneRecNet, and PlaneRCNN. Note that, hereafter, we refer to 3D-IN solely as inliers.

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

- **RSPD**
- **OPS**
- **3D-KHT**
- **OBRG**

Temporal Subdivision of Phases To enable a precise evaluation, we subdivide these algorithms into a pre-processing phase, a plane detection phase, and a post-processing phase, whereas we use the terms "phase" and "step" interchangeably in this work. In the following, we outline the pre-processing and post-processing steps taken by the selected algorithms. To avoid redundancy, we refer the reader to the Subsections 2.5.1-2.5.4 for more details regarding the plane detection steps of each algorithm.

The pre-and post-processing steps are summarized in Table 3.2. 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 of pre-determined size.

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.

Table 3.2: The Pre-processing and post-processing steps of the plane detection algorithms. "/" denotes the absence of a pre-/post-processing step.

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

3.2 Selection of Datasets

After having obtained a set of suitable algorithms and, according to requirements introduced at the beginning of this chapter (see Section 2), the selection of realistic dataset is necessary. Just like a substantial amount of algorithms are incomparable for different reasons, datasets differ in certain aspects as well. Therefore, this section deals with the selection of an appropriate dataset.

3.2.1 Criteria

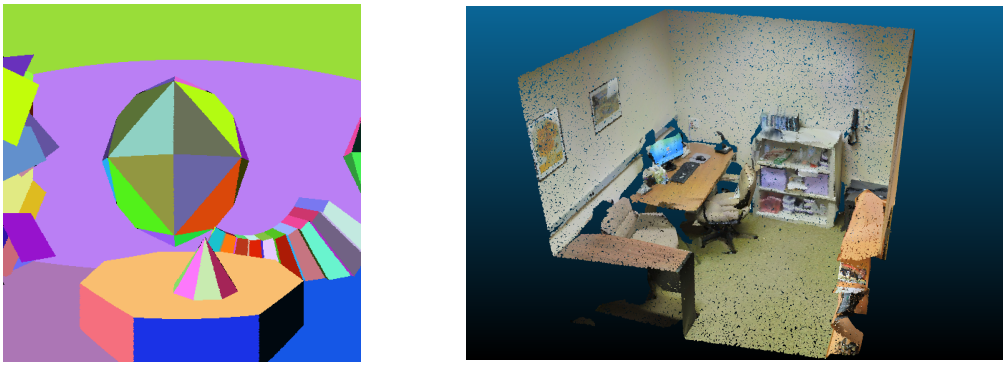
We introduce a set of criteria upon which currently popular datasets are compared. These criteria are widely influenced by the criteria used for the previous selection of algorithms (see Subsection 3.1.1).

Scene Format The Format of a scene in a given dataset corresponds directly to the input of algorithms. To avoid redundancy, we refer the reader to Subsection 2.1.1 and Paragraph 3.1.1 for more information. Thereby, the scene format can be divided into the same four classes:

- unorganized or unstructured point cloud (UPC)
- organized or structured point cloud (OPC)
- Depth-Image (DI)
- RGB-Image (RGBI)

Realism We differentiate between synthetic and real/realistic datasets. Real datasets represent real environments, as they are often recorded manually. In contrast, synthetic datasets often include a collection of geometric bodies and are usually created digitally. Two representatives for both types are shown in Figure 3.2.

Recorded Environment Since realistic datasets are created through manual recording, it is useful to differentiate between indoor and outdoor environments. Note that we consider this criterion only to apply for real datasets as synthetic datasets are not recorded manually.



(a) (b)

Figure 3.2: A synthetic dataset (a) and a real office scene (b). The former is from the *SegComp* [24] dataset, and the latter is from the 2D-3D-S [3] dataset. Note that we cropped the office point cloud for visualization purposes.

Ground Truth Lastly, not all datasets are created out of the same motivation. Therefore, the method of evaluation differs widely over the range of available datasets. In general, the ground truth (GT) of currently popular datasets can fall into one of the following categories (compare Table 3.3):

- Planes,
- classes, *or*
- trajectory.

In some datasets, the ground truth represents a set of detectable planes. The specific Therein, the format of the planes depends on the plane format of the algorithm (see Paragraph 3.1.1).

Often, datasets from the field of semantic segmenation or object detection are used for the evaluation of plane detection algorithms. Therein, objects in a scene are assigned specific object classes, e.g., "Wall", "Ceiling", "Table", or "Cup". The ground truth provides a labeling of these objects. This labeling can take the form of annotated 3D bounding boxes or sets of pixels in the input image.

Lastly, some datasets provide a GT that focuses on the trajectory of the recording sensor. This trajectory is usually taken from integrated sensors like *Inertial Measurement Units* (IMUs).

Naturally, this classification can only apply to datasets that include a ground truth.

3.2.2 Datasets

Table 3.3 summarizes currently prevalent datasets in the scientific literature regarding plane detection. Most of the datasets therein are used in the corresponding papers of the plane detection algorithms presented in Table 3.1. However, this does not influence the selection process. Note that SYNPEB and SegComp are synthetic datasets and, therefore, are neither indoor nor outdoor. Furthermore, we are not aware of a ground truth for the ARCO dataset.

Table 3.3: Plane detection Datasets. The GT column specifies what the ground truth of each dataset represents. The datasets are clustered by their type of format. The second column provides the placement within this work. Note that we include our *FIN* dataset in this table for completeness reasons.

Dataset	Section	Scene Format	Real	Indoor	GT
2D-3D-S [3]	2.6.1	UPC	Y	Y	classes
Leica[7]	2.6.2	UPC	Y	N	planes
Kinect [43]	2.6.3	OPC	Y	Y	planes
SYNPEB [52]	2.6.4	OPC	N	/	planes
ARCO [23]	2.6.5	OPC	Y	Y	/
SegComp [24]	2.6.6	DI	N	/	planes
NYU V2 [54]	2.6.7	DI	Y	Y	classes
ICL-NUIM [19]	2.6.8	DI	Y	Y	trajectory
SUNRGB-D [55]	2.6.9	DI	Y	Y	classes
TUM [57]	2.6.10	DI	Y	Y	trajectory
FIN (ours)	4.4	UPC	Y	Y	planes

In Subsection 3.1.2, we determine unorganized point clouds as the type of input. Furthermore, since we give special focus to the real-world applicability of plane detection algorithms, we must evaluate them on realistic datasets, thereby excluding all datasets except 2D-3D-S and Leica. Additionally, we are especially interested in realistic *indoor* environments, as motivated by Chapter 1. Therefore, Leica ceases to be an option and we subsequently choose 2D-3D-S as the dataset for the evaluation.

Nevertheless, we cannot use the ground truth included in 2D-3D-S because it represents the segmented scene at the level of objects [3, Section 4.1] instead of focusing on planes

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

within the scene. As a consequence thereof, we create a suitable ground truth of the 2D-3D-S dataset through manual segmentation. We provide details of this time-expensive process in Section 4.3.

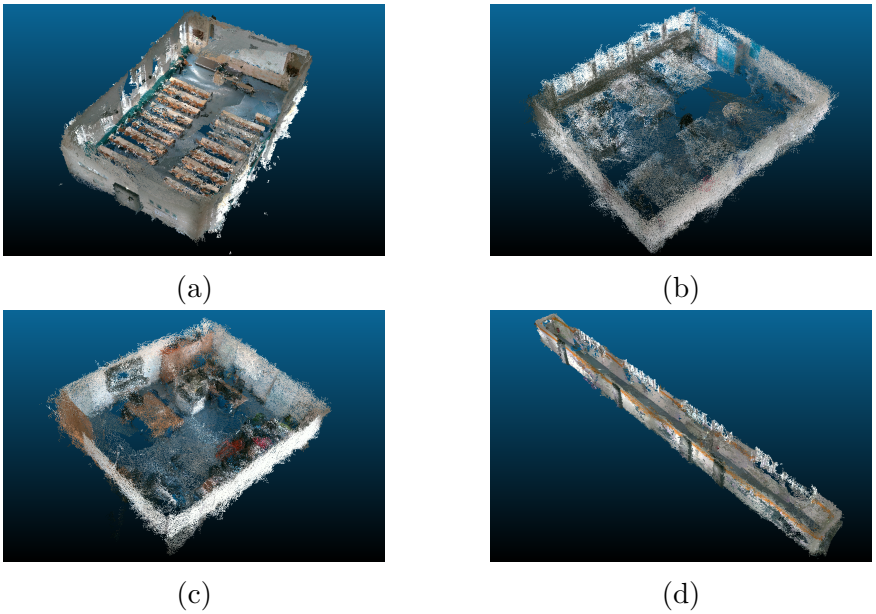
Since the unorganized point clouds do not grow incrementally over time, 2D-3D-S does not inherit any temporal component. Moreover, Armeni et al. [3] recorded the dataset with a static 360° camera. Therefore, the dataset is not entirely realistic, though being recorded in a real environment.

To our knowledge, no dataset meets the above criteria, is genuinely realistic, and includes a plane-focused ground truth. Therefore, we record an incrementally growing dataset in the Faculty of Computer Science at Otto-von-Guericke University Magdeburg, namely the FIN dataset.

To perform a comparison between the FIN and 2D-3D-S, we record a scene for each of the following scene types (see Figure 3.3):

- auditorium
- conference room
- hallway
- office

We focus on these four scene types because they are the most common in a realistic indoor environment. Lastly, since this is a novel dataset, we need to create a ground truth. The details thereof are explained in Section 4.4.



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

Figure 3.3: 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. Full-size figures can be found in Appendix B

3.3 Definition Real-Time

Finally, as mentioned in the beginning of this Chapter, we must define the meaning of *real-time* to determine the *real-time* applicability of a plane detection algorithm.

In Subsection 3.1.2, we introduce the differentiation between pre-processing and post-processing steps. It is possible that one phase of an algorithm accounts for the majority of the total calculation time and that the algorithm would be considered *real-time* applicable, if that phase were to be excluded. Because some steps can be covered by previous steps in the AR/VR system (see Figure 3.1), i.e., by the sensor or the SLAM algorithm, we give two definitions of *real-time*.

In general, and without taking the algorithms internal structure into consideration, 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 its internal map (Figure 3.1, [2]) or how frequently 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 $\mathbf{RT_{tot}}$ According to this upper limit of RTAB-MAP, we consider an algorithm to have *Total Real-Time* applicability, if it achieves an average frame rate of minimum 1, i.e., the total processing time of an algorithm lies under one second. In the remainder of this work, we use *Total Real-Time* and RT_{tot} interchangeably.

Real-Time Plane Calculation $\mathbf{RT_{calc}}$ Being a subset of *total 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 $1s$. Like RT_{tot}, we use *Real-Time Plane Calculation* and RT_{calc} interchangeably.

3.4 Summary

Many Augmented Reality applications have constraints in the form of a temporal component. Augmented or Virtual Reality applications that include plane detection are no exception. Thereby, these constraints apply to the plane detection algorithms as well.

In addition to time constraints, good quality is often tightly coupled to expensive or closed technology. In this work, we aim to evaluate the quality of *real-time* plane detection algorithms under the use of more affordable hardware, namely two Intel RealSense sensors. Therein, we are especially interested in the aspect of *real-time* applicability in a realistic environment.

At the beginning of this chapter, we state that three aspects are required for this evaluation: A set of plane detection algorithms, useful datasets, and a definition of *real-time*. The selection of the best plane detection algorithm, however, is non-trivial. After defining meaningful criteria for objective judgement, we select appropriate plane detection algorithms. Moreover, we select realistic datasets, one of which is a novel creation, and present two definitions of *real-time* namely RT_{tot} and RT_{calc}.

Chapter 4

Implementation

This chapter provides the implementation details of the outlined concept of the previous chapter. The following sections deal with the general system setup, the implementation of the algorithms selected in Section 3.1, and the creation and segmentation process of the chosen datasets (see Section 3.2).

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, *Noetic Ninjemys*[1], as well as *realsense-ros*[2] with all additional dependencies. Note that the version of *realsense-ros* changed over the course of this work.

4.2 Plane Detection Algorithms

In Chapter 2, we provide detailed information about the algorithms we select in Section 3.1. The following subsections deal with the implementation details thereof. Note that the subsections of RSPD and OPS are joined due to their similarities of implementation.

[1] http://wiki.ros.org/noetic
[2] https://github.com/IntelRealSense/realsense-ros/tree/ros1-legacy

4.2.1 RSPD & OPS

We implement RSPD[3] and OPS[4] using their respective open source implementations on GitHub. Note that, while the implementation of RSPD is provided by one of the authors, we could not determine whether the user who uploaded his implementation of OPS is affiliated with Sun and Mordohai [58]. Both methods are implemented in C++ and depend on the C++ linear algebra library *Eigen*[5], and the C++ API of the Point-Cloud Library [51], *libpcl-dev*.

4.2.2 3D-KHT

The authors of 3D-KHT, provide an implementation, in form of a Visual Studio project, on their website[6]. Since the laptop we use does not run Windows, we use *cmake-converter*[7] to convert the solution to a CMake project we can build using *make*. The dependencies of this implementation include the C++ library *Dlib* [31], as well as the OpenGL Utility Toolkit *GLUT*[8]. Lastly, the multi-processing API OpenMP[9] is required as well.

4.2.3 OBRG

To our knowledge, no open-source implementation is available for the algorithm. We, therefore, use our own implementation, which can be found on our Github repository[10].

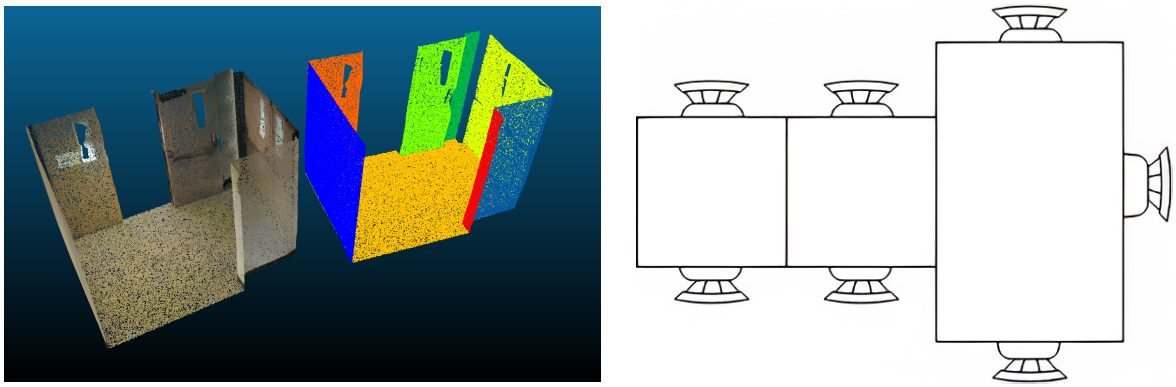
We implement the algorithm using Python. We choose to write our own octree implementation for spatial subdivision of our point cloud, since the implementation of public libraries like *open3d* [69] are limited in terms of leaf node functionality. The subdivision is followed by calculating the saliency features using *open3d*'s normal estimation function. We follow the pseudocode as stated in [60, 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.

[3] https://github.com/abnerrjo/PlaneDetection
[4] https://github.com/victor-amblard/OrientedPointSampling
[5] https://eigen.tuxfamily.org/index.php
[6] https://www.inf.ufrgs.br/ oliveira/pubs_files/HT3D/HT3D_page.html
[7] https://cmakeconverter.readthedocs.io/
[8] https://www.opengl.org/resources/libraries/glut/glut_downloads.php
[9] https://www.openmp.org/
[10] https://github.com/lupeterm/OBRG

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 [3, Table 1]. Since these annotated objects are not always planar, we cannot use them for the evaluation of plane detection algorithms. Thus, we create a ground truth that focuses on planar structures.

We use the open-source 3D point cloud and mesh processing software *CloudCompare*[11] 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) (b)

Figure 4.1: **(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 level of subjectivity involved. On average, the segmentation of a scene took 8-10 minutes, which, for all 272 scenes, would result in 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, thereby reducing the time to 18-23 hours.

[11] https://www.danielgm.net/cc/

The results of the manual segmentation process are documented in Table 4.1, which is inspired by [3, Table 7]. Like the original, it shows the amounts of scenes per scene type in each area. We extended the table with a column dedicated to the planes included in each scene type. According to the table, a total amount of 3410 planes are included in the dataset.

Table 4.1: 2D-3D-S dataset statistics. Shown are the number of scenes per category and for how many we created a ground truth ($\#GT/\#Total$). Note that the rightmost column reports the number of segmented planes per scene category and does *not* correspond to other columns in this table.

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**

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*. All scenes are recorded inside the Faculty of Computer Science 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 the environment to the best of our ability. We save each incremental map update to a file for further usage. The statistics of all recordings are summarized in Table 4.2.

Given the differences in spatial dimension, the recordings of each scene also differ in duration 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 (see Table 4.2).

Table 4.2: Statistics of the *FIN* dataset. The duration reports the time spent on a recording, the maximum size denotes the number of points in the most recent point cloud. The rightmost column shows the number of manually segmented planes for each scene. The bottom row shows the total values over the entire dataset.

Scene	Duration	Max. Size	Planes
Auditorium	296	656.599	41
Conference Room	113	387.183	11
Hallway	174	303.780	5
Office	125	364.165	15
	708	1.711.727	72

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 point cloud 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. Figure 4.2 shows the final point cloud and the manually created ground truth gt_{end} of the *hallway* scene on the far right. On the left thereof, two point clouds of earlier stages of the recording are shown, as well as their dynamically created ground truth. 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. The number of resulting planes are reported in the right-most column of Table 4.2.

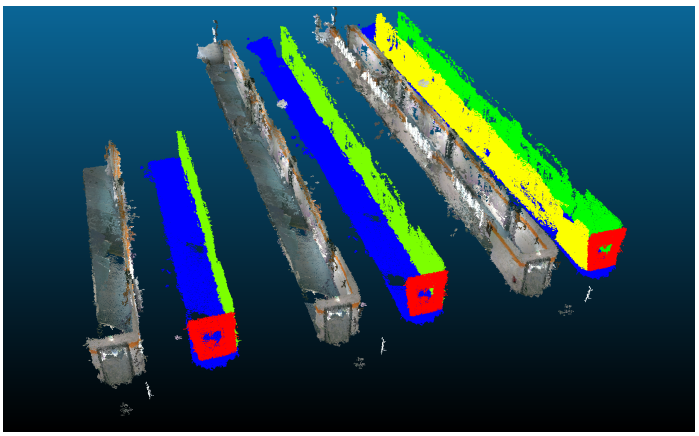


Figure 4.2: Dynamic ground truth generation shown by the example of three point clouds from the *hallway* scene, as well as the corresponding ground truth. The rightmost pair shows the final state of the point cloud and its ground truth.

Chapter 5

Evaluation

In this chapter, the plane detection algorithms selected in Section 3.1 are uniformly compared. We enter this chapter by outlining the protocol followed during this evaluation. Afterwards, we present and analyse the results thereof.

5.1 Protocol

This work aims to determine which plane detection algorithm is the most suitable for an AR/VR systems in a realistic environment. For this decision, we compare the algorithms selected in Chapter 3. In Section 3.2, we established that the 2D-3D-S dataset is not entirely suitable for evaluating real-world applicability due to its static nature, i.e., non-incremental growth and having been recorded by a stationary 360° camera. To mitigate this, we created the realistic FIN dataset. Since both datasets are fundamentally different, we will perform the experiments and the analysis separately and then compare the results. All experiments are performed on an AMD Ryzen 7 5800H CPU with 16GB of RAM. 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

The following two paragraphs detail the metrics we use in this evaluation. We introduce our methodology of measuring accuracy, as well as the calculation times. Note that by using the term "time frame t" or "time step t", we refer to the state of the point cloud at a given time t.

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. We calculate the *Precision*, as this reports the percentage of correctly detected points within the planes an algorithm returned. Similarly, we calculate the *Recall* because it gives information about the percentage of correctly detected points in comparison to the number of points that *could* be detected. Using these metrics separately is problematic as we cannot expect the distribution of "plane" to "non-plane" to be even. The *F1-Score* is the harmonic mean of the *Precision* and the *Recall* and therefore measures the balance between the *Precision* and *Recall*. Intuitively, 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. The procedure of calculation is taken from [2, 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.

Time Measurements We are also 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 below $1s$. Furthermore, we consider an algorithm to achieve *Real-Time Plane Detection* (RT_{calc}) applicability if its plane detection and post-processing steps run faster than $1s$.

When recording a real environment, the point clouds likely grow incrementally with each map update. Therefore, the average calculation time alone is of limited significance, as we assume the calculation times to grow with the cloud size. Based on this assumption, it is necessary to analyze the relationship between the point cloud size and the calculation time in addition to average values. Note that we refer to a point cloud's size by its number of contained points in the remainder of this chapter.

Following the separation of algorithms into phases (see Paragraph 3.1.2), we divide the calculation time into the pre-processing time t_{pre}, the time spent during plane detection t_{calc}, and the duration of post-processing steps t_{post}. This separation enables an in-depth analysis of both average results and the results over time. Since we have two definitions of *real-time*, we introduce two metrics of calculation time:

1. The sum of t_{calc} and t_{post} allows us to determine whether an algorithm is *Real-Time Plane Detection* applicable, *and*
2. to determine whether an algorithm is *totally real-time* applicable, we consider the total computation time t_{tot}, which is the sum of the individual times (see Eq. 5.1)

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

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

Table 5.1: Parameter configuration of RSPD used for the experiments.

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

2D-3D-S We use the parameters of the provided implementation[1] for the 2D-3D-S experiment. 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 [2, Section 3.3], $k = 30$ is used in the official implementation. We adopted the latter because, in our experience, it produces sufficient results while reducing the pre-processing time.

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.

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

5.1.2.2 OPS

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

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

2D-3D-S The parameter configuration used for the 2D-3D-S experiment is shown in the first row of Table 5.2. Like the parameterization of RSPD, we took this configuration from the provided implementation[2]. We use a sampling rate α_s of 3% and a neighborhood size 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 [58, Section 4A].

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

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

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

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 [33]

[2]https://github.com/victor-amblard/OrientedPointSampling

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 [33, 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}, 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: $\phi_{num} \in [20, 100]$, $\rho_{num} \in [50, 600]$, $s_{level} \in [1, 5]$, $\theta_N \in [50, 1000]$, $s_\alpha \in [5, 23]$, 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 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

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

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%**

2D-3D-S The used configurations for the experiments are shown in Table 5.4. The parameterization was determined during the implementation process and showed a reasonable compromise between efficiency and accuracy. We considered the reported parameterizations in [60, Tables 1, 4, 7]. However, since the authors used three large outdoor scenes for their evaluation, they would likely not be optimal for smaller, indoor scenes. We confirmed this assumption based on a small set of empiric experiments with scenes from the 2D-3D-S dataset, wherein we compared the results for the reported parameterization and the one we determined. Due to the low level of noise, we assign a very small tolerance to the residual threshold θ_{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 [60, 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 the *F1-Score* of each algorithm were calculated. The computation times were measured and divided into pre-processing t_{pre}, plane detection t_{calc}, and post-processing t_{post}. 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.

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 t_{tot}. Note that the absence of post-processing steps is denoted as "/". All times are measured in seconds.

Algorithm	**Precision**	**Recall**	**F1-Score**	$\mathbf{t_{pre}}$	$\mathbf{t_{calc}}$	$\mathbf{t_{post}}$	$\mathbf{t_{tot}}$
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

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 With an average of $0.71s$ spent in pre-processing, and an average of $1.03s$ spent in plane detection, 3D-KHT only needs an average total of $1.74s$ to process an entire point cloud and thereby achieves the lowest calculation times. RSPD is the only algorithm that scores similar t_{calc} values, with an average of $1.04s$. However, RSPD's time spent in pre-processing is the highest among the algorithms. With an average t_{tot} of $\sim 25s$ and $\sim 63s$, 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-Time Plane Detection* applicability, exceeding it by only $0.4s$ and $0.3s$, respectively

Relationship of Time and Size 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.

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 $\sim 7 \cdot 10^6$ and $\sim 9 \cdot 10^6$ is hardly noticeable considering .

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 small number of values above 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 in the range of $10s - 100s$, even for smaller cloud sizes at $\sim 1 \cdot 10^6$. The t_{calc} values of OBRG do not appear to be dependent on the

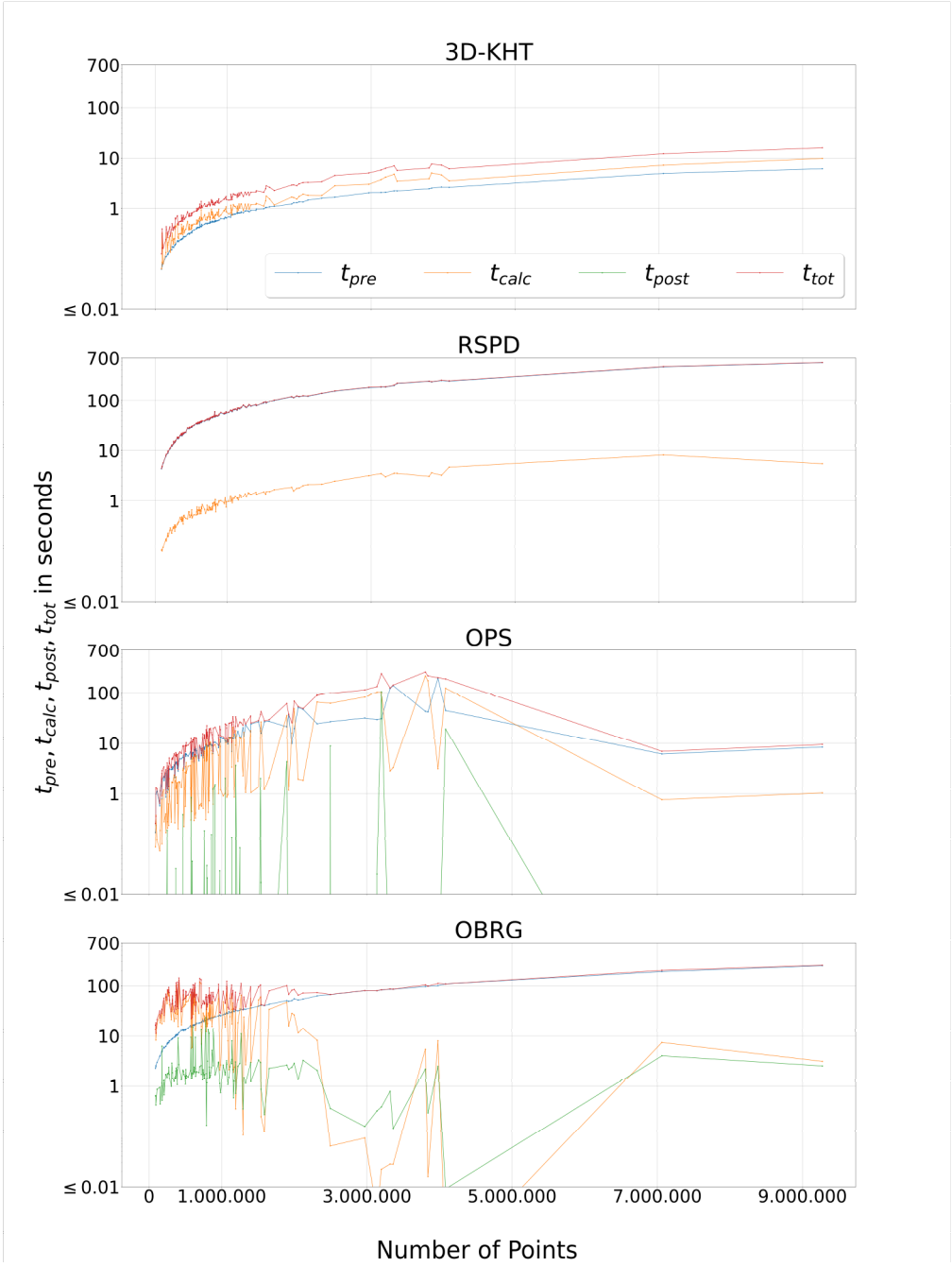


Figure 5.1: The time spent in pre-processing t_{pre}, plane detection t_{calc}, post-processing t_{post}, and the total calculation time t_{tot} per point cloud size of the 2D-3D-S dataset. Note that the y-axis is scaled logarithmically to the base of ten.

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}. As this parameter is vital for the calculations in all phases, all calculation times of OBRG are likely to show improvement upon applying a parameter optimization of l_{max} based on the dataset. From our experience, the mid-sized scenes in the 2D-3D-S dataset are often long and narrow hallways. Assuming an inappropriate octree depth parameter, the subdivision could effectively split the hallway, returning a set of chunks that are in no way to be considered planar. This would consequently lead to an early termination due to failed thresholds, thus reducing the calculation times (and most likely yielding low accuracy).

Summary 2D-3D-S Experiment With an average of ∼89%, OPS has the highest value in *Precision*, while RSPD achieves the highest *Recall* and *F1-Score* with ∼90% and ∼87%, respectively. 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.

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.

Adhering strictly to our definitions of *real-time*, no algorithm achieves *real-time* applicability in this experiment. However, we might consider RSPD and 3D-KHT to be *Real-Time Plane Detection* applicable since the point cloud sizes vary greatly, and the average duration of their plane detection phase is only $0.03s - 0.04s$ greater than the $1s$ threshold. Furthermore, as Figure 5.1 suggests, RSPD and 3D-KHT are *Real-Time Plane Detection* applicable for point clouds with $\lesssim 1 \cdot 10^6$ points.

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, again divided into pre-processing t_{pre}, plane detection t_{calc}, and post-processing t_{post}.

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.

Algorithm	**Precision**	**Recall**	**F1-Score**	$\mathbf{t_{pre}}$	$\mathbf{t_{calc}}$	$\mathbf{t_{post}}$	$\mathbf{t_{tot}}$
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

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 and the total time.

Accuracy OPS has the highest average *Precision* of the algorithms, with almost 70%. RSPD achieves the highest values for *Recall* and the *F1-Score* with ∼61% and ∼59%, respectively. 3D-KHT and OBRG achieve 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 With almost 15s, RSPD spends the most time in pre-processing among the algorithms. In contrast, RSPD has the shortest time spent during plane detection, with an average of 0.19s. Overall, 3D-KHT needs the shortest time for the complete computation t_{tot} of a time step with an average of 0.43s. OPS achieves comparatively average total calculation time with ∼6s, and OBRG takes the longest overall to compute a time step with more than 20s.

RSPD and 3D-KHT achieve *Real-Time Plane Detection* applicability. Additionally, 3D-KHT also achieves *Total Real-time* applicability, with an average total time of 0.43s. OPS can almost be considered to be in RT_{calc} with a subtotal calculation time of 1.02s, thereby taking 0.02s too long. With an average pre-processing time of ∼6s and an average of ∼15s spent in plane detection, OBRG qualifies for neither of the introduced definitions of *real-time*.

Relationship of Time and Size As mentioned before in Paragraph 5.1.1, we are interested in 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 number of points in the *auditorium* scene of the FIN data set.

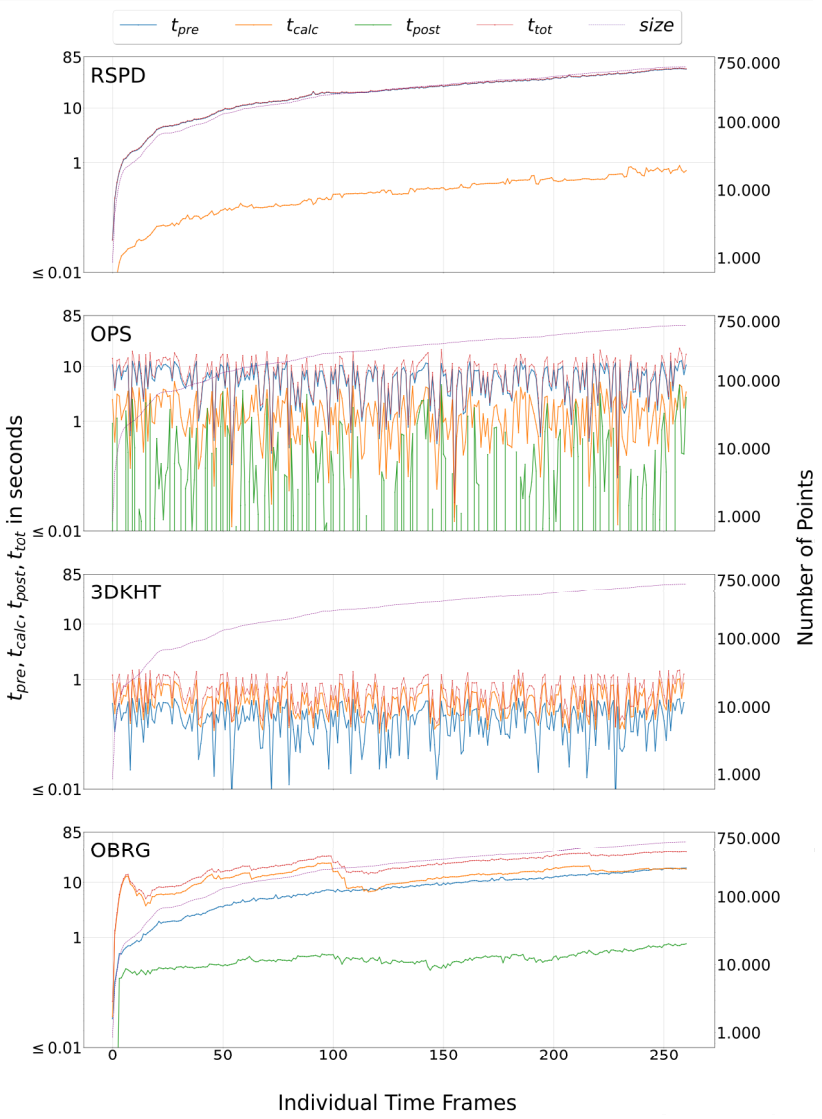


Figure 5.2: Time spent in pre-processing t_{pre}, plane detection t_{calc}, post-processing t_{post}, and total calculation time t_{tot} of the *auditorium* scene, and point cloud size of each time frame. Note that both y-axes are scaled logarithmically to the base of ten. *Time Frames* are defined in Subsection 5.1.1.

Note that we created similar graphs for the other scenes of the FIN dataset. However, they do not introduce new information into the argumentation and are presented in Appendix A for completeness and spatial reasons. We consider the *auditorium* scene as the most representative because it contains the longest recording, and thus contains the most data.

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, $\sim 1s$ for 3D-KHT and $\sim 6s$ for OPS. 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.

The apparent upper bound of $\sim 1s$ in the calculation times qualifies 3D-KHT for *Total Real-time* applicability. We consider RSPD *Real-Time Plane Detection* applicable, as the duration of the plane detection step consistently stays under $1s$.

Summary FIN Experiment OPS has the highest average *Precision*, and RSPD has the largest percentages of *Recall* and the *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 $t_{pre} = 0.14s$ and $t_{tot} = 0.43s$, 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.

The development over time supports the average calculation times of 3D-KHT, as the calculation seems independent of the cloud size. Thereby, we determine 3D-KHT to be *Total Real-time* applicable. Furthermore, RSPD achieves *Real-Time Plane Detection* applicability in the FIN experiment, as the plane detection step takes $< 1s$ for all time frames. Lastly, the average subtotal calculation time of OPS is slightly longer than $1s$. However, given the apparent upper bound and considerable fluctuations, we cautiously consider OPS to be *Real-Time Plane Detection* applicable in the FIN experiment.

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 oriented around a given value, however, this value differs between the experiments ($\sim 0.3s$ for the FIN experiment and $\sim 3s$ for the 2D-3D-S experiment). 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 worth noting that the cloud sizes widely differ between the experiments, as the maximum size of the FIN experiment ($\sim 0.75 \cdot 10^6$) is very small, compared to largest scene of the 2D-3D-S dataset ($\sim 9 \cdot 10^6$). Nonetheless, since Figure 5.1 portrays a proportionality, even for smaller clouds, the reason for different curves is likely the difference of parameterization.

In Paragraph 5.2.1, we report that no algorithm is *real-time* applicable. We state, that RSPD and 3D-KHT can be considered *Real-Time Plane Detection* applicable since the cloud sizes vary and the difference between their average t_{calc} times and the threshold of $1s$ is very small. In Paragraph 5.2.2, we consider 3D-KHT *totally real-time* applicable due to the observed upper bound in calculation time in combination with a low average t_{tot} value of $0.43s$. Additionally, we consider $RSPD \in RT_{calc}$ because RSPD takes consistently less than $1s$ during the plane detection step. The results of neither experiment show a *real-time* applicability of OBRG.

5.3 Summary

OBRG achieves neither the highest nor lowest values in any experiment. OPS has the highest *Precision* 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.

3D-KHT is *Real-Time Plane Detection* applicable in the 2D-3D-S experiment and *totally real-time* applicable in the FIN experiment. RSPD achieves *Real-Time Plane Detection* applicable in both experiments. We consider OPS to be *Real-Time Plane Detection* applicable. These *real-time* applicabilities of all algorithms are summarized in Table 5.7.

Table 5.7: Real-time applicabilities of the selected plane detection algorithms given the results from both experiments. Note that RT_{tot} implies RT_{calc}. "/" denotes no *real-time* applicability. "*" denotes that the given *real-time* applicability is coupled with a restriction. We refer to the text for details.

Experiment	**RSPD**	**OPS**	**3D-KHT**	**OBRG**
2D-3D-S	RT_{calc}*	/	RT_{calc}*	/
FIN	RT_{calc}	RT_{calc}*	$\mathbf{RT_{tot}}$	/

Chapter 6

Conclusion

Modern man-made environments, especially indoors, contain a large number of planar structures. The automatic detection thereof has become a vital part of many Augmented or Virtual Reality systems. Underlying temporal constraints often dictate the processing times of these applications and, therein, the process of plane detection. Real-time plane detection is already possible, although the hardware cost that enables efficient and precise detection is likely not affordable to the general consumer. Moreover, the real-world applicability of plane detection algorithms depends on numerous aspects rendering the selection of a suitable algorithm non-trivial. Therefore, we performed a uniform comparison of algorithms on affordable hardware to evaluate their applicability in a realistic environment.

A set of plane detection algorithms, appropriate datasets, and a definition of *real-time* are needed to perform this evaluation. In the *Concept* (see Chapter 3), we first introduced a set of helpful criteria for the subsequent selection of plane detection algorithms. We followed the same approach for the selection of datasets. Lastly, we introduced two definitions of *real-time*, wherein we differentiate between *real-time* calculation time including and excluding pre-processing.

In Chapter 4, we provided details regarding the implementation of the selected algorithms. Moreover, since its provided ground truth does not focus on planes, we described our manual segmentation process of the 2D-3D-S dataset. Lastly, we presented the novel FIN dataset and outlined how the corresponding ground truth is dynamically created based on a single ground truth of the last recorded point cloud.

We enter Chapter 5 with the evaluation protocol. Therein, we outline the evaluation metrics to calculate and specify the algorithm parameterizations of each experiment. We presented and subsequently compared the individual results. In both experiments, RSPD has the overall best accuracy among the algorithms. While OPS and OBRG achieve a similar average *Precision*, no other algorithm yields comparable values for *Recall* or *F1-Score* (see Tables 5.5 and 5.6). The accuracy metrics of all algorithms drop

by roughly 30% between the experiments. The results uniformly show that 3D-KHT is the fastest among the selected algorithms. The *Hough Transform*-based algorithm proposed by Limberger and Oliveira [33] ran, on average, 32x faster than the other algorithms. As Table 5.7 indicates, 3D-KHT is the only algorithm that achieves RT_{tot} in the FIN experiment, which is supported by the apparent upper limit shown in Figure 5.2. RSPD achieves RT_{calc}, and OPS borders on *Real-Time Plane Detection* applicability as well. RSPD has the longest pre-processing times with an average of $\sim 63s$ for the 2D-3D-S experiment and $\sim 15s$ for the FIN experiment, respectively. In contrast, RSPD has the shortest average plane detection times in the FIN experiment and takes only a tenth of a second longer than 3D-KHT's plane detection phase in the 2D-3D-S experiment.

Based on these results, we conclude that 3D-KHT is the only algorithm that achieves *Total Real-Time* applicability. However, considering accuracy, 3D-KHT is inferior to RSPD.

6.1 Limitations

This section deals with the limitations of the concept, evaluation, and results thereof.

Algorithm Parameterization Since the focus of this work is not the optimization of plane detection algorithms but rather the evaluation thereof, the parameterizations used during the experiments are likely non-optimal. Furthermore, a thorough optimization, including its required effort, would go beyond the scope of this work. Integrating a plane detection algorithm into an application would require further parameter optimization with regard to the expected environment, the used sensors, and the general use case.

Manual Segmentation The subjective nature of the manual segmentation process described in Section 4.3 influences the evaluation. While this does not seem to pose a dramatic effect on the 2D-3D-S results, the segmentation can lead to errors due to the level of noise in the FIN dataset. For instance, with increasing "thickness" of planes, the number of possible orientations of a detected plane also increases (see Figure 6.1). The red segment in the figure represents the ground truth we would choose to represent the plane as it is more aligned to the true wall, and has a lower ratio of noise compared to the other options in green and blue. If an algorithm detected a plane in one of the four other variants, both the voxel overlap and the accuracy would decrease, even if both planes correspond.

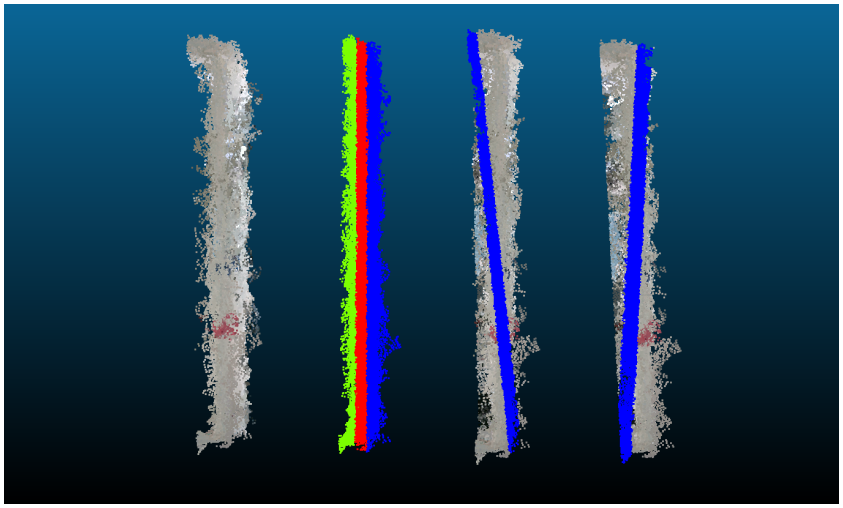


Figure 6.1: Possible Plane Orientations. Shown is the top view of a wall with a width of ∼1m from the *conference room* scene of the FIN dataset is shown on the left. The colors indicate different possible plane orientations.

Used Technology A small caveat to note is that the FIN dataset is recorded using a specific combination of technology, namely Intel's T265, D455, and the corresponding software including RTAB-MAP. Since these sensors were chosen as representatives for consumer off-the-shelf hardware, the results point out the general applicability in a realistic environment. However, the results may vary when employing different technology.

6.2 Future Work

In this section, we elaborate on topics of further research.

Normal Estimation We see the most potential for improvement in the pre-processing steps of the algorithms. In our experience, the extent of the normal vector estimation influences the pre-processing time. For instance, RSPD estimates the normal vectors of the entire point cloud, whereas OPS only estimates a certain percentage of the point cloud. The differences in pre-processing times (see Tables 5.5 and 5.6) raise the question of what the extents of *real-time* applicability are if the normal vectors of the point cloud are known before the plane detection step in the application (compare Figure 3.1). For instance, RTAB-MAP can estimate and export the normal vectors of the point cloud by modifying its odometry approach. Therefore, RSPD, OPS, and OBRG would not need

to estimate the normal vectors, reducing the pre-processing time greatly. However, this would require further research.

Cloud Size Reduction When recording environments similar to the hallway or the *auditorium* scene, it is often the case that the spatial dimension of the point cloud grows beyond the distance limitations of the sensor, e.g., the recorded hallway spans longer than the sensor can "see". It is, therefore, not necessary to re-calculate the planes in areas past the sensor's reach. We are interested in a plane detection method that restricts the plane detection to a certain radius around the sensor's position and a subsequent merging of old and new planes. Additionally, this could be the basis for a plane-based SLAM approach.

Outdoor Environment In Chapter 1, we limited ourselves to evaluating plane detection algorithms in indoor environments. Therefore, we cannot make any statements about the applicability in outdoor environments. Since we compiled results in a realistic indoor environment, it would be interesting to evaluate the generalization of these algorithms. As discussed throughout this thesis, a comparison needs uniformity. Since we created an indoor dataset, selected datasets, and provided all necessary metrics and definitions, only an appropriate dataset is needed.

Appendix A

FIN Scene Results

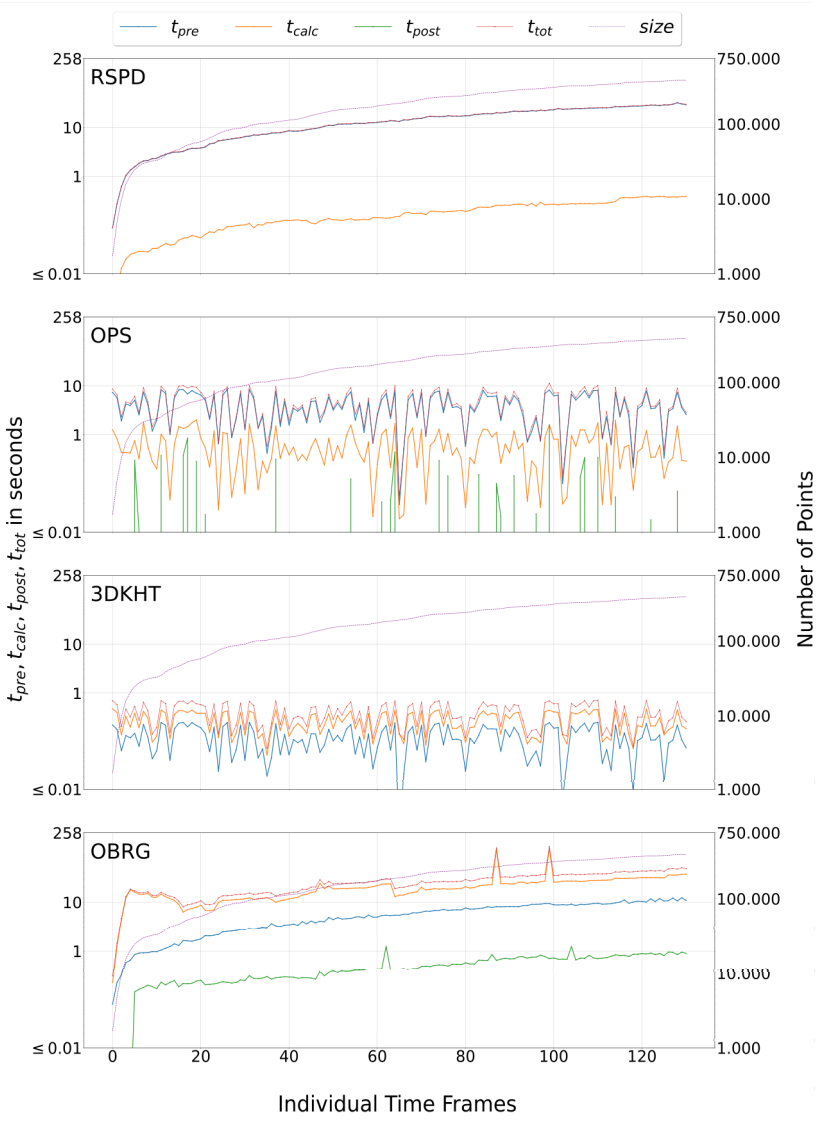


Figure A.1: Time spent in pre-processing t_{pre}, plane detection t_{calc}, post-processing t_{post}, and total calculation time t_{tot} of the *conference room* scene, and point cloud size of each time frame. Note that both y-axes are scaled logarithmically to the base of ten. *Time Frames* are defined in Subsection 5.1.1.

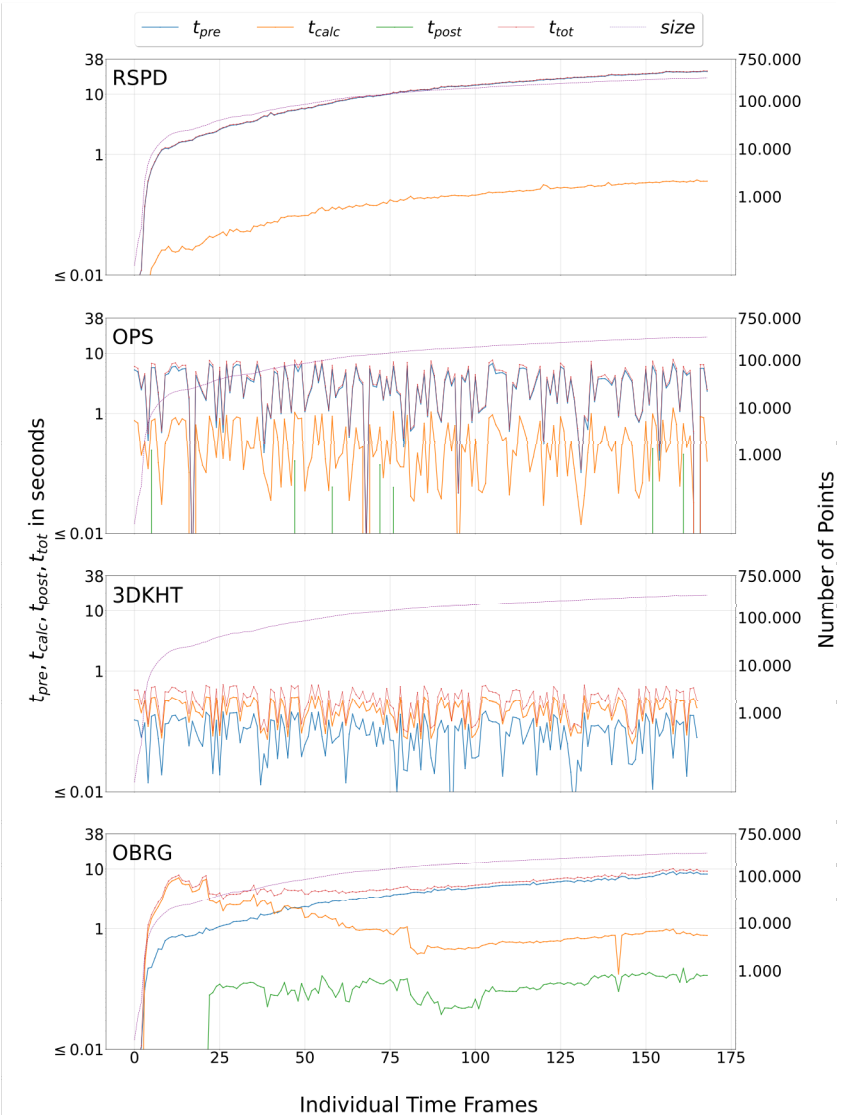


Figure A.2: Time spent in pre-processing t_{pre}, plane detection t_{calc}, post-processing t_{post}, and total calculation time t_{tot} of the *hallway* scene, and point cloud size of each time frame. Note that both y-axes are scaled logarithmically to the base of ten. *Time Frames* are defined in Subsection 5.1.1.

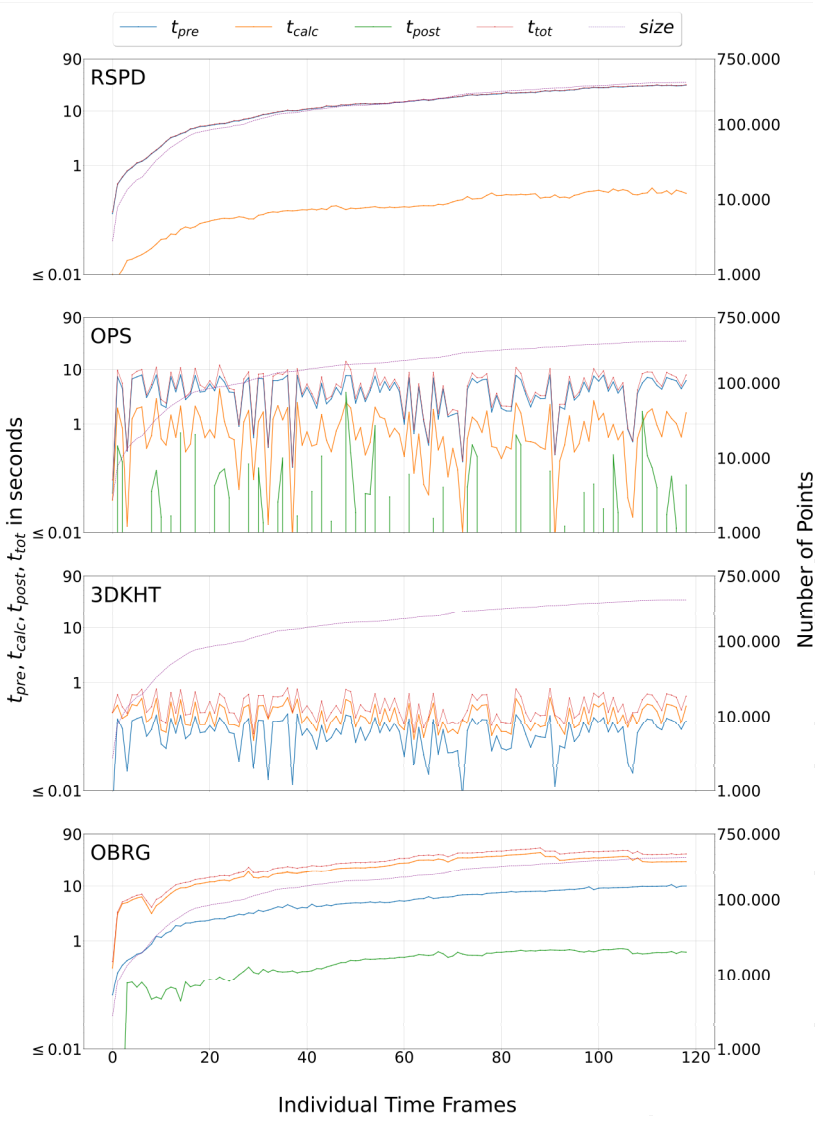


Figure A.3: Time spent in pre-processing t_{pre}, plane detection t_{calc}, post-processing t_{post}, and total calculation time t_{tot} of the *office* scene, and point cloud size of each time frame. Note that both y-axes are scaled logarithmically to the base of ten. *Time Frames* are defined in Subsection 5.1.1.

Appendix B

FIN Dataset Scenes

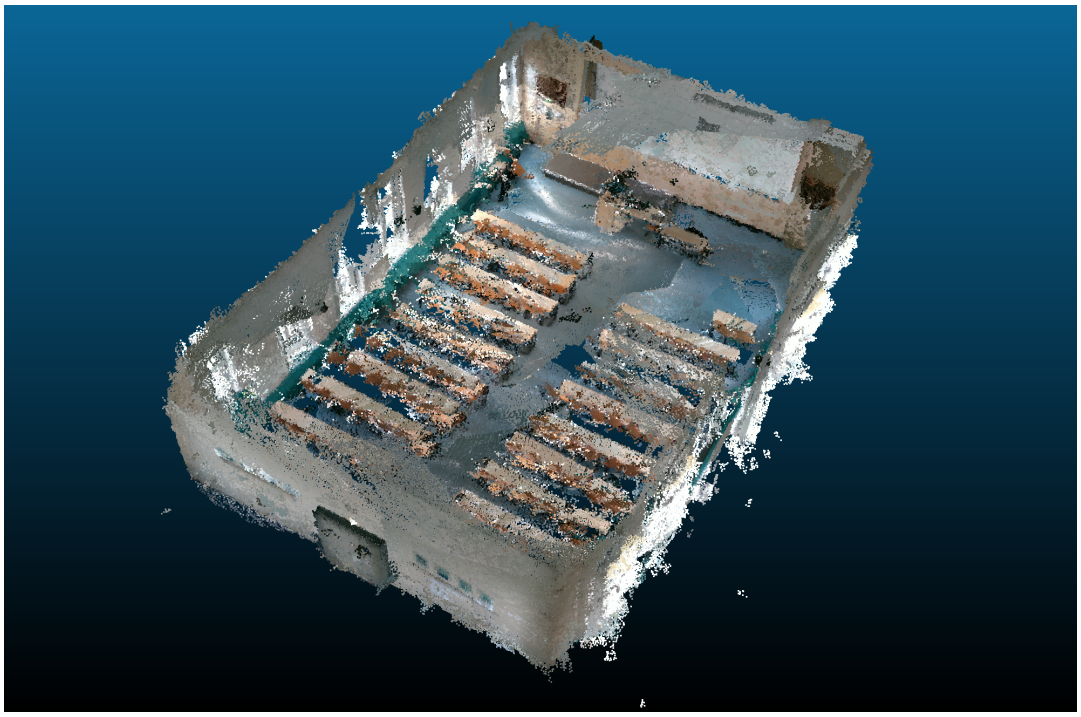


Figure B.1: Full-size view of the hallway scene of the FIN dataset.

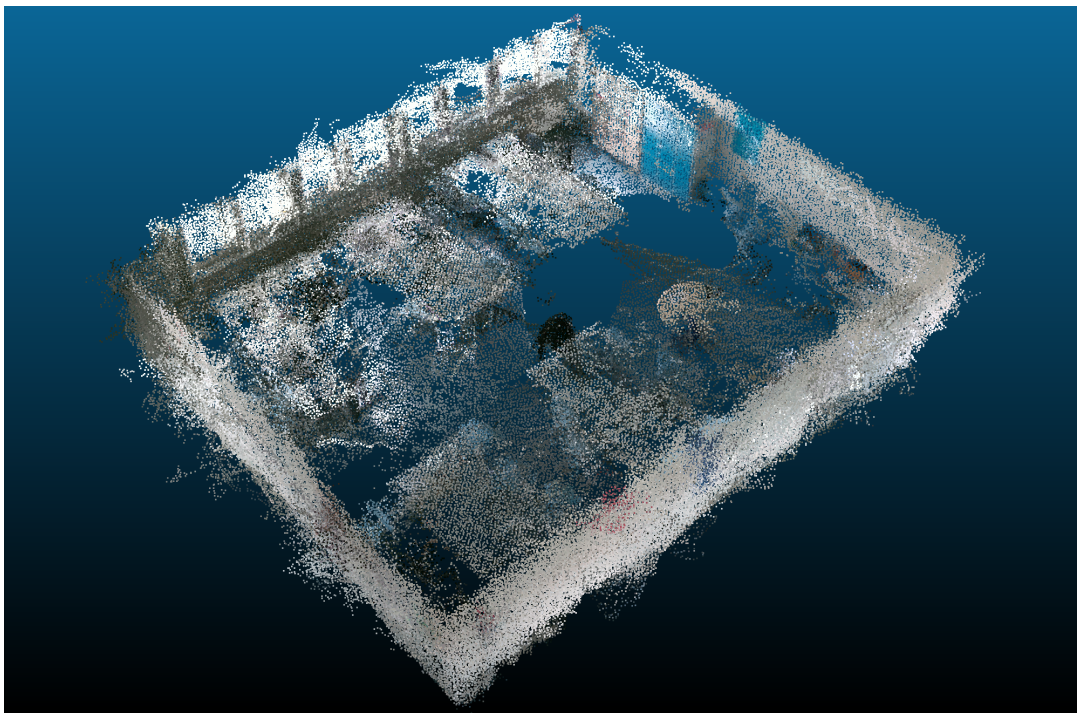


Figure B.2: Full-size view of the conference room scene of the FIN dataset.

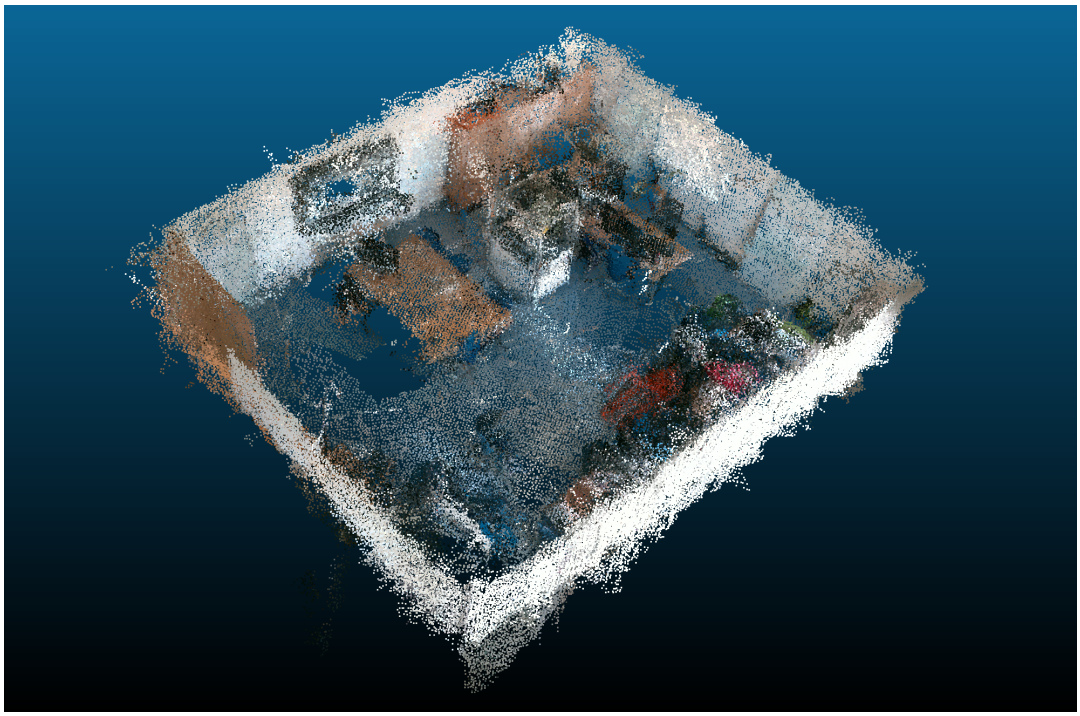


Figure B.3: Full-size view of the office scene of the FIN dataset.

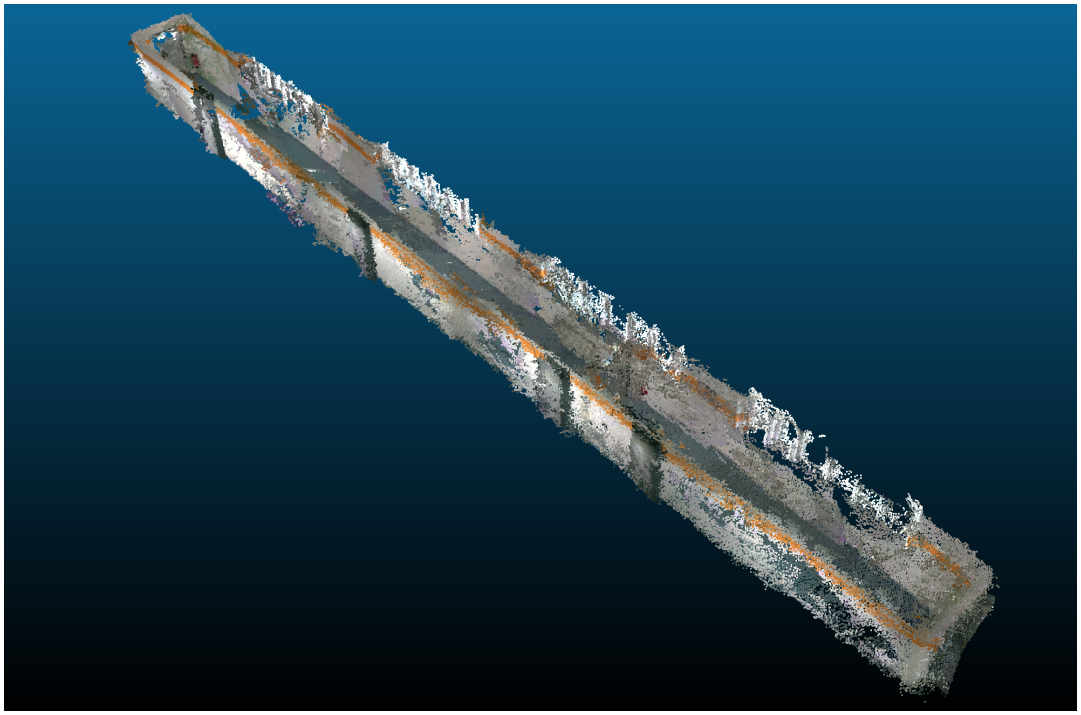


Figure B.4: Full-size view of the hallway scene of the FIN dataset.

Bibliography

[1] Samir Agarwala, Linyi Jin, Chris Rockwell, and David F Fouhey. "Planeformers: From sparse view planes to 3d reconstruction". In: *European Conference on Computer Vision (ECCV)*. Springer. 2022, pp. 192–209.

[2] Abner M. C. Araújo and Manuel M. Oliveira. "A robust statistics approach for plane detection in unorganized point clouds". In: *Pattern Recognition* 100 (2020), pp. 107–115.

[3] I. Armeni, A. Sax, A. R. Zamir, and S. Savarese. "Joint 2D-3D-Semantic Data for Indoor Scene Understanding". In: *ArXiv e-prints* (2017). arXiv: `1702.01105 [cs.CV]`.

[4] Ramy Ashraf and Nawal Ahmed. "FRANSAC: Fast RANdom Sample Consensus for 3D Plane Segmentation". In: *International Journal of Computer Applications* 167.13 (2017), pp. 30–36.

[5] Joydeep Biswas and Manuela M. Veloso. "Fast Sampling Plane Filtering , Polygon Construction and Merging from Depth Images". In: *Robotics: Science and Systems, RGB-D Workshop*. 2011.

[6] Michael Bloesch, Michael Burri, Sammy Omari, Marco Hutter, and Roland Siegwart. "Iterated extended Kalman filter based visual-inertial odometry using direct photometric feedback". In: *The International Journal of Robotics Research* 36.10 (2017), pp. 1053–1072.

[7] Dorit Borrmann, Jan Elseberg, Kai Lingemann, and Andreas Nüchter. "The 3D Hough Transform for plane detection in point clouds: A review and a new accumulator design". In: *3D Research* 2.2 (2011), p. 3.

[8] Carlos Campos, Richard Elvira, Juan J Gómez Rodriguez, José MM Montiel, and Juan D Tardós. "Orb-slam3: An accurate open-source library for visual, visual–inertial, and multimap slam". In: *Transactions on Robotics* 37.6 (2021), pp. 1874–1890.

[9] Denis Chekhlov, Andrew Gee, Andrew Calway, and Walterio Mayol-Cuevas. "Ninja on a Plane: Automatic Discovery of Physical Planes for Augmented Reality Using Visual SLAM". In: 2007, pp. 153–156.

[10] J.M. Coughlan and A.L. Yuille. “Manhattan World: compass direction from a single image by Bayesian inference”. In: *International Conference on Computer Vision (ICCV)*. Vol. 2. IEEE, 1999, pp. 941–947.

[11] Adam Dai, Greg Lund, and Grace Gao. “PlaneSLAM: Plane-based LiDAR SLAM for Motion Planning in Structured 3D Environments”. In: arXiv:2209.08248 (2022). arXiv:2209.08248 [cs].

[12] Davison. “Real-time simultaneous localisation and mapping with a single camera”. In: *Proceedings Ninth IEEE International Conference on Computer Vision*. IEEE, 2003, 1403–1410 vol.2.

[13] Andrew J Davison, Ian D Reid, Nicholas D Molton, and Olivier Stasse. “MonoSLAM: Real-time single camera SLAM”. In: *transactions on pattern analysis and machine intelligence* 29.6 (2007), pp. 1052–1067.

[14] Richard O. Duda and Peter E. Hart. “Use of the Hough Transformation to Detect Lines and Curves in Pictures”. In: *Commun. ACM* 15.1 (1972), pp. 11–15.

[15] Felix Endres, Jürgen Hess, Jürgen Sturm, Daniel Cremers, and Wolfram Burgard. “3-D mapping with an RGB-D camera”. In: *transactions on robotics* 30.1 (2013), pp. 177–187.

[16] Jakob Engel, Vladlen Koltun, and Daniel Cremers. “Direct sparse odometry”. In: *Transactions on pattern analysis and machine intelligence* 40.3 (2017), pp. 611–625.

[17] Chen Feng, Yuichi Taguchi, and Vineet R. Kamat. “Fast plane extraction in organized point clouds using agglomerative hierarchical clustering”. In: *International Conference on Robotics and Automation (ICRA)*. IEEE, 2014, pp. 6218–6225.

[18] Ankur Handa, Thomas Whelan, John McDonald, and Andrew J. Davison. “A benchmark for RGB-D visual odometry, 3D reconstruction and SLAM”. In: *International Conference on Robotics and Automation (ICRA)*. IEEE, 2014, pp. 1524–1531.

[19] Ankur Handa, Thomas Whelan, John B. McDonald, and Andrew J. Davison. “A benchmark for RGB-D visual odometry, 3D reconstruction and SLAM”. In: *International Conference on Robotics and Automation (ICRA)* (2014), pp. 1524–1531.

[20] Peter E. Hart. “How the Hough transform was invented [DSP History]”. In: *Signal Processing Magazine* 26.6 (2009), pp. 18–22.

[21] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. “Mask R-CNN”. In: *International Conference on Computer Vision (ICCV)*. IEEE, 2017, pp. 2980–2988.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. “Deep Residual Learning for Image Recognition”. In: (2016), pp. 770–778.

[23] Alejandro Hidalgo-Paniagua, Miguel A. Vega-Rodriguez, Nieves Pavón, and Joaquin Ferruz. “A Comparative Study of Parallel RANSAC Implementations in 3D Space”. In: *International Journal of Parallel Programming* 43.5 (2015), pp. 703–720.

[24] Adam Hoover, Gillian Jean-Baptiste, Patrick Flynn, Horst Bunke, Dmitry Goldgof, Kevin Bowyer, David Eggert, Andrew Fitzgibbon, and Robert Fisher. “An Experimental Comparison of Range Image Segmentation Algorithms”. In: *Trans. Pattern Anal. Mach. Intell.* 18 (1996), pp. 673–689.

[25] Daniel Huber. “The ASTM E57 File Format for 3D Imaging Data Exchange”. In: *Proceedings of SPIE Electronics Imaging Science and Technology Conference, 3D Imaging Metrology*. Vol. 7864. 2011.

[26] Lam Huynh, Phong Nguyen-Ha, Jiri Matas, Esa Rahtu, and Janne Heikkilä. “Guiding Monocular Depth Estimation Using Depth-Attention Volume”. In: *European Conference on Computer Vision (ECCV)*. .Part XXVI. Springer-Verlag, 2020, pp. 581–597.

[27] David Jurado, Juan M. Jurado, Lidia Ortega, and Francisco R. Feito. “GEUINF: Real-Time Visualization of Indoor Facilities Using Mixed Reality”. In: *Sensors* 21.4 (2021), p. 1123.

[28] Michael Kaess. “Simultaneous localization and mapping with infinite planes”. In: *International Conference on Robotics and Automation (ICRA)*. IEEE, 2015, pp. 4605–4611.

[29] Christian Kerl, Jurgen Sturm, and Daniel Cremers. “Dense visual SLAM for RGB-D cameras”. In: *International Conference on Intelligent Robots and Systems (IROS)*. IEEE/RSJ, 2013, pp. 2100–2106.

[30] Dongchul Kim, Seungho Chae, Jonghoon Seo, Yoonsik Yang, and Tack-Don Han. “Realtime plane detection for projection Augmented Reality in an unknown environment”. In: *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2017, pp. 5985–5989.

[31] Davis E. King. “Dlib-ml: A Machine Learning Toolkit”. In: *Journal of Machine Learning Research* 10 (2009), pp. 1755–1758.

[32] 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”. In: *Journal of Field Robotics* 36.2 (2019), pp. 416–446.

[33] Frederico A. Limberger and Manuel M. Oliveira. “Real-Time Detection of Planar Regions in Unorganized Point Clouds”. In: *Pattern Recognition* 48.6 (2015), pp. 2043–2053.

[34] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. “Feature pyramid networks for object detection”. In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2017, pp. 2117–2125.

[35] Chen Liu, Kihwan Kim, Jinwei Gu, Yasutaka Furukawa, and Jan Kautz. “PlaneRCNN: 3D Plane Detection and Reconstruction From a Single Image”. In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE/CVF, 2019, pp. 4445–4454.

[36] Chen Liu, Jimei Yang, Duygu Ceylan, Ersin Yumer, and Yasutaka Furukawa. “PlaneNet: Piece-Wise Planar Reconstruction from a Single RGB Image”. In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE/CVF, 2018, pp. 2579–2588.

[37] Alice Lo Valvo, Daniele Croce, Domenico Garlisi, Fabrizio Giuliano, Laura Giarré, and Ilenia Tinnirello. “A Navigation and Augmented Reality System for Visually Impaired People”. In: *Sensors* 21.9 (2021), p. 3061.

[38] Andréa Macario Barros, Yoann Moline, Frédérick Carrel, Maugan Michel, and Gwenolé Corre. “A Comprehensive Survey of Visual SLAM Algorithms”. In: *Robotics* 11 (2022).

[39] Aminah Abdul Malek, Wan Eny Zarina Wan Abdul Rahman, Siti Salmah Yasiran, Abdul Kadir Jumaat, and Ummu Mardhiah Abdul Jalil. “Seed point selection for seed-based region growing in segmenting microcalcifications”. In: *International Conference on Statistics in Science, Business and Engineering (ICSSBE)*. IEEE, 2012, pp. 1–5.

[40] Hannes Mols, Kailai Li, and Uwe D. Hanebeck. “Highly Parallelizable Plane Extraction for Organized Point Clouds Using Spherical Convex Hulls”. In: *International Conference on Robotics and Automation (ICRA)*. IEEE, 2020, pp. 7920–7926.

[41] Raúl Mur-Artal and Juan D. Tardós. “ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo, and RGB-D Cameras”. In: *Transactions on Robotics* 33.5 (2017), p. 12551262.

[42] V. Nguyen, A. Martinelli, N. Tomatis, and R. Siegwart. “A comparison of line extraction algorithms using 2D laser rangefinder for indoor mobile robotics”. In: *International Conference on Intelligent Robots and Systems (IROS)*. IEEE/RSJ, 2005, pp. 1929–1934.

[43] Bastian Oehler, Joerg Stueckler, Jochen Welle, Dirk Schulz, and Sven Behnke. “Efficient Multi-resolution Plane Segmentation of 3D Point Clouds”. In: *Intelligent Robotics and Applications*. Vol. 7102. Springer Berlin Heidelberg, 2011, pp. 145–156.

[44] Martin Peternell and Tibor Steiner. “Reconstruction of piecewise planar objects from point clouds”. In: *Computer-Aided Design* 36.4 (2004), pp. 333–342.

[45] Pedro F Proença and Yang Gao. “Probabilistic RGB-D odometry based on points, lines and planes under depth uncertainty”. In: *Robotics and Autonomous Systems* 104 (2018), pp. 25–39.

[46] Pedro F. Proença and Yang Gao. “Fast Cylinder and Plane Extraction from Depth Cameras for Visual Odometry”. In: *International Conference on Intelligent Robots and Systems (IROS)* (2018), pp. 6813–6820.

[47] Xiangfei Qian and Cang Ye. “3D object recognition by geometric context and Gaussian-Mixture-Model-based plane classification”. In: *International Conference on Robotics and Automation (ICRA)*. IEEE, 2014, pp. 3910–3915.

[48] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. “U-net: Convolutional networks for biomedical image segmentation”. In: *International Conference on Medical image computing and computer-assisted intervention (MICCAI)*. Springer. 2015, pp. 234–241.

[49] Arindam Roychoudhury, Marceli Missura, and Maren Bennewitz. “Plane Segmentation Using Depth-Dependent Flood Fill”. In: *International Conference on Intelligent Robots and Systems (IROS)*. IEEE/RSJ, 2021, pp. 2210–2216.

[50] Arindam Roychoudhury, Marcell Missura, and Maren Bennewitz. “Plane Segmentation in Organized Point Clouds using Flood Fill”. In: *International Conference on Robotics and Automation (ICRA)*. IEEE, 2021, pp. 13532–13538.

[51] Radu Bogdan Rusu and Steve Cousins. “3D is here: Point Cloud Library (PCL)”. In: *International Conference on Robotics and Automation (ICRA)*. 2011.

[52] Alexander Schaefer, Johan Vertens, Daniel Büscher, and Wolfram Burgard. “A Maximum Likelihood Approach to Extract Finite Planes from 3-D Laser Scans”. In: *International Conference on Robotics and Automation (ICRA)*. IEEE/RSJ. 2019.

[53] Tobias Schwarze, Martin Lauer, Manuel Schwaab, Michailas Romanovas, Sandra Bohm, and Thomas Jurgensohn. “An Intuitive Mobility Aid for Visually Impaired People Based on Stereo Vision”. In: *International Conference on Computer Vision (ICCV)*. IEEE, 2015, pp. 409–417.

[54] Nathan Silberman, Derek Hoiem, Pushmeet Kohli, and Rob Fergus. “Indoor Segmentation and Support Inference from RGBD Images”. In: *Computer Vision – ECCV 2012*. Ed. by Andrew Fitzgibbon, Svetlana Lazebnik, Pietro Perona, Yoichi Sato, and Cordelia Schmid. Springer Berlin Heidelberg, 2012, pp. 746–760.

[55] Shuran Song, Samuel P. Lichtenberg, and Jianxiong Xiao. “SUN RGB-D: A RGB-D scene understanding benchmark suite”. In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 567–576.

[56] Sashank Sridhar and Sowmya Sanagavarapu. “Instant Tracking-Based Approach for Interior Décor Planning with Markerless AR”. In: *Zooming Innovation in Consumer Technologies Conference*. IEEE. 2020, pp. 103–108.

[57] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers. “A Benchmark for the Evaluation of RGB-D SLAM Systems”. In: *International Conference on Intelligent Robot and Systems (IROS)*. 2012.

[58] Bo Sun and Philippos Mordohai. “Oriented Point Sampling for Plane Detection in Unorganized Point Clouds”. In: arXiv:1905.02553 (2019). arXiv:1905.02553 [cs].

[59] Eduardo Vera, Djalma Lucio, Leandro A.F. Fernandes, and Luiz Velho. “Hough Transform for real-time plane detection in depth images”. In: *Pattern Recognition Letters* 103 (2018), pp. 8–15.

[60] Anh-Vu Vo, Linh Truong-Hong, Debra F. Laefer, and Michela Bertolotto. “Octree-based region growing for point cloud segmentation”. In: *Journal of Photogrammetry and Remote Sensing* 104 (2015), pp. 88–100.

[61] Xinlong Wang, Rufeng Zhang, Tao Kong, Lei Li, and Chunhua Shen. “Solov2: Dynamic and fast instance segmentation”. In: *Advances in Neural information processing systems* 33 (2020), pp. 17721–17732.

[62] Yi Wang, Haoyu Bu, Xiaolong Zhang, and Jia Cheng. “YPD-SLAM: A Real-Time VSLAM System for Handling Dynamic Indoor Environments”. In: *Sensors* 22.21 (2022), p. 8561.

[63] Yaxu Xie, Fangwen Shu, Jason R. Rambach, Alain Pagani, and Didier Stricker. “PlaneRecNet: Multi-Task Learning with Cross-Task Consistency for Piece-Wise Plane Detection and Reconstruction from a Single RGB Image”. In: *British Machine Vision Conference (BMVC)*. 2021.

[64] Jinxuan Xu, Qian Xie, Honghua Chen, and Jun Wang. “Real-Time Plane Detection with Consistency from Point Cloud Sequences”. In: *Sensors* 21.1 (2020), p. 140.

[65] Michael Ying Yang and Wolfgang Forstner. “Plane Detection in Point Cloud Data”. In: *Proceedings of the 2nd int conf on machine control guidance* 1 (2010), p. 16.

[66] Tian Yifei, Wei Song, Long Chen, Yunsick Sung, Jeonghoon Kwak, and Su Sun. “Fast Planar Detection System Using a GPU-Based 3D Hough Transform for LiDAR Point Clouds”. In: *Applied Sciences* 10 (2020), p. 1744.

[67] Hyun Yoo, Woo Kim, Jeong Woo Park, Won Lee, and Myung Chung. “Real-time plane detection based on depth map from Kinect”. In: 44. 2013, pp. 1–4.

[68] Fisher Yu, Vladlen Koltun, and Thomas Funkhouser. “Dilated residual networks”. In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2017, pp. 472–480.

[69] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. “Open3D: A Modern Library for 3D Data Processing”. In: (2018).

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.

Date: .....

(Signature)