Robust Efficient Mapping Applicable for a Multi RGBD-Camera Setup

Robuste Effiziente Kartierung anwendbar mit mehreren RGBD-Kameras
Bachelor thesis by Lars Pühler, Studienbereich Computational Engineering
Date of submission: October 14, 2021

1. Review: Prof. Dr. Oskar von Stryk
2. Review: Dr.-Ing. Alexander Stumpf

Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department

Fachgebiet Simulation, Systemoptimierung und Robotik

Robust Efficient Mapping Applicable for a Multi RGBD-Camera Setup
Robuste Effiziente Kartierung anwendbar mit mehreren RGBD-Kameras

Bachelor thesis by Lars Pühler, Studienbereich Computational Engineering

1. Review: Prof. Dr. Oskar von Stryk
2. Review: Dr.-Ing. Alexander Stumpf

Date of submission: October 14, 2021

Darmstadt

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 und §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Lars Pühler, Studienbereich Computational Engineering, die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß §23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 14. Oktober 2021

L. Pühler

Abstract

Autonomous robots got increasingly refined in the last decade, being able to plan their paths, having built-in obstacle avoidance or moving completely on their own. However, before an autonomous legged robot can perform high-level path and foothold planning, a representation of the environment is needed. In this thesis, a robust and efficient Mapping system is proposed to retrieve a consistent map representation using an RGBD-Camera Setup, which mainly will be used in the *L3 Terrain Model Generator*. The resulting elevation map uses a probabilistic approach to better cope with sensor noise. Each grid cell stores the height, the variance, the color, the estimated surface normal and the traversable area score. In this thesis, a new approach is proposed to estimate the surface normals, especially focusing on increasing the efficiency of the system.
A consistent map representation is especially challenging to achieve in a dynamic environment, as objects get detected, but do not get cleared as they move away. The proposed mapping approach considers dynamic objects using a ray-tracing based method to clear passed dynamic objects from the map.
The dense data produced by an RGBD-Camera makes real-time processing on the limited hardware of mobile robots difficult. In order to achieve an efficient mapping approach, this thesis uses massive parallelization. The main goal of the approach is an efficient and robust mapping system, which works highly paralleled on both the central processing unit (CPU) and the graphics processing unit (GPU).
The proposed approach is tested on both ground truth datasets and real-world experiments. The efficiency is verified using an Nvidia Jetson AGX Xavier to measure the computation time on both CPU and GPU. The evaluation shows using a GPU can significantly increase the efficiency of the system. Additionally, the evaluation shows that the probabilistic mapping approach proposed in this thesis is more robust against sensor noise than the previously used approach in the *L3 Terrain Model Generator*. Furthermore, the proposed surface normal estimation approach shows to be significantly more efficient than the previously used in the *L3 Terrain Model Generator*.

Zusammenfassung

Autonome Roboter wurden in den letzten zehn Jahren immer weiterentwickelt. Sie sind in der Lage ihre eigenen Pfade zu planen, Hindernisse zu vermeiden oder sich völlig selbstständig zu bewegen. Bevor jedoch ein autonomer Laufroboter solche Aufgaben durchführen kann, ist eine Darstellung der Umgebung erforderlich. In dieser Arbeit wird ein robustes und effizientes Kartierungssystem vorgeschlagen, um eine konsistente Kartendarstellung unter Verwendung eines RGBD-Kamera-Setups zu erhalten. Die resultierende Höhenkarte verwendet einen probabilistischen Ansatz, um das Sensorrauschen besser zu bewältigen. Jede Gitterzelle speichert die Höhe, die Varianz, die Farbe, die geschätzte Oberflächennormale und die die Bewertung der Passierbarkeit. In dieser Arbeit wird ein neuer Ansatz zur Schätzung der Oberflächennormalen vorgestellt, der sich insbesondere auf die Steigerung der Effizienz des Systems konzentriert.
Zusätzlich werden dynamische Objekte in dem Kartierungsansatz berücksichtigt. Durch die Verwendung einer auf Raytracing basierenden Methode zum Löschen vergangener, dynamischer Objekte wird eine konsistentere Kartierung in dynamischen Umgebungen erreicht.
Die enorme Datenmenge, die von einer RGBD-Kamera erzeugt wird, erschwert die Echtzeitverarbeitung auf der begrenzten Hardware eines mobilen Roboters. Um einen effizienten Mapping-Ansatz zu erreichen, verwendet diese Arbeit massive Parallelisierung. Hauptziel der Arbeit ist ein effizientes und robustes Mapping-System zu entwickeln, das sowohl auf der CPU als auch auf der GPU hochparallelisiert arbeitet.
Der vorgestellte Ansatz wird sowohl auf Basisdaten als auch in realen Experimenten getestet. Die Effizienz wird verifiziert, indem die verwendete Berechnungszeit auf CPU und GPU eines Nvidia Jetson AGX Xavier gemessen wird. Die Evaluierung zeigt, dass die Verwendung einer GPU die Effizienz des Systems erheblich steigern kann. Darüber hinaus hat die Auswertung gezeigt, dass der in dieser Arbeit vorgeschlagene probabilistische Mapping-Ansatz robuster gegenüber Sensorrauschen ist als der zuvor in der *L3 Terrain Model Generator* verwendete Ansatz. Darüber hinaus erwies sich der vorgeschlagene Ansatz zur Schätzung der Oberflächennormalen als wesentlich effizienter als der zuvor in der *L3 Terrain Model Generator* verwendete.

Contents

1. Introduction

Autonomous mobile robots need to overcome many different hurdles before they can navigate safely through any sort of terrain. Safe and reliable (loco)motion of mobile robots requires feasible motion plans, which in turn rely on an accurate environment representation. With sensors like laser scanners or cameras, the surrounding can be observed and captured. However, all the data from these sensors need to be processed and transferred into an efficient data representation for the robot to use. Especially in Simultaneous localization and Mapping (SLAM) the map representation is either a set of features or a point cloud, which are not very suitable for motion planning due to sparsity [25]. A dense map of the environment can be conducted using RGBD-Cameras but the processing of such dense data is often too computational intensive for the given hardware on mobile robots [26].

Motivation for this work is the rapid advance of capturing sensors over the last few years. However, processing data becomes more challenging, as the sensors become capable of producing even denser data. Due to the power and space constraints of mobile robots, there is often not enough computational power to perform all the processing in real-time using the robot's equipped computers [26]. To address this problem, cloud computing can be used. However, that is not feasible in communication constrained scenarios. These limitations anticipate an efficient and robust mapping framework that can work in real-time using the limited hardware on mobile robots.
3D LIDARs are commonly known for their long-range, large field of view (FOV), and highly accurate distance measurements. In contrast, RGBD-Cameras are usually used for near-field sensing up to 5 m with a limited FOV [31], but can provide highly dense 3D data as depth images [6]. With exception of Time-of-flight cameras, the depth image quality highly depends on the light conditions and the available surface texture. The use of such RGBD-Cameras challenges the handling of the relatively large sensor noise imposed by the stereo matching problem they have to solve.
As mobile robots also have use cases in a dynamic environment, for example in factory halls, with other vehicles, robots or humans moving around, the mapping approach needs to be adapted to tackle the challenges presented. The main challenge is that dynamic objects get captured into the map well, but never get cleared off again. This leaves back artifacts in the map representation, which lead to wrong information for other applications, for example obstacle avoidance or path planning.

Goal of this thesis is to provide an efficient and robust mapping framework, suitable for the challenging characteristics of RGBD-Cameras. The relative low cost of such cameras encourages using multiple sensors for a higher combined FOV, which requires very efficient mapping techniques to handle the vast amount of input data. In order to make the algorithms more computational efficient different techniques are used to highly parallelize the computation steps, which are efficiently processed by the graphics processing units (GPUs) using the CUDA environment from Nvidia [8]. For a meaningful use of the GPU, the code needs to be highly

parallelized to take full advantage of the high amount of cores given on a GPU. In addition, the challenges of dynamic environments are tackled, more specifically an approach using ray-tracing is proposed to clear off passed dynamic objects in the map.
The framework should be centered around the *Terrain Generator Framework* from the Legged Locomotion Library (L3). Additionally, the elevation map is built on the universal grid_map framework [15], providing an efficient data storing method. The new elevation map method should use a probabilistic approach to cope with sensor noise and uncertainty in the localization, based on the approaches from Fankhauser et al. [13] and Pan et al. [25]. The surface normals shall be computed using the approach from Fan et al. [12]. In order to process the data from the RGBD-Cameras, the Point Cloud Library (PCL) [29] can be used as a backend, as it provides 2D/3D image and point cloud processing. The main contributions of this thesis are:

- Adding a probabilistic elevation map representation to the *L3 Terrain Model Generator*.
- Providing a highly parallelizable approach to generate an elevation map from RGBD sensor data while maintaining the option to use either the central processing unit (CPU) or the graphics processing unit (GPU).
- Provide a ray-tracing based method to clear of passed dynamic objects from the map.
- Providing efficient and parallelizable methods to sufficiently retrieve the surface normal estimation on the CPU or GPU.

In order to evaluate the framework, the provided functions from the grid_map library [15] are used to generate ground truth map data. Additionally, a RealSense D455 and a RealSense T265 are used to provide real data for testing the framework.

The rest of the thesis is structured as follows. Chapter 2 explains the mathematical foundations and concepts used in this work, chapter 3 summarizes the literature already published in this academic field. In chapter 4, the basic concepts and methods that were developed and examined in this thesis are presented, while their concrete implementation is explained in chapter 5. An evaluation is presented in chapter 6 using ground truth data. Furthermore, a mock-up is created using the above mentioned RealSense cameras and an Nvidia Jetson AGX Xavier board to evaluate the system on real data, but without a real robot system. In the end, a conclusion and an outlook for future work are given in chapter 7.

2. Foundations

The following chapter gives an overlook and introduction to the mathematical formulations and concepts later used in the thesis.

2.1. Mahalanobis Distance

In order to determine the distance between objects, most commonly the Euclidean distance and the Mahalanobis distance is used, which both can be computed in the original variable space and in the principal component space [11]. In this thesis, the Mahalanobis distance is used for the detection of outliers.
As presented from De Maesschalck et al. [11], the Mahalanobis distance in the original space can be calculated as

$$MD_i^o = (x_i - \bar{x})\mathbf{C}_x^{-1}(x_i - \bar{x})^T,$$

where $\mathbf{C}_x$ is the variance-covariance matrix, which is computed as

$$\mathbf{C}_x = 1(n-1)(\mathbf{X}_c)^T(\mathbf{X}_c).$$

$\mathbf{X}$ is the data matrix containing n objects in the rows measured for p variables, $\mathbf{X}_c$ is the column-centred data matrix $(\mathbf{X} - \bar{\mathbf{X}})$ [11].
If all principal components are used, the Mahalanobis distance is equal in the original space as well as in the normalized or unnormalized principal component space [11].

2.2. Kalman Filters

The Kalman filter algorithm uses observed measurements over time to calculate estimates of some unknown variables [16]. These filters have a simple form and require only small computational power, which makes them suitable for various applications [16]. For example in this thesis the Kalman filter is used to update the height and variance in the probabilistic elevation map.
Being based on linear dynamical systems in state space formate, the Kalman filters estimates new states based on the previous states [16]. As stated from Govaers [16], the evolution of the state from time $k-1$ to time k is defined by the process model as

$$\mathbf{x}_k = \mathbf{F}\mathbf{x}_{k-1} + \mathbf{B}\mathbf{u}_{k-1} + \mathbf{w}_{k-1},$$

where $\mathbf{F}$ is the state transition matrix and $\mathbf{B}$ is the control-input matrix. $\mathbf{x}_{k-1}$ is the previous state vector, $\mathbf{u}_{k-1}$ is the control vector and $\mathbf{w}_{k-1} \sim \mathcal{N}(0, \mathbf{Q})$ is the process noise vector that is assumed to be zero-mean Gaussian with the covariance $\mathbf{Q}$ [16]. Additionally, the measurement model describing the relationship between the state and the measurement at the current time step k is defined from Govaers [16] as

$$\mathbf{z}_k = \mathbf{H}\mathbf{x}_k + \mathbf{v}_k$$

where $\mathbf{H}$ is the measurement matrix and $\mathbf{v}_k \sim \mathcal{N}(0, \mathbf{R})$ is the measurement noise vector that is assumed to be zero-mean Gaussian with the covariance $\mathbf{R}$ [16]. Given the initial state estimate x_0, a series of measurements, $z_1, z_2, ..., z_k$, and the system information described by $\mathbf{F}$, $\mathbf{B}$, $\mathbf{H}$, $\mathbf{Q}$ and $\mathbf{R}$ the Kalman filter can provide an estimation of x_k at time step k [16].
The Algorithm can be divided into a prediction stage and an update stage [16]. In the prediction step an estimation for the state $\hat{\mathbf{x}}_k^-$ and error covariance $\mathbf{P}_k^-$ is predicted as follows [16]

$$\begin{aligned}\hat{\mathbf{x}}_k^- &= \mathbf{F}\hat{\mathbf{x}}_{k-1}^+ + \mathbf{B}\mathbf{u}_{k-1},\\ \mathbf{P}_k^- &= \mathbf{F}\hat{\mathbf{x}}_{k-1}^+\mathbf{F}^T + \mathbf{Q}.\end{aligned}$$

After that prediction, the measurement residual $\tilde{\mathbf{y}}_k$ and the Kalman gain $\mathbf{K}_k$ can be computed through [16]

$$\begin{aligned}\tilde{\mathbf{y}}_k &= \mathbf{z}_k - \mathbf{H}\hat{\mathbf{x}}_k^-,\\ \mathbf{K}_k &= \mathbf{P}_k^-\mathbf{H}^T(\mathbf{R} + \mathbf{H}\mathbf{P}_k^-\mathbf{H}^T)^{-1}.\end{aligned}$$

Finally the updated state estimate $\hat{\mathbf{x}}_k^+$ and the updated error covariance $\mathbf{P}_k^+$ can be computed with [16]

$$\begin{aligned}\hat{\mathbf{x}}_k^+ &= \hat{\mathbf{x}}_k^- + \mathbf{K}_k\tilde{\mathbf{y}}_k,\\ \mathbf{P}_k^+ &= (\mathbf{I} - \mathbf{K}_k\mathbf{H})\mathbf{P}_k^-.\end{aligned}$$

3. Related Work

The idea of an internal map representation has already been widely researched in many different approaches in the past, for example in [13], [15], [25], [31]. Simultaneous mapping and Localisation (SLAM) is a highly researched topic among the robotics community, which combines the self-localisation of the robot with the map generation. There, both the localization and the map are updated and corrected from each other. Even though this thesis assumes that the robot has a functional self-localization, a possible integration into a SLAM project should not be overlooked.

This chapter is structured as follows. In the first section, an overview of different already existing open-source libraries commonly used for mapping is given. In the next section different mapping approaches are discussed, after which the topic of surface normal estimation is presented. Following, an overview over the topic of RGBD SLAM is presented and finally a comparison between LIDAR and RGBD-Camera sensors is given.

3.1. Existing Libraries

In order to implement the mapping, the sensor data needs to be processed, transformed and represented by an efficient data representation. To achieve the goal, this thesis uses different open-source libraries. This section gives a short explanation of suitable libraries and why they are or not used in this thesis. Namely Open3D [37], Point Cloud Library (PCL) [29], OpenCV [3], Voxblox [23] and grid_map [15] are presented.

Open3D [37] is a open-source library, with both algorithms in C++ and Python dealing with 3D data. The main three provided data structures are point clouds, meshes and RGB-D images. In addition also processing algorithms for I/O, sampling, data conversion and visualization are provided. Open3D has less dependencies than other big libraries, for example the Point Cloud Library. The few existing dependencies are all distributed as a part of Open3D, thus they can be compiled from source, which is especially useful on operating systems with no package manager, like Windows. In order to keep the library as simple as possible, new algorithms are only added if they solve a problem of broad interest and their implementation demonstrates significantly stronger results on a well-known benchmark.
For this thesis, mainly the representation of RGB-D images and the transformation of these into 3D points is of interest. However seeing that the Point Cloud Library [29] is already fully integrated into the L3 library, Open3D is not used in this thesis. This is because it shows no clear strength in the needed topics over the Point Cloud Library justifying the addition of an otherwise not useful additional library.

Point Cloud Library [29] is an open-source C++ library for 2D/3D image and point cloud processing, containing algorithms for segmentation, registration, model fitting, surface reconstruction, filtering and feature estimation. The Point Cloud Library, or short PCL, is fully integrated into the Robot Operating System (ROS) and is designed to handle point clouds efficiently. For good performance PCL provides support for Intels Threading Building Blocks (TBB) for multi-core parallelization. In addition the Fast Library for Approximate Nearest Neighbors (FLANN) provides the fast k-nearest neighbor search and all passed data in PCL is handled using Boost shared pointers.
In this thesis, PCL is mainly used for it's data representation of point clouds. As the L3 pipelines already work with PCL it is also chosen over the newer Open3D [37] library.

OpenCV [4], [24] is an open source computer vision library, written in C and C++ and having interfaces for Python, C++, Python, Java and MATLAB. Currently the library has over 2500+ algorithms, solving all sorts of problems in both computer vision and machine learning. These algorithms serve a wide range of applications, for example producing 3D point clouds from stereo cameras, tracing of moving objects and stitching images together. Main goals while devolving OpenCV were advancing the vision research by providing open and optimized code for basic vision infrastructure, providing a common infrastructure that developers could build on and advancing vision-based commercial application by making the optimized code available for free. Today OpenCV can be used one multiple different operating systems, including Windows, Linux, Android and Mac OS.
In this thesis, OpenCV is used for its data representation of depth and color images, as well as for its image processing algorithms. Additionally, the framework is used in the Three-Filters-To-Normals framework [12], which is later used to estimate surface normals in this thesis (see section 4.1). OpenCV is already used in the *L3 Terrain Model Generator* and therefore does not add any extra bindings to the library.

grid_map [15] is an universal open-source grid map library, especially designed to use as a mapping framework for mobile robotics. To provide the capability for different types of maps, grid_map uses multi-layered maps, where every layer can be used for a different data representation, such as the elevation map, occupancy of the grid cell or surface normals. In order to provide a wide range of data processing algorithms, the data storage is based on the linear algebra library Eigen [17]. In addition, other features, such as computationally efficient repositioning of the map boundaries and compatibility with existing Robot Operating System (ROS) [27] map message types are provided.
grid_map is already integrated into the *L3 Terrain Model Generator*. In this work the library is used in order to create and update a computationally efficient map representation, especially of the 2.5D elevation map with traversable area annotation. In addition, the thesis is based on the robot-centric elevation mapping framework form Fankhauser et al. [13], as further explained in 3.2.

Voxblox [23] is an open-source C++ mapping library, using Truncated Signed Distance Fields (TSDFs) in order to give a good environment representation for planning applications. In comparison with traditional TSDFs, the voxel size in Voxblox is increased not only to cope with the demand of real-time processing, but also because a small voxel size is not required in most planning applications. Furthermore, it implements an algorithm to construct Euclidean Signed Distance Fields (ESDFs) straight from TSDFs. Using the library, the addition of new voxels, layers and integrators is made easy. The integrators are single-threaded and run on a

CPU for prototyping, but they can easily be parallelized. The library on it is own has minimum requirements, however ROS bindings are provided for easier import and visualization of data. Real-time performance and reconstruction accuracy of the system is tested on both stereo and RGB-D imagery datasets.
Just like grid_map the Voxblox library is already integrated into L3. It is not used in this thesis, but it can be used in a future work in order to have a better representation of the map for obstacle avoidance, especially for trajectory generation.

The Benefits and Challenges of the above introduced open-source libraries are further illustrated in table 3.1, especially considering their use case for the proposed mapping approach with its main application in the *L3 Terrain Model Generator*.

Open Source Library	**Short description**	**Benefits for this thesis**	**Challenges for this thesis**	**Used in this thesis**
Open3D [37]	Library for 3D data processing, conversion and visualisation	Data representations for RGBD-Camera images, conversion of RGBD-Camera images into 3D point clouds	No clear advantage of already used libraries in the *L3 Terrain Model Generator*	No
PCL [29]	C++ library for point cloud processing	Fully integrated into the *L3 Terrain Model Generator*, data representation of point clouds	-	Yes
OpenCV [4], [24]	Library for computer vision, machine learning	Data representation for images, image filtering tools	-	Yes
grid_map [15]	Mapping framework for mobile robots	Data representation for grid-based elevation map, fully integrated into *L3 Terrain Model Generator*	Only CPU implementation	Yes
Voxblox [23]	Mapping library for planning applications	TSDF and ESDF map representation	Only CPU implementation	No

Table 3.1.: Comparison between the available open-source libraries considered for the proposed mapping approach.

3.2. Mapping

Many approaches use a 2.5D grid-based elevation map to represent the terrain, with each cell corresponding to 2D coordinates and storing the terrain height [13]. Clearly, this map representation has some limitations compared to a full 3D map representation. However, the elevation map allows far more efficient data access and processing due to its efficient structure and simple scalability, hence, making it more suitable for real-time mapping approaches [13].

The sensor data need to be fused into an efficient map representation in order to provide the useful information for further processing. The fusion step needs to be computational efficient in order to be as fast as possible. The mapping approach of this thesis is built on three main approaches:

- Elevation Map from Fankhauser et al. [13]
- GPU accelerated approach from Pan et al. [26]
- Local Dense Mapper from GEM proposed by Pan et al. [25]

Elevation Map [13] is a mapping approach for mobile legged robots with uncertain localization. In order to cope with uncertainty, a probabilistic terrain mapping approach is used. In addition, a robot-centric map representation is used instead of the classic world-centric representation, to further cope with the pose estimate drift probabilistically. In the robot-centric elevation map, currently by the sensor observed parts have the highest accuracy. Currently not observed terrain however accumulate more and more uncertainty over time due to the robots motion. The map is built on the grid_map library [15].
The system is partitioned into a map update and a map fusion step. In the update step the data are collected from the sensors and the map gets updated from the range sensor measurements as well as from the robot motion. Therefore, the uncertainty of the robot's motion is propagated to the map data. These two update methods provide a real-time elevation map. The size of the map is limited to the local environment to keep the computational cost low, as they are proportional to the map size. When the map is requested by other applications for further processing, the map fusion part is triggered. For each cell the surrounding data are considered in order to retrieve a consistent map representation with lower and upper confidence estimations. To adapt the map fast to dynamic environments, the map update process is extended with two parts. The first part adds a constant noise value, improving the map adaptation when the real terrain is lowered. The second part is a visibility check based on ray-tracing. As this step is very computationally expensive, it is only performed at a much lower rate.

The GPU Accelerated Approach from [26] is based on Elevation Map [13] and the work is extended by using a GPU to accelerate the system and adding a structure-based drivable area detection. With these additions, the system can create a dense map representing drivable region in real-time.
In a robot-centric elevation map, each step of the robot results in a movement of the map, which causes data at the edge of the map to be cleared and thus changes the position of the data in the map. However, mutual positional relationships between the data never change. Based on this, not the position of existing data is updated, but rather the positional shift values are updated according to the robot movement. These values can then be used to query data from all locations.
To update the map, the height of measuring points are considered as a Gaussian probability distribution retrieving the mean from the height of the measured point and the variance through the range sensor noise

model. Furthermore, the map needs to be updated every time the robot moves. Localization errors require an update of the variance of valid data. The estimated height and variance of the corresponding cell is updated using a Kalman filtering.
For calculating of drivable area, only easily calculated variables like surface normal and neighbor height deviation are used, because of the computational limits on mobile robots. A grid score is introduced for each cell of the grid denoting the probability of traversability. The grid score combines the evaluated scores of both the slop and the roughness of the terrain.
To cope with dynamic objects ray-tracing is used due to the principal that an area can only be observed with no other obstacles between the sensor and the area. So for every new sensor measurement it is checked whether another cell in the grid is blocking the light path. If that is the case, it is cleared. To reduce the computational effort only cells with a bad traversability need to be checked, as they are marked as a potential object.
The experimental results of Pan et al. [26] show that due to the adoption of the GPU, their approach produces competitive results in real-time, the results being significantly better than those of Elevation Map [13].

GEM [25] is the newest approach and combines SLAM with local consistent mapping. Therefore, the global map consists of local submaps, while the trajectory estimation is corrected through loop closing, the local maps are kept unchanged except for their relative poses. Being implemented for CPU or GPU coordinates processing, has the advantage of real-time mapping and in-time handling of dynamic obstacles. Especially interesting for this thesis is the local dense mapper building the dense robo-centric elevation map from the GEM system, which is implemented on the GPU while the global map and its corrections is handled by the CPU.
The local dense mapper consists out of three parts, namely the construction of the dense map, a traversable region detection and the dynamic object processing. Just like the earlier proposed approach from Pan et al. [26], GEM uses a selective updating mechanism and a GPU implementation in order to ensure real-time performance. To use the parallelization possibilities on the GPU, all steps of the map building are done in parallel. Firstly, the sensor measurements are transformed to the map frame, while also calculating their variance. Secondly, multiple measurements lying in the same grid get fused, checking the highest measurement against the most recently captured height in the grid. If a much higher elevation is measured the grid gets reinitialized else the new measurements get fused with the existing data of the grid. Thirdly, the traversability annotation is computed using both normal vectors to reflect the slope of the terrain and the height deviation to reflect the roughness of the terrain. Based on these two factors a traversability value is calculated. If this value is over a certain threshold that is based on the robots capabilities, the grid gets marked as traversable. Finally, dynamic objects get processed by applying ray-tracing but only evaluating grids that got marked as not traversable along the ray in order to keep the computational load lower.
The global mapper also consists of three parts: submap building, map deformation and submap maintenance, integrating the local map into an existing submap or creating a new submap. To avoid ever growing complexity whenever a completed submap has a large overlap with existing submaps, the new submap gets integrated into the other submaps. Using the corrected trajectory provided from SLAM, only the frames of the submaps are transformed to eliminate drift.

3.3. Surface Normal Estimation

Surface normal estimation is a well-studied field due to its theoretical appeal and its many practical applications [19]. One of the first approaches in the field of surface normal estimation from unorganized points was

presented by Hoppe et al. [18]. If the data is retrieved from a range image, assumptions about the estimated surface can be made because every data point is visible from the camera position so the surface orientation can be inferred from the viewing direction [18].
In general, Surface Normal Estimators (SNE) can be categorized into geometry-based and machine/deep learning-based approaches. Additionally, they can be classified by whether they use unstructured range data like an unorganized 3D point cloud or structured range date like a depth/disparity image or a organized 3D point cloud [12]. Using unstructured range data normally requires the generation of an undirected graph, for example a k-neares neighbor graph, which is computationally expensive to generate [12].

Geometric Approaches typically uses statistical analysis or optimization techniques in order to fit planar or curved surfaces to a local set of points [12]. According to Jordan et al. [19], the main methods that can be classified as optimization-based are PlanSVD, PlanePCA, VectorSDV and QuadSVD, where SVD stands for Singular Value Decomposition and PCA stands for Principal Component Analysis.
Jordan et al. [19] define the matrix of input points as $\mathbf{P} = [p_1, ..., p_n]^T$, neighbor matrix for each point as $\mathbf{Q}_i = [q_{i,1}, ..., q_{i,n}]$ and the augmented neighbor matrix $\mathbf{Q_i}^+$, which holds additionally the reference point $\mathbf{p}_i = [p_{i,x}, p_{i,y}, p_{i,z}]$. The main methods to retrieve the normal $\mathbf{n}_i = [n_{i,x}, n_{i,y}, n_{i,z}]$ can now be described as follows.
Minimizing the fitting error of the local plane in the neighborhood of the reference point, the objective function of PlaneSVD resolves to [19]

$$J_{PlaneSVD}(\mathbf{b}_i) = ||[\mathbf{Q_i^+} \, \mathbf{1_{k+1}}]\mathbf{b}_i||_2, \tag{3.1}$$

where $\mathbf{b}_i = [\mathbf{n}_i^T \; d]^T$.
Instead, PlanePCA computes the direction of the minimum variance in the neighborhood. If the points formed a perfect plane than this direction would be the normal to the plane, resulting in the objective function as [19]

$$J_{PlanePCA}(\mathbf{n}_i) = ||[\mathbf{Q_i^+} - \mathbf{\bar{Q}_i^+}]\mathbf{n}_i||_2. \tag{3.2}$$

The current SNE of the L3 library uses the PlanePCA implementation form the Point Cloud Library [29] to compute the normals of the input point cloud.
VectorSDV seeks the vector that maximizes the angle between itself and the vectors from the neighboring points to the reference point, which forces the estimated plane to go through the given reference point [19]. The object function is [19]

$$J_{VectorSVD}(\mathbf{n}_i) = || \begin{bmatrix} (\mathbf{q}_{i1} - p_i)^T \\ (\mathbf{q}_{i2} - p_i)^T \\ \vdots \\ (\mathbf{q}_{ik} - p_i)^T \end{bmatrix} \mathbf{n}_i||_2 \tag{3.3}$$

Finally, assuming that the surface is composed of small quadratic patches, QuadSVD computes the final normal by evaluating the gradient of the surface at the reference point. The objective function can be written as [19]

$$J_{QuadSVD}(\mathbf{c}_i) = ||\mathbf{R}_i\mathbf{c_i}||_2, \tag{3.4}$$

where $\mathbf{c}_i$ denotes the vector of the coefficients for the surface

$$\mathbf{S}_i(x,y,z) = c_1x^2 + c_2y^2 + c_3z^2 + c_4xy + c_5xz + c_6yz + c_7x + c_8y + c_9z + c_10. \tag{3.5}$$

The rows of $\mathbf{R}_i$ each contain the linear and quadratic terms corresponding to a point in $\mathbf{Q}_i^+$ [19].
The result of the comparison between these methods of Jordan et al. [19] shows that anchoring to the neighborhood mean has increased accuracy and is more robust against noise. Jordan et al. [19] conclude

that the use of PlanePCA with added vector normalization outperformed the other methods. Furthermore vector normalization significantly improves the numerical stability and accuracy of the total least squares (TLS) minimization [19].

Two other approaches to compute surface normals using linear least squares are presented by Badino et al. [1]. In order to minimize the computational effort, the traditional least square loss function was reformulated, such that a box-filtering method could be applied. Furthermore, experimental results showed that without a proper normalization, the traditional least squares does not perform accurately and is not robust against noise [1]. In order to address this problem, an appropriate scaling of the data matrix is necessary [1]. The unconstrained least squares and fast approximate least squares approach proposed by Badino et al. [1] shows the same accuracy as the normalized least squares, while being up to 17 times faster in their experiments.

Deep/Machine Learning Approaches got more and more attention from researchers due to the rapid advance in machine/deep learning. Many researchers resorted to deep convolution neural networks (DCNNs) in the last few years [12]. Especially when using RGBD data, the main problems are missing pixels due to glossy or transparent objects or the sensor noise along object edges, thus the quality of the normals can suffer from the corruption in depth [35].
Many approaches use either RGB data or depth data in order to construct the surface normals [35]. However, using only an RGB input it is difficult to extract the exact geometric information, especially struggling with insufficient or high lighting areas [35]. Using only the depth brings its own challenges, often the noise in depth will undermine the depth based normal estimations performance, also large holes in the depth can not be handled [35].
Zeng et al. [35] propose an approach using both the RGB and the depth informations provided by a RGB-D camera, therefore introducing a hierarchical structure to fuse data from RGB and depth. This results in a normal map with fine details due to use of RGB informations to fill the missing pixels in the depth [35]. Additionally, sharp edges get enhanced by merging the depth clue into the RGB results to enhance sharp edges and correct erroneous estimation [35].

Three-Filters-To-Normal (3F2N) is a novel approach proposed by Fan et al. [12], which is designed for structured range sensor data. 3F2N can compute surface normals using only three filters: namely a horizontal image gradient filter, a vertical image gradient filter and a mean/median filter [12].
In order to evaluate the approach, it was tested along with earlier approaches, for example PlaneSVD [19] and FALS [1] on three own datasets, in addition to the DIODE [33] and ScanNet [10] dataset. Furthermore, to measure the trade-off between accuracy and speed, a new metric

$$\pi = e_{\mathrm{A}} t \tag{3.6}$$

is introduced [12]. A low π score inducts a fast and precise surface normal estimator, with e_{A} being the average angular error and t the processing time [12]. Results also show, that when using the mean filter the system is faster in comparison to using the median filter [12]. Even though the median filter is more accurate, it is only advised to be used when high accuracy is required, because the mean filter still achieves a lower π Score [12]. In addition, results show that both filter approaches are more accurate than the previously mentioned FALS [1] and when using the mean filter also less processing time is needed [12]. This approach is especially interesting for this thesis due to its already given implementation for a GPU using the CUDA Framework.

3.4. RGBD-SLAM

In yet unknown terrain, a mobile robot does not only need to map its surrounding, but also needs self-localization in the surrounding. Simultaneous Localization and mapping (SLAM) was first proposed in the 1980s and tackles this exact problem [36]. With more and more emerging applications like mobile robot navigation or autonomous driving, SLAM has become a highly researched topic in the last ten years, with many approaches, especially differing due to the sensors used. Today most SLAM approaches use either LIDAR, RGBD-Cameras or a combination of both. This section particularly takes a look at different RGBD-SLAM approaches.

Visual SLAM traditionally used monocular or stereo cameras as input sensor using complex algorithms, which are easy to fail, in order to create a 3D map reconstruction [36]. However, in the last few years with the emerge of the RGBD-Camera the visual SLAM approach changed, as now depth information is provided the sensor and do not need to be computed with complex algorithms [36].
The main three parts of RGBD-Slam are camera tracking, mapping and loop detection [36]. After years of research, the pipeline of visual SLAM (vSLAM) is nowadays basically fixed [36]. In most modern vSLAM systems, the tasks of local mapping and tracking are completed in separated threads, because they are designed using the concept PTAM [20] [36]. PTAM splits the tracking and mapping into two separate tasks, which can then be processed in two parallel threads [20]. The multi-thread vSLAM systems can be divided into frontend and backend [36]. The frontend, which is also called RGBD Odometry, provides real-time camera poses, while the backend updates and optimizes the map [36].
Historically, direct or indirect methods have been used for camera tracking. Nowadays, a combination of both is used [36]. While the indirect method minimizes the geometric error over a set of matching artificial features in two frames, the direct method tries to estimate the camera motion based on minimizing the photometric error over corresponding pixels [36]. In order to combine the advantages of the two different methods, the hybrid approaches minimize a combination of the two errors to align two frames [36].

BAD SLAM from Schops et al. [30] stands out, because mainly point-based or volumetric methods are used for mapping in SLAM [36], however BAD SLAM uses surfels instead. The main advantage of the surfels is their ability to represent thinner surfaces, while the minimal surface thickness is limited to the voxel size in grid-based representations [30]. These surfels are later only optimized along their normal direction, using only measurements with similar normals to the current surfel [30]. Before the algorithm runs, the images provided by the RGBD-Camera get preprocessed with a bilateral filter to smooth out the depth image and remove outlier depth measurements [30].

3.5. LIDAR and RGBD-Camera

The last section of the chapter gives an overview and a comparison between 3D LIDAR and RGBD-Camera. The benefits and challenges can be seen in table 3.2.

Modern 3D LIDAR and RGBD-Camera systems do not differ much in their output, most of them provide a point cloud. However, the output point clouds may vary due to RGBD-Cameras providing color information

Topic	3D LIDAR	RGBD-Camera
Output	Point cloud Intensity	Point cloud RGB-/Depth-Image
Benefits	uses less computational power for mapping [21] large field of view [6]	more robust to vibration (no moving parts) [6] rich in texture -> better classification [6] dense and more complete informations [6]
Challanges	Sparse representation of scene flow [6] No colored points [6]	small effective depth range [31] Limited field of view [6] Real-Time challenging for mapping [6]
Cost	3000-12000 € [28]	200-300 € [28]

Table 3.2.: Comparison between LIDAR and RGBD-Camera.

for every point where the LIDAR can only provide the intensity.
Bigger differences between 3D LIDAR and RGBD-Cameras are notable in the field of view, the maximum range, the density and the prices. RGBD-Camera have a clear advantage regarding the prize, costing only a fraction of the price of a typical LIDAR system [6].
Additionally, the scene understanding is made both easier and more robust due to the greater amount of information provided by the camera [6]. The 3D LIDAR on the other hand has a much higher range capability. Furthermore, modern LIDAR systems often can provide a 360° field of view again exceeding the capability of a RGBD-Camera, which can only provide a narrow field of view of typically 70° [6]. It should be noted that the field of view can be increased by fusing multiple cameras together [6]. In addition, the actual usable field of view is mainly determined by the sensor position on the robot [6]. Due to this the large field of view of the 3D LIDAR can often not be fully used.
All these factors lead to the desire to create an efficient and robust mapping algorithm using multiple RGBD-Cameras. However, the great amount of information provided by such an RGBD-Camera setup makes real-time mapping very challenging, whereas 3D LIDAR requires less computational power in order to map the sensor data [21].

4. Method

This chapter explains the proposed method to efficiently generate a map representation using dense data from RGBD-Cameras. To achieve an efficient framework it needs to be highly parallelized. Furthermore, a probabilistic map representation is chosen to cope with the noise of the sensor data. To improve the map representation in dynamic scenes a ray-tracing based approach is proposed to clear passed dynamic objects from the map. Furthermore, two approaches to estimate the surface normals are presented. The resulting normals are used to compute the traversable area of the map for a given robot.
A quick graphical overview of the two possible update method are illustrated in figures 4.1 and 4.2, differing only in the used approach to estimate the surface normals. Both approaches are later evaluated. To update the map the variance of each point in the point cloud is computed. After the new height and the variance get fused into the map, the traversable area is computed and ray-tracing is performed to clear dynamic obstacles from the map representation.
The following chapter is structured as follows. First, the method for acquiring the surface normals is explained. Then, a foundation of the definitions used for the map representation is given. Afterwards, the other main parts of the update method are addressed, namely the computation of the variance at every point, the update of the height and variance in the map, the classification of the traversable area and the handling of dynamic objects. Finally, an overview of the whole system is given using pseudo code.

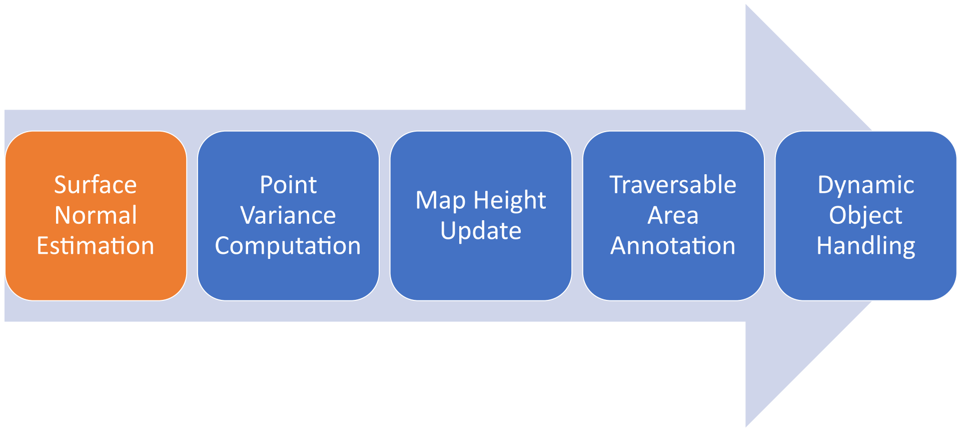

Surface Normal Estimation
Point Variance Computation
Map Height Update
Traversable Area Annotation
Dynamic Object Handling

Figure 4.1.: System overview with normals estimation from the input cloud.

4.1. Surface Normal Estimation

Currently, the surface normals in the *L3 Terrain Model Generator* are estimated using PlanePCA from a unstructured point cloud. This work uses the 3F2N approach from Fan et al. [12] that takes advantage of

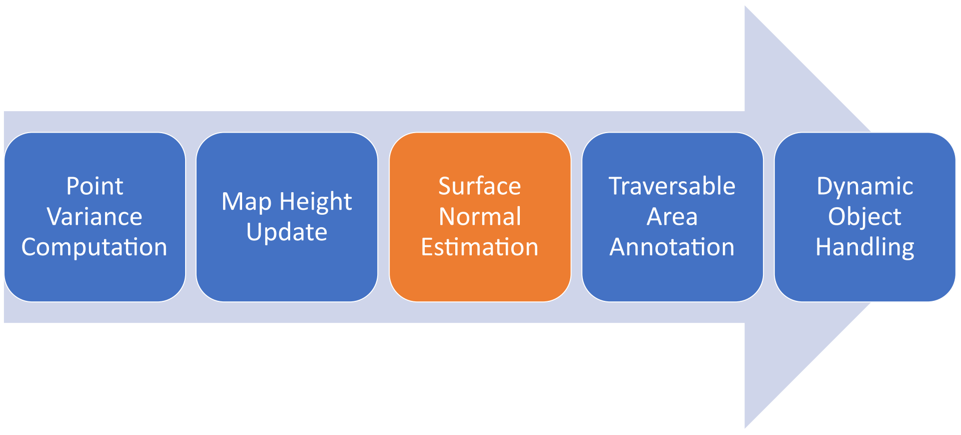

Point Variance Computation
Map Height Update
Surface Normal Estimation
Traversable Area Annotation
Dynamic Object Handling

Figure 4.2.: System overview with normals estimation from the map.

depth images and structured point clouds provided by RGBD-Cameras. This approach is explained in the following section. The 3F2N approach applies three filters on a depth or disparity image to retrieve the surface normals, namely a horizontal image gradient, a vertical gradient and a mean or median filter [12].
The surface normal $\mathbf{n}_i = [n_x, n_y, n_z]^T$ of an observed 3D point $\mathbf{p}_i = [x, y, z]^T$ in the camera coordinate system is estimated by fitting the local plane

$$n_x x + n_y y + n_z z + b = 0 \tag{4.1}$$

to the points in $\mathbf{Q}_i^+ = [\mathbf{Q}_i^T, \mathbf{p}_i]^T$. $\mathbf{Q}_i = [\mathbf{q}_{i1}, ..., \mathbf{q}_{ik}]$ is a set of k neighboring points of $\mathbf{p}_i$.
In order to transform a 3D point $\mathbf{p}_i$ in the camera coordinate system into the point $\tilde{\mathbf{p}}_i = [u, v]^T$, according to Fan et al. [12] the following equation can be used

$$z \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \mathbf{K}\mathbf{p}_i = \begin{bmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \tag{4.2}$$

where $\mathbf{K}$ denotes the camera intrinsic matrix and $\mathbf{p}_0 = [u_0, v_0]^T$ is the image principal point. f_x denotes the camera focal length in pixel in x direction and f_y in y direction, respectively [12]. As stated from Fan et al. [12], combining 4.1 and 4.2 leads to

$$\frac{1}{z} = -\frac{1}{b}\left(n_x \frac{u - u_0}{f_x} + n_y \frac{v - v_0}{f_y} + n_z\right). \tag{4.3}$$

Differentiating the resulting equation with respect to u and v results in [12]

$$\begin{aligned} \frac{\partial 1/z}{\partial u} &= -\frac{n_x}{b f_x}, \\ \frac{\partial 1/z}{\partial v} &= -\frac{n_y}{b f_y}. \end{aligned} \tag{4.4}$$

By applying horizontal and vertical image gradient filters on the inverse depth image, for example Prewitt, Scharr or Sobel, the shown derivations in (4.4) can be approximated [12]. An inverse depth image stores the values of $1/z$. The expressions for n_x and n_y can then be found by rearranging 4.4 to [12]

$$\begin{aligned} n_x &= -bf_x \frac{\partial 1/z}{\partial u}, \\ n_y &= -bf_y \frac{\partial 1/z}{\partial v}. \end{aligned} \tag{4.5}$$

The corresponding n_{zj} can now be computed with a given arbitrary $\mathbf{q}_{ij} \in \mathbf{Q}_i$ by combining 4.1 and 4.5 to

$$n_{zj} = b \frac{f_x \Delta x_{ij} \frac{\partial 1/z}{\partial u} + f_y \Delta y_{ij} \frac{\partial 1/z}{\partial v}}{\Delta z_{ij}}, \tag{4.6}$$

where $[\mathbf{x_{ij}}, \mathbf{y_{ij}}, \mathbf{z_{ij}}] = \mathbf{q}_i - \mathbf{p}_i$ [12]. Simplifying 4.5 and 4.6, because they have a common factor of $-b$, leads to the following equations

$$\begin{aligned} n_x &= f_x \frac{\partial 1/z}{\partial u}, \\ n_y &= f_y \frac{\partial 1/z}{\partial v}, \\ and n_z &= -\Phi \left\{ \frac{\Delta x_{iy} n_x + f_y \Delta y_{iy} n_y}{\Delta z_{ij}} \right\}, j = 1, ..., k, \end{aligned} \tag{4.7}$$

where $\Phi\{\cdot\}$ denotes the manner of $\hat{n}_z$ estimation, which is the optimum n_z [12]. Additionally, the mean or median filtering operation used to estimate n_z is represented as $\Phi\{\cdot\}$[12]. If the depth value of $\mathbf{p}_i$ is identical to those of all its neighbors, the surface normal n_i is simply set to $[0, 0, -1]^T$ as its direction is considered to be perpendicular to the image plane [12].
The approach from Fan et al. [12] can be further extended to be able to compute the surface normal of any given organized point cloud. In this thesis the organized point cloud of the RGBD-Camera is used to estimate the normals. To use the 3F2N approach the organized cloud needs to be in the camera frame and the camera intrinsic matrix needs to be known.
However, this can be extended to also be able to compute the normals from a given 2.5D map representation. In order to do so, the identity matrix is passed as the camera matrix. With these adaptations, the normals can be computed from the map representation, as the data resembles an organized point cloud.
This approach is chosen to be tested under the assumption that the map works like a filter while fusing in the new data. Therefore, the resulting normals should be less affected from the noise of the input cloud.

4.2. Definitions for the Elevation Map

Before the further update scheme can be explained, a general definition of all used identifiers needs to be established first, which is based on [13], [25].
First, the used coordinate systems are defined as inertial frame **I**, the robot base frame **B**, the sensor frames $\mathbf{S_i}$ and the map frame **M** [13]. This differs from [13], because multi sensors could be used. The rotation and translation between the robot frame **B** and the sensor frames $\mathbf{S_i}$ are assumed to be known as they should be clear from the robot kinematics, being both attached to the base and the sensors [13]. In the following, the fixed transformation between the sensors and base is referred to as $(\mathbf{r}_{BS_i}, \Phi_{BS_i})$ [13]. Additionally, the transformation $(\mathbf{r}_{IB}, \Phi_{IB})$ between the robot base frame **B** and the inertial frame **I** are obtained through the

pose estimation, which also provides the pose covariance matrix $\Sigma_{IB} = cov(\mathbf{r}_{IB}, \Phi_{IB})$ [13]. As stated from Fankhauser et al. [13], the rotation between the inertial and base frame Φ_{IB} can be split as

$$\Phi_{IB} = \Phi_{I\tilde{B}}(\psi) \circ \Phi_{\tilde{B}B}(\theta, \varphi), \tag{4.8}$$

with $\Phi_{I\tilde{B}}(\psi)$ describing the rotation with the yaw angle ψ around the vertical axis $\mathbf{e}_z^I$ for the frame $\tilde{\mathbf{B}}$. $\Phi_{\tilde{B}B}(\theta, \varphi)$ describes the tilt rotation about the pitch and roll angle θ and φ between $\tilde{\mathbf{B}}$ and $\mathbf{B}$ [13].
Finally, the elevation map is defined as a 2.5D probabilistic elevation grid map, where for each cell in the grid multiple entries are stored. In a probabilistic map, the height measurement $\tilde{p}$ at each cell is approximated by a Gaussian probability distribution as $\tilde{p} \sim \mathcal{N}(p, \sigma_p^2)$, with p being the mean and σ_p^2 the variance [13].
Since in this thesis RGBD-Cameras are used, the RGB value c_{rgb} is stored for each cell additionally. Also, each cell has a traversable value t stating whether this cell should be traversable by the robot or not. The map frame $\mathbf{M}$ in this thesis is defined as the inertial frame $\mathbf{I}$, differing from [13], because not a robot-centric map representation is chosen.

4.3. Variance Computation

To create a probabilistic elevation map, the variance of every point needs to be computed. This thesis builds on [13] in order to retrieve the variance of the sensor data.
The measurement p can be computed from a single measurement ${}_S r_{SP}$ captured in the sensor frame $\mathbf{S}_i$ using

$$p = \mathbf{P}(\Phi_{SM}^{-1}({}_S r_{SP}) - {}_M r_{SM}), \tag{4.9}$$

with $\mathbf{P} = [\mathbf{0}, \mathbf{0}, \mathbf{1}]$ being the projection matrix to retrieve the scalar height measurement p from the three-dimensional measurement in the map frame $\mathbf{M}$ [13]. In order to obtain the variance of the measurement σ_p^2 as suggested by Fankhauser et al. [13], first the Jacobian matrices for the sensor measurement $\mathbf{J}_S$ and the sensor frame rotation $\mathbf{J}_\Phi$ need to be calculated as

$$\mathbf{J}_S = \frac{\partial p}{\partial_S r_{SP}} = \mathbf{PC}(\Phi_{SM})^T, \tag{4.10}$$

$$\mathbf{J}_\Phi = \frac{\partial p}{\partial \Phi_{SM}} = \mathbf{PC}(\Phi_{SM})^T \; {}_S r_{SP}^{\times}. \tag{4.11}$$

$C(\Phi)$ describes the mapping of the corresponding rotation matrix, which Bloesch et al. [2] define as $\mathbf{C} : SO(3) \to \mathbb{R}^{3\times3}$, such that $\Phi(\mathbf{r}) \widehat{=} \mathbf{C}(\Phi) r$ with $\Phi \in SO(3)$. Based on that, the error propagation for the variance σ_p^2 can be retrieved as

$$\sigma_p^2 = \mathbf{J}_S \Sigma_S \mathbf{J}_S^T + \mathbf{J}_\Phi \Sigma_{\Phi_{IS}} J_\Phi^T, \tag{4.12}$$

with Σ_S being the covariance matrix of the sensor model [13]. Using the sensor noise models denoted from Fankhauser et al. [14], the values for Σ can be obtained. $\Sigma_{\Phi_{SM}}$, which is a submatrix of Σ_{IS}, resembles the covariance matrix of the sensor rotation [13]. The uncertainty of the yaw-rotation around the axis e_z^I of the sensor does not need to be taken into account, because it gets excluded from the measurement with the use of the projection matrix $\mathbf{P}$ [13].
This method can be efficiently parallelized, as the variance of every point can be computed independently.

4.4. Height Update

After the variance for all points in the point cloud is computed, the data need to be transferred into the map representation. Therefore, again the approaches from Fankhauser et al. [13] and Pan et al. [25] are used and explained in this section.
The new height measurement is denoted as $(\tilde{p}, \sigma_p^2)$ and gets fused into the map with the existing elevation estimation $(\hat{h}, \sigma_h^2)$ [13]. Using parallelization, the grid map is updated for every cell, thus it is possible that multiple points are laying in the same grid cell. Like in the approach from Pan et al. [25], the χ^2 test is applied to evaluate the difference Δ between the highest most recent measured elevation $\tilde{p}$ and the corresponding grid elevation $\hat{h}$ with variance σ_h

$$\Delta = \frac{|\hat{h} - \tilde{p}|}{\sigma_h}. \tag{4.13}$$

In case a higher elevation is measured and Δ not passing the χ^2 test, the grid cell is reinitialized with the elevation $\tilde{p}$ and the corresponding variance σ_p^2 [25]. Then all elevations passing the test are fused into the grid cell by means of a one dimensional Kalman filter as stated in [13] as

$$\begin{aligned} \hat{h}^+ &= \frac{\sigma_p^2 \hat{h}^- + \sigma_h^2 \tilde{p}}{\sigma_p^2 + \sigma_h^{2-}}, \\ \sigma_h^{2+} &= \frac{\sigma_h^{2-} \sigma_p^2}{\sigma_h^{2-} + \sigma_p^2}. \end{aligned} \tag{4.14}$$

Estimates before the update are denoted by an added - superscript and estimates after the update have an added + superscript [13]. The χ^2 test is used to consider dynamic objects and vertical obstacles [25]. The color of the grid cell is updated from every point falling into the grid cell, leading to the color of the grid cell being represented by the color of the last point falling into the grid cell.
In order for the method to be efficiently paralleled, the update can be done independently concerning every grid cell.

4.5. Traversable Area

After the height of the grid cells is updated, the traversable area is noted for all grid cells. A score t_{free} is being calculated according to the equation given by Pan et al. [25]

$$t_{free} = max\left(0, 1 - w_s \frac{v_{slope}}{v_{s,crit}} - w_r \frac{v_{rough}}{v_{r,crit}}\right), \; t_{free} \in [0,1], \tag{4.15}$$

where v_{slope} denotes the value of the slope at the given grid cell using the estimated surface normals, and v_{rough} denotes the value of the roughness based on height deviation [25]. $v_{s,crit}$ and $v_{r,crit}$ are constants based on the capability of the robot [25]. They represent the threshold between the grid cell being traversable or not [25]. Additionally the weight factors w_s and w_r are the weight factors for slope and roughness, respectively [25]. A score t_{free} of one indicates that an area is easy accessible, whereas a score of zero indicates a none traversable are (e.g. an obstacle) [26].
The surface normals were already acquired using 3F2N [12]. In order to now obtain the value v_{slope}, the

cosinus of the angle α between the surface normal and the z-axis $\mathbf{e}_z = [0, 0, 1]$ is evaluated. The formulation of the dot product between two vectors $\mathbf{v}_1$ and $\mathbf{v}_2$, is given as

$$\mathbf{v}_1 \cdot \mathbf{v}_2 = ||\mathbf{v}_1|| \cdot ||\mathbf{v}_2|| \cdot \cos(\alpha). \tag{4.16}$$

Since the vector $\mathbf{e}_z$ is normalized, the norm $||\mathbf{e}_z||$ evaluates to one, so all that needs to be done to obtain $\cos(\alpha)$ directly from the dot product is to normalize the computed normal vector. When the surface normal vector and the z-axis are equal, the dot product now evaluates to one which denotes no slope. The final value v_{slope} is now obtained through

$$v_{slope} = 1 - \cos(\alpha) = \mathbf{n}_{normalized} \cdot \mathbf{e}_z. \tag{4.17}$$

Finally, the deviation of the height H_d is computed as presented from [26]

$$H_d = |h_{x,y} - \bar{h}|, \tag{4.18}$$

where $h_{x,y}$ is the height of the current grid cell and $\bar{h}$ is the average height of all cells in a 3×3 window centered on the current grid cell [26]. H_d directly translates as v_{rough}.
This can be done in parallel with respect to every grid cell.

4.6. Dynamic Object Handling

While fusing the point cloud data into the map, the grid cells between the new measurement and the sensor are not updated. Obstacles get captured very well, but, in contrast to dynamic objects, they are not cleared from the map if they move away.
This thesis tackles the problem using a ray-tracing based method to determine passed dynamic objects. The concept of ray-tracing assumes that no obstacle can be located on the direct ray between the sensor and a captured endpoint [25]. Otherwise, the endpoint could not have been captured by the sensor. This thesis uses a 3D Bresenham line drawing algorithm for ray-tracing. The original 2D algorithm was already proposed by Bresenham [5] in 1965. The algorithm is especially fast due to not using any floating-point-numbers in its computation, but only relying on integer operations [7]. Today, the original approach got extended by many different people, for example, L. Chiang [7] and Basma et al. [34], in order to cope with 3D data. This new approach can be used for ray-tracing in a 3D voxel grid. The used 3D Bresenham algorithm is illustrated as a flow chart in figure 4.3, which is based on the presented pseudo code from [7].
The presented 3D Bresenham works on a 3D voxel grid, the map representation in this thesis however is not a full 3D representation. In order to still be able to use the algorithm, the data from the map gets transformed into a pseudo-3D representation.
For the ray-tracing purpose of the 3D algorithm in this thesis, each line resembles a ray between the sensor position and an endpoint in the input point cloud. The height of each grid cell is stored as a floating-point-number. The algorithm however only computes with integer values, thus the information needs to be transformed into a 3D voxel $v = [x_{3D}, y_{3D}, z_{3D}]$ with an integer index. The height h is transformed using the same grid resolution res_{grid} as in the 2D grid. The corresponding voxel index z_{3D} is computed as

$$z_{3D} = \lfloor h/res_{grid} \rfloor \tag{4.19}$$

As seen in figure 4.3, for one ray, only two voxels are needed. In this case, the start voxel V_1 describes the sensor position and the end voxel V_2 describes one of the points from the input point cloud. Whether grid cell g is a passed dynamic object or not is checked by performing ray-tracing for all points in the input cloud. Every

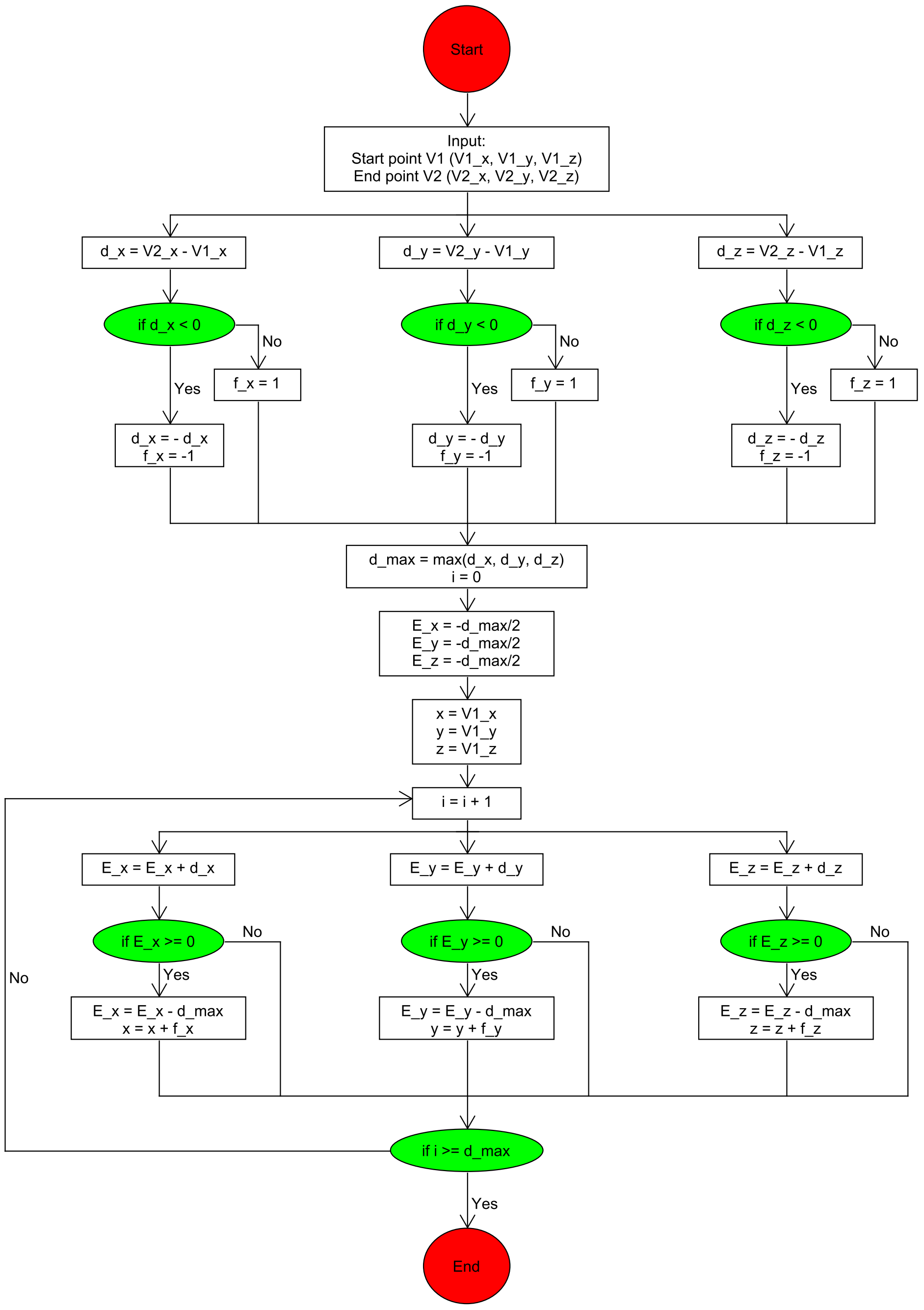


Figure 4.3.: Illustration of the 3D Bresenham algorithm based on the pseudo code by Chiang [7].

time the algorithm evaluates a new voxel of the ray, the indices x_{3D} and y_{3D} are compared with the x_g and y_g indices of the grid cell g. If both are equal, the height of the voxel is compared to the height stored in the grid cell g. If the height of the voxel is lower, this cell is handled as a passed obstacle and hence, gets reinitialized. Vertical walls are challenging for the proposed approach, as due to sensor noise and grid resolution often points of a vertical wall can fall into neighboring cells. Usually, the surface ends up with a width of more than one voxel. Using the algorithm explained above, the points in the row furthest away from the sensor of

the vertical surface could clear the voxels in the closer rows. Additionally, neighboring cells on horizontal surfaces could clear of each other, as the transformation into 3D voxel is only an approximation. This leads to unwanted holes in the map. To address both problems the algorithm stops a few voxels early and never reaches the created voxel for the endpoint. With this adaptation, the surfaces stay intact, at all times. However, if the algorithm stops too early also small artifacts of passed dynamic objects are not cleared off anymore.
Based on [25], only cells being noted as obstacles get checked to reduce the computational cost due to ray casting being a very computational expansive operation. The cells get determined as obstacles with a threshold and the above explained traversable score. The ray-tracing can be done in parallel with regard to every grid cell that needs to be checked.

4.7. System Overview

Concluding this chapter, a full overview of the whole system is given. The two algorithms 1 and 2 show the two proposed update cycles as pseudo-code. Algorithm 1 shows the approach of computing the normals directly from the input point cloud of the sensor, whereas algorithm 2 shows the approach of computing the normals from the updated map representation.
Given a grid map, the robot obstacle parameters and the sensors, both algorithms estimate the surface normals, compute the point variance, fuse the measurement into the grid cell, compute the traversable area and handle dynamic obstacles.
Differing in the method of the surface computation, as they either are computed first as it can be seen in 1 or after the map got update with the new sensor data as it can be seen in 2. However, the rest of the update method is the same and is structured as follows:

1. The variance for each point in the point cloud is computed.

2. The height and variance for every grid cell in the elevation map is updated using the points and variances calculated earlier.

3. The traversable score for each grid cell in the map is updated using the previously computed surface normals.

4. Dynamic objects are handled. Therefore all grid cells, which are currently marked as obstacles get checked using ray-tracing. The system uses a 3D version of the Bresenham algorithm [7] for ray-tracing. If a grid cell is ruled as a passed dynamic object, it is reinitialized.

Additionally, another configuration of the system is tested, from that the surface normals do not get estimated directly from the input cloud of the sensor. Rather, they get computed from the resulting grid map, after the input points got fused into it. Therefore, the intrinsic camera matrix just need to set to be the identity matrix, so the 3F2N [12] method can work on the 3D data provided by the grid map. This approach is chosen to be tested under the assumption, that the map works like a filter, while fusing in the new data. Therefore, the resulting normals should be less affected form the noise of the input cloud.

Algorithm 1 Update Method (Normals from input cloud)

```
Require: $S$ : sensors , $GM$ : grid map , $RP$ : robot parameters
1: for each $s_i$ in $S$ do
2:
3:     // surface normal estimation
4:     $P_i \leftarrow s_i$.getPointCloud()
5:     $N_i \leftarrow$ 3F2N($P_i$, $s_i$.getIntrinsicMatrix())
6:     transformToMapFrame($N_i$, $s_i$.getPose())
7:     fuseNormalsIntoMap($GM$, $N_i$, $P_i$)
8:
9:     // point variance computation
10:    for in parallel for each $p_j$ in $P_i$ do
11:        $\sigma_j^2 \leftarrow$ getVariance($p_j$,$s_i$.getPose())
12:    end for
13:
14:    transformToMapFrame($P_i$,$s_i$.getPose())
15:
16:    // height update
17:    for in parallel for each $g_k$ in $GM$ do
18:        for each $p_j$ in $P_i$ do
19:            if $p_j$ lies in $g_k$ then
20:                fusePointInMap($GM$, $p_j$, $\sigma_j^2$)
21:            end if
22:        end for
23:    end for
24:
25:    // traversable area computation
26:    for in parallel for each $g_k$ in $GM$ do
27:        $v_{k,slope} \leftarrow$ computeSlope($g_k$, $RP$)
28:        $v_{k,roughness} \leftarrow$ computeRoughness($g_k$, $RP$)
29:        $g_k$.addTraversableScore(
           computeTraversableScore($v_{k,slope}$, $v_{k,slope}$,
           $RP$))
30:    end for
31:
32:    // process dynamic objects
33:    for in parallel for each $g_k$ in $GM$ do
34:        if $GM$.getTraversableScore < Threshold then
35:            rayTracingBresenham($g_k$, $GM$, $P_i$, $s_i$.get-
               Pose())
36:        end if
37:    end for
38: end for
```

Algorithm 2 Update Method (Normals from map)

```
Require: $S$ : sensors , $GM$ : grid_map , $RP$ : robot parameters
1: for each $s_i$ in $S$ do
2:
3:     $P_i \leftarrow s_i$.getPointCloud()
4:
5:     // point variance computation
6:     for in parallel for each $p_j$ in $P_i$ do
7:         $\sigma_j^2 \leftarrow$ getVariance($p_j$,$s_i$.getPose())
8:     end for
9:
10:    transformToMapFrame($P_i$,$s_i$.getPose())
11:
12:    // height update
13:    for in parallel for each $g_k$ in $GM$ do
14:        for each $p_j$ in $P_i$ do
15:            if $p_j$ lies in $g_k$ then
16:                fusePointInMap($GM$, $p_j$, $\sigma_j^2$)
17:            end if
18:        end for
19:    end for
20:
21:    // surface normal estimation
22:    $N_i \leftarrow$ 3F2N($GM$.asPointCloud(), IdentityMatrix)
23:    fuseNormalsIntoMap($GM$, $N_i$)
24:
25:    // traversable area computation
26:    for in parallel for each $g_k$ in $GM$ do
27:        $v_{k,slope} \leftarrow$ computeSlope($g_k$, $RP$)
28:        $v_{k,roughness} \leftarrow$ computeRoughness($g_k$, $RP$)
29:        $GM$.addTraversableScore(
           computeTraversableScore($v_{k,slope}$, $v_{k,slope}$,
           $RP$))
30:    end for
31:
32:    // process dynamic objects
33:    for in parallel for each $g_k$ in $GM$ do
34:        if $GM$.getTraversableScore < Threshold then
35:            rayTracingBresenham($g_k$, $GM$, $P_i$, $s_i$.get-
               Pose())
36:        end if
37:    end for
38: end for
```

5. Implementation

In order to achieve a real-time mapping from RGBD-Camera data, the framework needs to be very efficient. In order to do so, the algorithm relies on parallelization. Especially the GPU code needs to be strictly optimized towards high parallelization, to capitalize on the high core count. Also on the CPU parallelization can bring a significant performance boost. In order to achieve parallelism, the code is oriented to the Single Instruction, multiple Data (SIMD) approach. In addition, the whole framework should work either with the use of a GPU or without, so all computation steps should be available as GPU and CPU code. So that with loading in one parameter the framework can adjust from just relying on the CPU to also using the GPU.
This chapter first gives an overview of the process chain of the L3 library. Then, the newly developed classes are presented. Afterwards, the implementation of the CPU and GPU code is outlined.

5.1. L3 Overview

An overview is illustrated in figure 5.1. As can be seen, the library mainly consists of sensor, generator and publisher plugins. Additionally, process chains can be triggered periodically at a fixed rate. The sensor plugins resemble a sensor, for example a 3d LIDAR or a RGBD-Camera. They capture the sensor data from any source, such as ROS messages and trigger a process chain forwarding the input data. A process chain is composed of subsequently called process plugins, such as generators, publishers or filters. Generally, process plugins produce a certain out, based on the input data given. The main purpose of generators is to produce new data or fuse data into an existing data field such as a grid map. Exemplary, the surface normal plugin produces a point cloud representing the surface normals and adds them into the grid map as a separate layer. The filters serve as chained filtering step while usually not altering the data type. Publisher plugins take the acquired data and publish them using ROS messages.

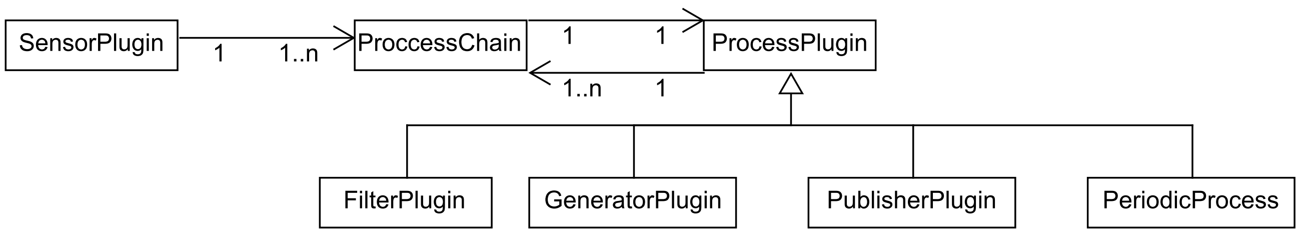
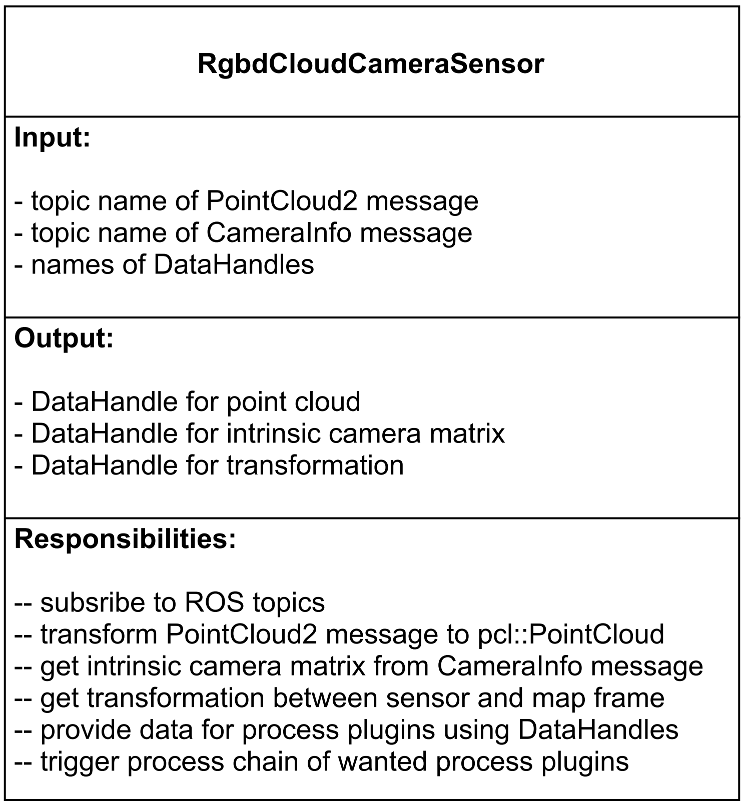


Figure 5.1.: Illustration of the structure of the *L3 Terrain Model Generator*.

5.2. Implemented Classes

In this thesis, four classes got added to the L3 library, namely three generator plugins and one sensor plugin. These plugins are outlined in the following.

The `RgbdCloudCameraSensor` Plugin provides the data from an RGBD-Camera to the generators in the framework. Additionally, it triggers the process chain of the needed generator plugins to fuse the data into the map. A brief description of the class is provided in figure 5.2.
The plugin collects the data through subscribing to the ROS topics provided by the RGBD-Camera, in particular it subscribes to the colored point cloud and the camera info topic. The point cloud and the intrinsic camera matrix are then added to the data handler. Additionally, it provides the transformation between the sensor frame and the map frame, so the generator plugins can access the data.



RgbdCloudCameraSensor

Input:

- topic name of PointCloud2 message
- topic name of CameraInfo message
- names of DataHandles

Output:

- DataHandle for point cloud
- DataHandle for intrinsic camera matrix
- DataHandle for transformation

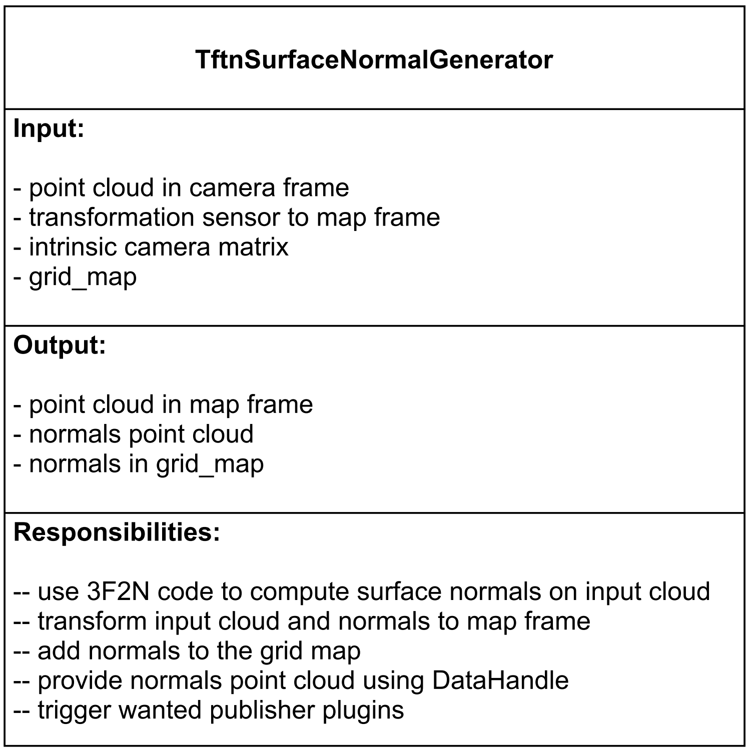
Responsibilities:

-- subsribe to ROS topics
-- transform PointCloud2 message to pcl::PointCloud
-- get intrinsic camera matrix from CameraInfo message
-- get transformation between sensor and map frame
-- provide data for process plugins using DataHandles
-- trigger process chain of wanted process plugins

Figure 5.2.: Illustration of the RgbdCloudCameraSensor plugin.

The `TftnSurfaceNormalGenerator` Plugin implements the method explained in chapter 4.1 to estimate the surface normals from the input point cloud. A brief overview of the class is provided in figure 5.3.

As seen in 5.3, input is the RGB point cloud provided by the `RgbdCloudCameraSensor`, the transformation between sensor and map frame and the intrinsic camera matrix, which is used by 3F2N [12] to compute the surface normals. As 3F2N computes the normals in the camera frame, both the resulting point cloud and normals need to be transformed into the map frame using the provided functions from PCL [29]. The output of the class are the surface normals and the transformed point cloud to the map frame. Additionally, the surface normals also get directly added into the map representation.



TftnSurfaceNormalGenerator

Input:

- point cloud in camera frame
- transformation sensor to map frame
- intrinsic camera matrix
- grid_map

Output:

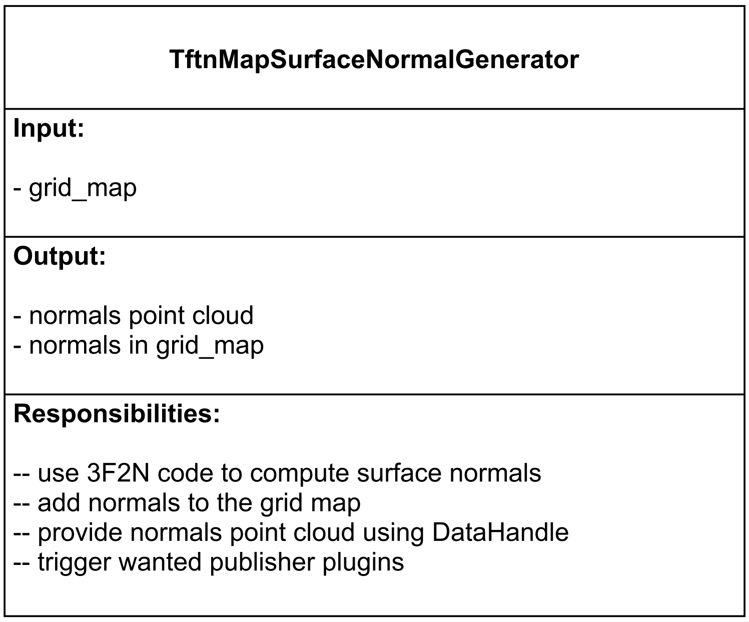
- point cloud in map frame
- normals point cloud
- normals in grid_map

Responsibilities:

-- use 3F2N code to compute surface normals on input cloud
-- transform input cloud and normals to map frame
-- add normals to the grid map
-- provide normals point cloud using DataHandle
-- trigger wanted publisher plugins

Figure 5.3.: Illustration of the TftnSurfaceNormalGenerator plugin.

The `TftnMapSurfaceNormalGenerator` Plugin also implements the method explained in chapter 4.1 to estimate the surface normals. The main difference to the previously introduced class is that now the surface normals are not directly estimated from the input point cloud of the sensor, but using the provided 3D data in the map. Therefore, no transformation and no intrinsic camera matrix are required, as the intrinsic camera matrix becomes an identity matrix. The estimated normals then are directly fused in to the map. A brief overview of the class is provided in figure 5.4.



TftnMapSurfaceNormalGenerator

Input:

- grid_map

Output:

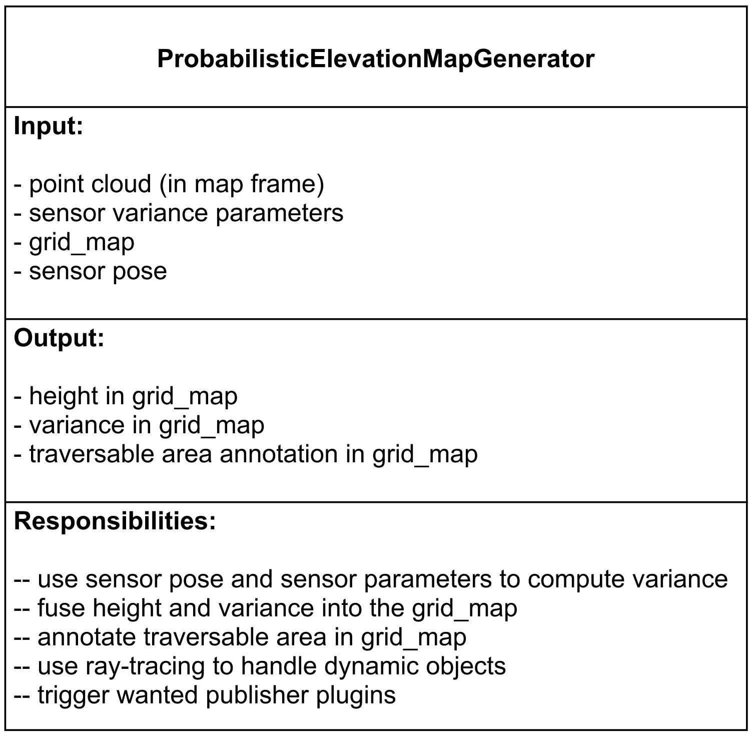
- normals point cloud
- normals in grid_map

Responsibilities:

-- use 3F2N code to compute surface normals
-- add normals to the grid map
-- provide normals point cloud using DataHandle
-- trigger wanted publisher plugins

Figure 5.4.: Illustration of the TftnMapSurfaceNormalGenerator plugin.

The `ProbabilsiticElevationMapGenerator` Plugin implements the remaining part of the update method discussed in chapter 4. A brief overview of the class is provided in figure 5.5.
The exact implementation of the class differs between the GPU and the CPU implementation, but the core functionality stays the same. The input is a point cloud optionally providing color data which is already transformed into the map frame, as well as the map itself. Using the provided data in the point cloud the variance, the height and the traversable area are computed in this class. Afterwards, dynamic objects are handled using the 3D Bresenham algorithm from section 4.6 for ray-tracing. The map, with all the computed annotations fused in to it, is output of this class.



ProbabilisticElevationMapGenerator

Input:

- point cloud (in map frame)
- sensor variance parameters
- grid_map
- sensor pose

Output:

- height in grid_map
- variance in grid_map
- traversable area annotation in grid_map

Responsibilities:

-- use sensor pose and sensor parameters to compute variance
-- fuse height and variance into the grid_map
-- annotate traversable area in grid_map
-- use ray-tracing to handle dynamic objects
-- trigger wanted publisher plugins

Figure 5.5.: Illustration of the ProbabilisticElevationMapGenerator plugin.

5.3. CPU Implementation

Modern CPUs are usually multi core units that can execute instructions in parallel. In order to take advantage of such CPUs, software must use parallelization techniques to support multi-threading. To achieve the parallelization, in this thesis the multi-threading framework already existing in the *L3 footstep planing library* is used.
The structure of the multi-threading framework is shown in figure 5.6. With initialization the `Threading Manager` assigns a `Work` to every specified thread. While in progress, the `ThreadingManager` takes the created job and hands them over to the `Queue`, which internally stores all jobs in a normal queue structure. Afterwards, the jobs are handed over to the `Worker`, which got spawned by the `ThreadingManger` when initializing the framework. For the initialization two parameters are read in, determining how many workers are spawned and how many jobs should be assigned to every worker. After the `Worker` got the jobs handed over from the `Queue`, it runs the `run()` method of these jobs, which means completing them. After completing all jobs, it checks back with the `Queue` in order to fetch new jobs if there are still jobs queued. In order to handle concurrency, threadsafe mutexes are used so data cannot be accessed concurrently.
The main framework now only needs to initialize the multi-threading framework with the required parameters, create and hand over these jobs to the `ThreadingManager` to get them processed. It needs to be checked whether every job was completed successfully afterwards. Important to note is that for every job type a new `ThreadManager` with additional `Queue` and `Worker` needs to be configured.

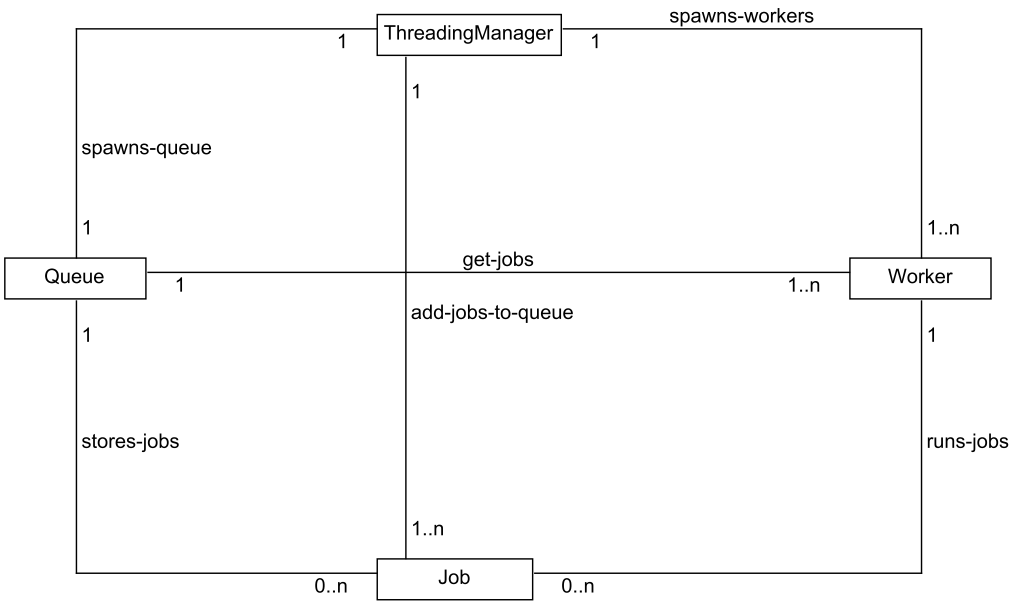


Figure 5.6.: Class diagram of the threading framework used in this thesis.

The Variance is mainly computed in the `PointVarianceJob`. However, commonly used matrices are pre-computed reducing computational overhead. For every point in the point cloud a job is created, where the variance is computed according to the method described in section 4.3. The resulting variance is stored in a class variable of the generator so it can later be used to update the map.

The Height Update is performed paralleled for each grid cell for which new data is provided by the input point cloud. Therefore, first all grid cells with new points need to be determined. This step is also very important to ensure a thread-save computation, as otherwise the data of a grid cell could change while another job is using it. Afterwards, a `HeightUpdateJob` is generated for each grill cell. In the job itself, the computation is performed as explained in 4.4. Because every job works directly on the grid map, no data transfer needs to be done afterwards.

The Traversable Area gets updated in parallel for every grid cell by a `TraversableJob` after the height and variance in the grid map was updated. The computation of the traversable score is implemented as described in section 4.5. Again, no data is transferred as the jobs directly work on the grid map and store the new score also directly there. The generator just checks whether all jobs were completed successfully.

Dynamic Object Handling is done through ray-tracing in parallel for all grid cells. It should be noted that ray-tracing is only applied, if a grid cell is annotated as an obstacle using the computed traversable score to reduce the computational load. In a rectangular submap, which includes all cells between the sensor position and the furthest point from the sensor in the point cloud, a `RayCastingJob` is created for each grid cell flagged as an obstacle. The job uses the implementation of a 3D Bresenham algorithm discussed in section 4.6 to check along the rays between the sensor and every point in the point cloud. If a ray crosses the grid cell and its height is lower than the height stored in the grill cell, the grid cell is reinitialized by setting the elevation layer to not a number (`NaN`).

5.4. GPU Implementation

Running highly parallelized programs using Nvidia GPUs a dedicated toolchain named CUDA [8] is required. It provides a C++ API to program for the Nvidia GPUs.
The GPU however can not directly access the memory of the computer, where the map is stored. In order to make this data available to the GPU it needs to be transferred to the GPUs memory. This requires the map and point data to be stored in additional arrays, which then are transferred to the GPUs memory. After the computation the data on the GPU memory also needs to be transferred back to be put back in to the map representation. As coping data from computer memory to GPU memory is a time expensive operation, only the needed submap, which includes the sensor position and all points of the input cloud, is transferred. Because the grid_map functions are not available for the GPU, all data are transferred as linear arrays. The data copied to the GPU include the stored elevation, variance, traversable score and surface normals of the submap, as well as the indices of the grid cells. Additionally, the 3D coordinates, the color value and the corresponding grid cell are transferred for each input point.
Four different CUDA kernels are implemented, each being a representation of the above explained job types on the GPU. These kernels can be launched to update the map once all the data is accessible from the GPU.
After all the computation is done on the GPU, the data need to be transferred back to the computer memory. For that, new arrays were proactively pre-allocated and their pointers were given to the GPU to transfer the data directly. After all data are back, the CPU now just needs to read the linear memory blocks and put the

stored data back into the map representation.
On the GPU itself, the data are all stored in arrays, then the data gets processed in parallel using the kernels. For each job explained in the CPU section, a corresponding kernel exists for the GPU implementation performing the identical job.

6. Evaluation

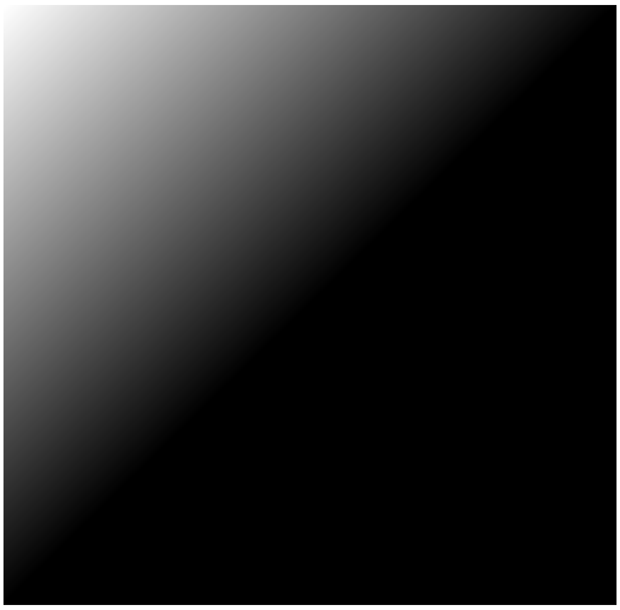
In the following chapter, the evaluation of the framework is presented. The main goal of the evaluation is to show that the system works with both ground truth and real data. In order to achieve this goal, the framework is tested using ground truth data produced from the implemented functions from the grid_map library [15] and using real data produced by a mock-up, which consists of a D455 Realsense depth camera [9] and a T265 Realsense tracking camera [32]. Additionally, the whole system is tested on an Nvidia Jetson AGX Xavier developer kit [22].
The chapter is structured as follows. First, the evaluation with ground truth data is presented. Second, the results of using real data from the mock-up are shown. Third, an overview of the runtime of the framework on both the CPU and the GPU is given. Fourth, the handling of dynamic objects is evaluated. Fifth, the new proposed surface normal estimation approaches are tested and compared, including a comparison with the previously used method in the *L3 Terrain Model Generator*. Lastly, the results are discussed.

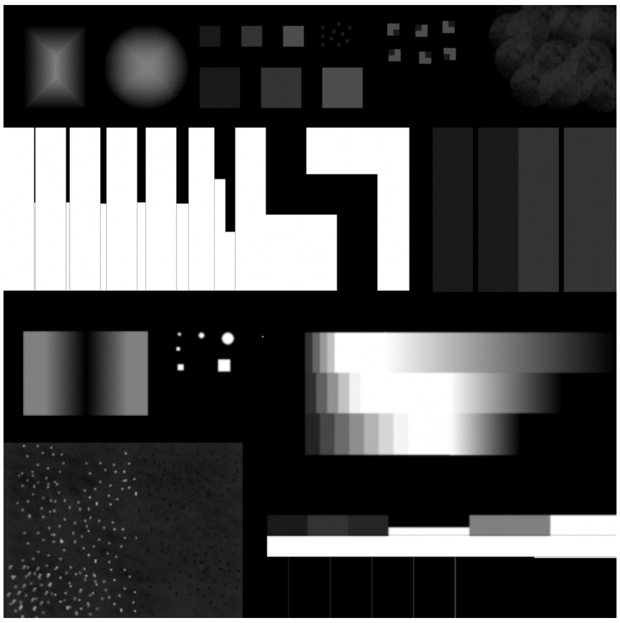
6.1. Ground Truth

To ensure and evaluate the capabilities of the framework, it is tested with ground truth data first before using real data.

6.1.1. Ground Truth Datasets

To create the ground truth data set, the function to create a grid map directly from a mono-colored image provided by the grid_map framework [15] is used. Given a maximum and a minimum height, the function translates every pixel as one grid cell. The height is created by linear mapping the color channel between the specified minimum and maximum height. To further create an input cloud for the framework, the resulting grid_map is transformed to 3D points. Every grid cell is resembled by one 3D point.
Three main scenarios are used to evaluate the ground truth behaviour of the framework. Two of which are provided from the grid_map library [15], namely a test terrain and a flat surface with an attached even slope. The mono-colored pictures used to create the maps can be seen in figure 6.1. Additionally, an image resembling a stair is used and can be seen in 6.1c. The test terrain differs from the two other scenes, because it already represents a non-perfect map. Noise is added to the map to better represent real terrain.
For all scenes, the maximum and minimum height is set accordingly, so the resulting data points lay between 0 m and 1 m. Additionally, the resolution of the maps is set to 2 cm.


(a) flat surface with even slope dataset


(b) test terrain from grid_map [15]

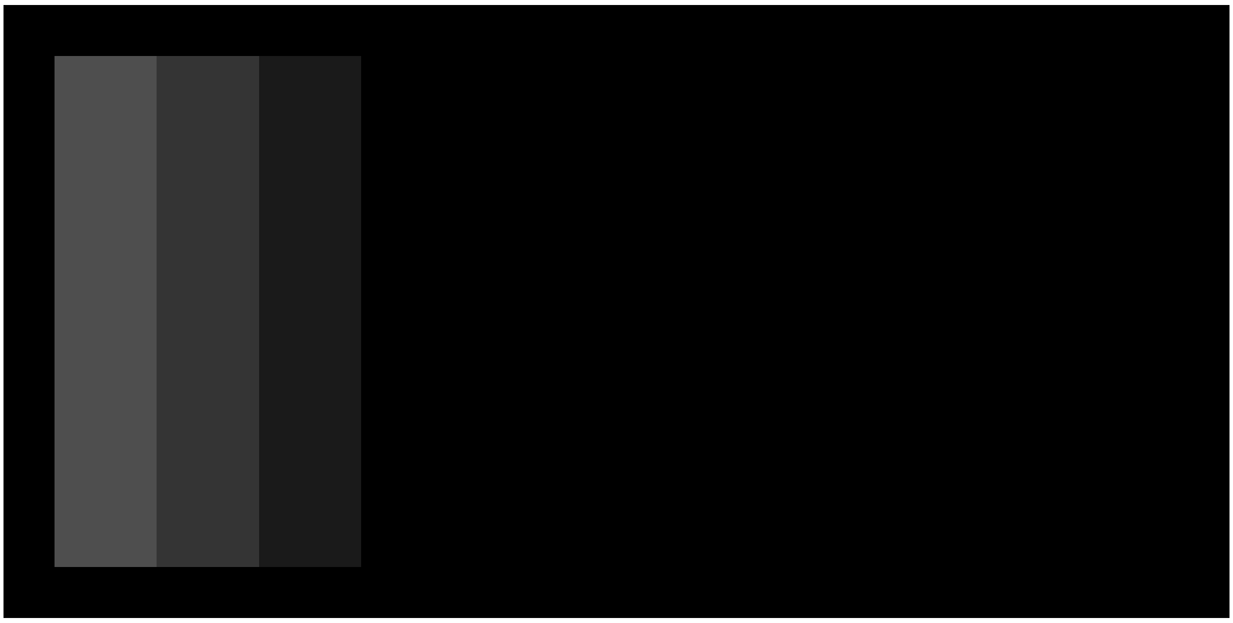
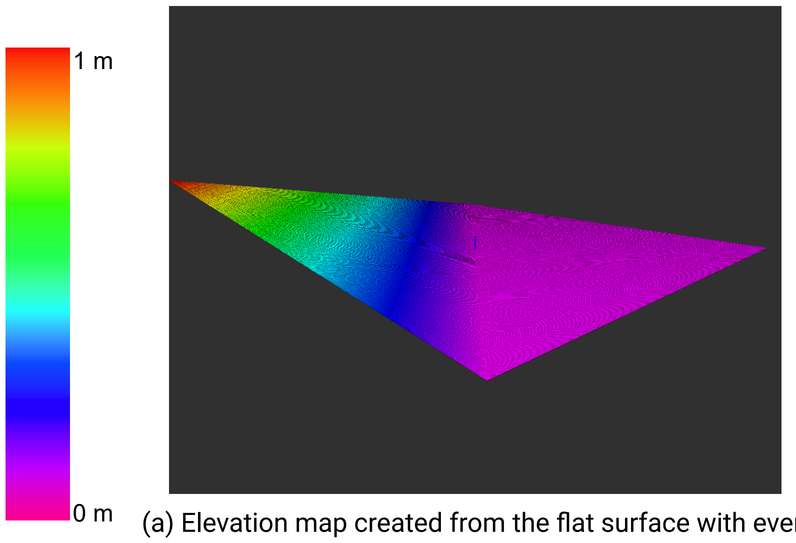

(c) stairs dataset

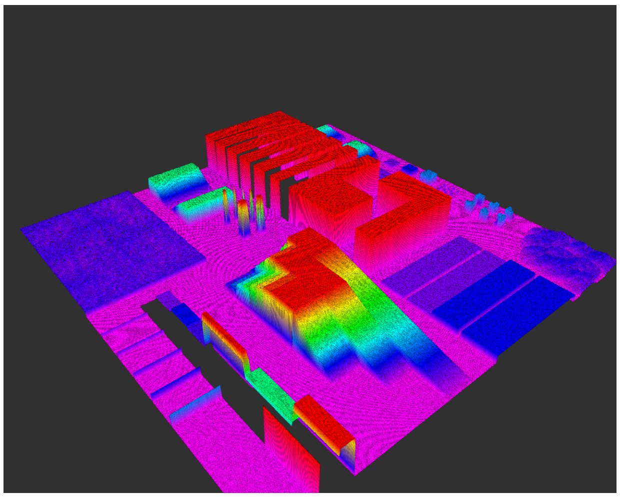
Figure 6.1.: The mono-colored images used to create the ground truth point cloud data.

6.1.2. Map Representation

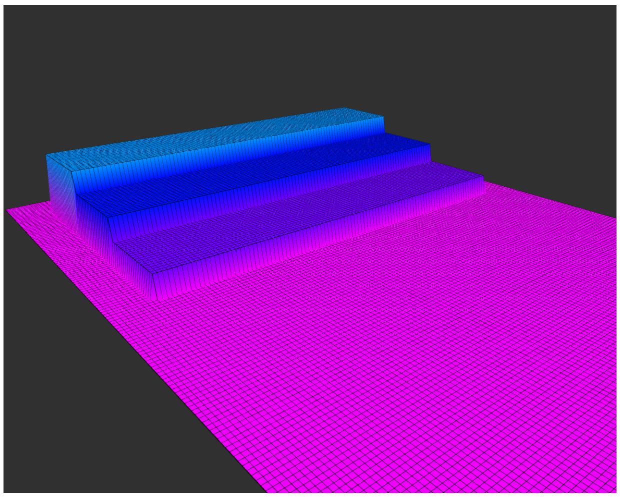
The capability of the framework to produce a map representation from an input cloud is tested using the ground truth datasets explained above. The resulting map representations can be seen in figure 6.2. Results show that given a perfect ground truth point cloud, without any sensor noise and position error from the robot pose, the framework can create a perfect representation of the given point cloud.



(a) Elevation map created from the flat surface with even slope dataset



(b) Elevation map created from the terrain dataset provided by grid_map [15]

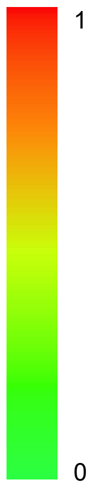


(c) Elevation map created from the stairs datasets

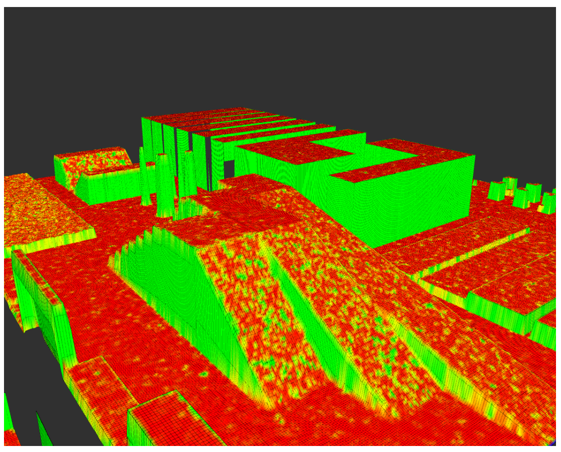
Figure 6.2.: Resulting elevation maps from the framework using ground truth point clouds created from the mono-colored pictures in 6.1.

6.1.3. Traversable Score

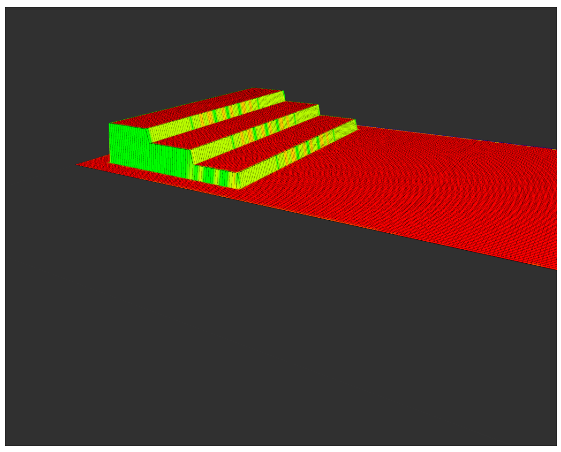
Additionally, also the traversable score computation can be tested using the provided ground truth data. As explained in chapter 4.5, to compute the traversable score, four parameters need to be specified. The parameters should resemble the capabilities of the robot, so they differ from robot to robot. In order to compute the traversable score shown in figure 6.3, the parameters got specified as: $w_s = 0.4$ (weight of slope), $v_{s,crit} = 0.3$ (slope threshold), $w_r = 0.6$ (weight of roughness) and $v_{r,crit} = 0.05$ (roughness threshold).
The results show that the traversable score gets computed as expected. Large edges get denoted with a traversable score of one, meaning an untraversable area, which are colored in green in figure 6.3. Even surfaces are colored in red and have a traversable score of zero, meaning an easily traversable area. Also, it can be seen that the noise and imperfections of the test terrain get resembled in the traversable score. To translate the traversable score into an occupancy grid, only a threshold function needs to be used, with the threshold again resembling the capabilities of the used robot.



(a) color scale



(b) traversable score of test terrain from grid_map [15]



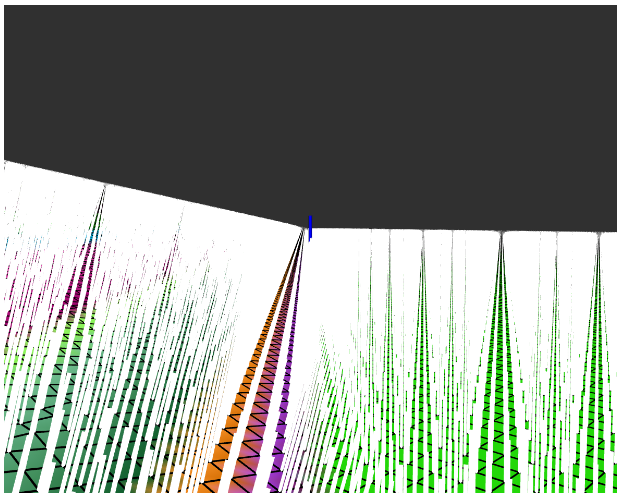
(c) traversable score of ground truth stairs dataset

Figure 6.3.: Traversable score on the provided terrain from the grid_map library [15] and on the stairs dataset. A score of one describes a traversable area, whereas a score of zero denotes an untraversable area.

6.1.4. Surface Normals

The surface normal estimator is tested using the ground truth data. The resulting normals of the slope and the stairs are shown in figure 6.4. These surface normals are computed using the explained method in 4.1. As no RGBD-Camera is used to produce the input cloud, the camera matrix is set to be the identity matrix.
In figure 6.4a, it can be seen that even slopes are represented perfectly by the normals. However, in figure 6.4b, the normals at the edges of the stairs do not look as expected. The expected normals should be pointing away from the edge, but most of them point along the edges. This however should not be a problem, as the slope score in the traversable area is computed only from the angle between the z-axis and the normal. As this angle is nearly the same for both the expected and the real normals, it should not affect the computation of the traversable score significantly.

The surface normal estimation suffers greatly from noise. This can be especially seen in figure 6.5. As the surface normals are computed using the neighboring cells, their computation is especially challenging in rough terrain.


(a) Ground truth slope map with normals computed from the map representation.

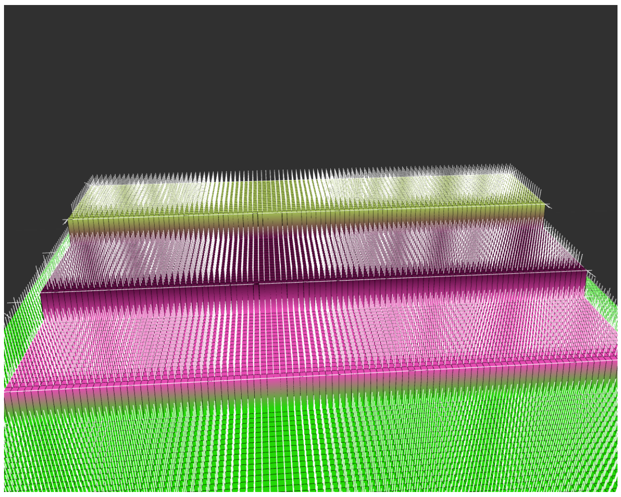

(b) Ground truth stairs map with normals computed from the map representation.

Figure 6.4.: Resulting maps with estimated surface normals from the framework using ground truth point clouds.

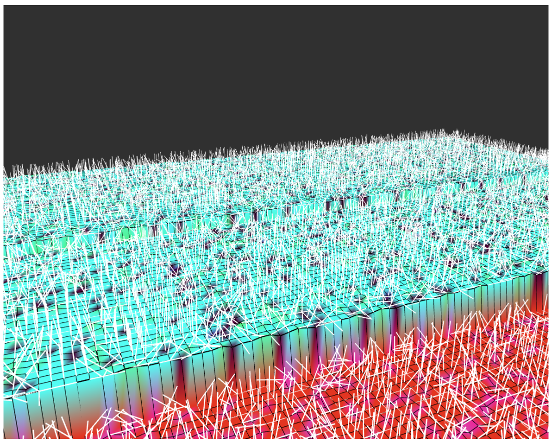


Figure 6.5.: Surface normal estimation from the map on even surfaces with added noise.

6.1.5. Discussion

Given perfect ground truth data, the whole system produces results as expected. However, the framework also showed weaknesses, especially when computing the normals. Surface normals on edges do not get computed as expected. In addition, the surface normals suffer greatly from noise in the point cloud.

6.2. Real Data

After showing the capabilities of the framework using ground truth data, data from a real camera setup are now used to further evaluate the framework.

6.2.1. Used Hardware

To produce the real data, a mock-up consisting of a D455 Realsense depth camera [9] and a T265 Realsense tracking camera [32] is used. The D455 depth camera provides a 3D colored point cloud in the sensor frame. Additionally, the T265 tracking camera provides a pose estimation of the camera in the map frame. Both cameras are attached to a custom 3D printed plate ensuring that the orientation and translation between the two cameras are always fixed and known. With this knowledge, the point cloud provided by the D455 depth camera can be transformed into the map frame using the pose estimation of the T265 tracking camera. A picture of the used camera setup is shown in figure 6.6.

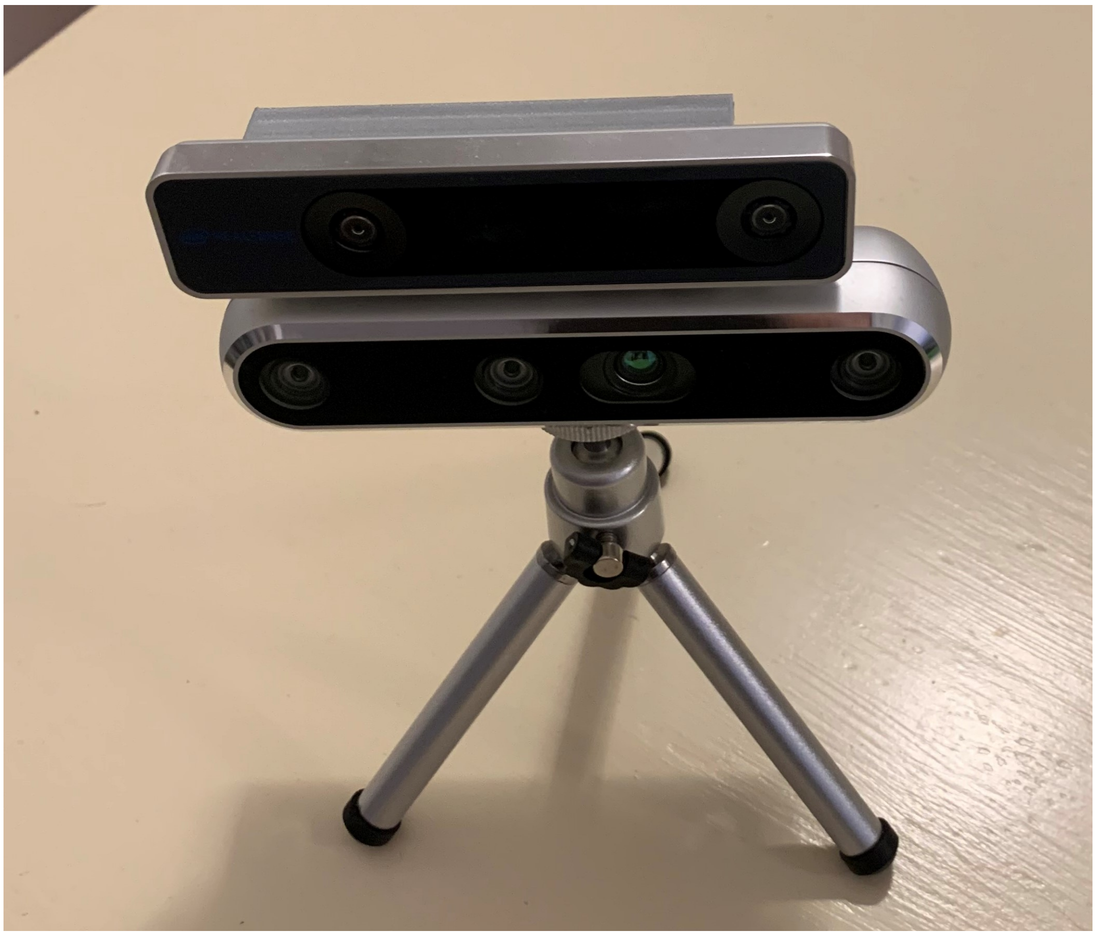


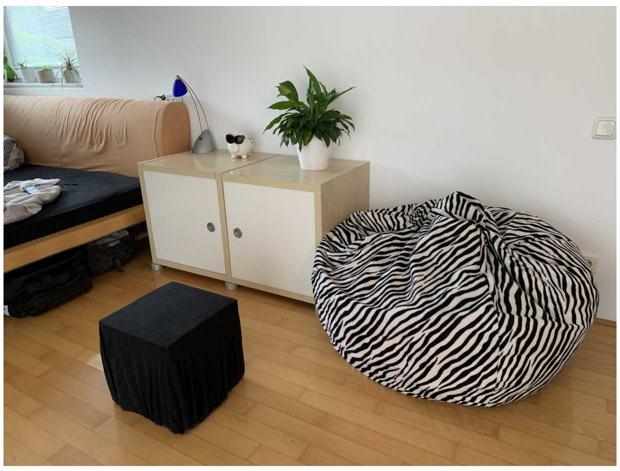
Figure 6.6.: Camera setup consisting of a D455 depth camera [9] and a T265 tracking camera [32].

6.2.2. Static Scenes

To show the functionality of the framework with real data, two static scenes were captured first. One is showing a scene of a typical students' room, the other is a black box on a floor. Both maps got captured with a grid resolution of 2 cm.

Student Room

A picture of the captured scene is shown in 6.7a. The resulting map can be seen in figure 6.7b. Apart from the color mapping, the results show a good mapping of the scene. As the map is only a 2.5D representation, small details like the plant can not be captured, because only one height can be stored in each cell. The color of the grid cell is determined by the point, which is last fused into the cell. This leads to errors in the coloring of the map, especially at the borders of objects, which can be seen in 6.8b.


(a) photo of scene

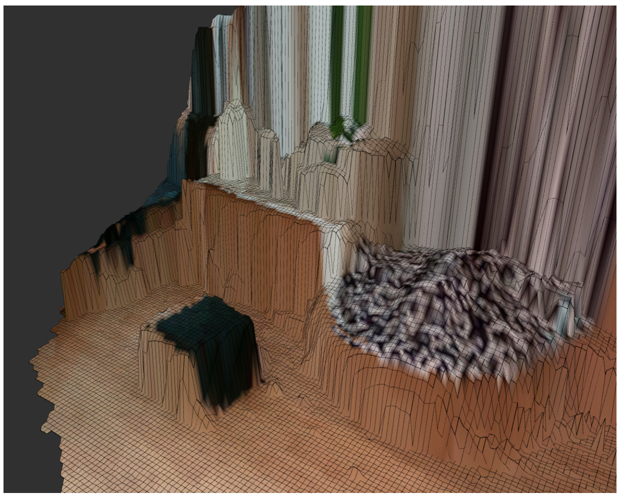

(b) map representation of scene

Figure 6.7.: On the left a picture of the scene and on the right the map representation of the scene.

Black Box Comparison with Ground Truth

Furthermore, a black box was placed on the floor and captured using the mock-up. The box has the dimensions 30 cm×30 cm×30 cm. The resulting map and a ground truth map for comparison can be seen in figure 6.8. Comparing the two maps, it becomes clear that the noise of the camera sensor and the tracking module has a negative effect on the quality of the map. However, it can also be seen that the box, as well as the floor, were captured quite well. Using a grid resolution of 2 cm, the box should be exactly 15 grid cells wide. Counting the grid cells in the map shows that the box is represented by 17 grid cells. This error could stem from the resolution of the map, because the edges of the box could not line up the with edges of the grid cells. Additionally, both the depth camera noise as well as the position error of the tracking camera could lead to such an error. Taking these factors into account, the box is represented fairly well. In comparison to the ground truth map, the map representation is not as smooth, which cloud also be down to the fact that the box

and the floor are not perfectly shaped in reality either.
The biggest drawback can be seen on the computed surface normals, as they greatly suffer from the noise provided by a real camera system.

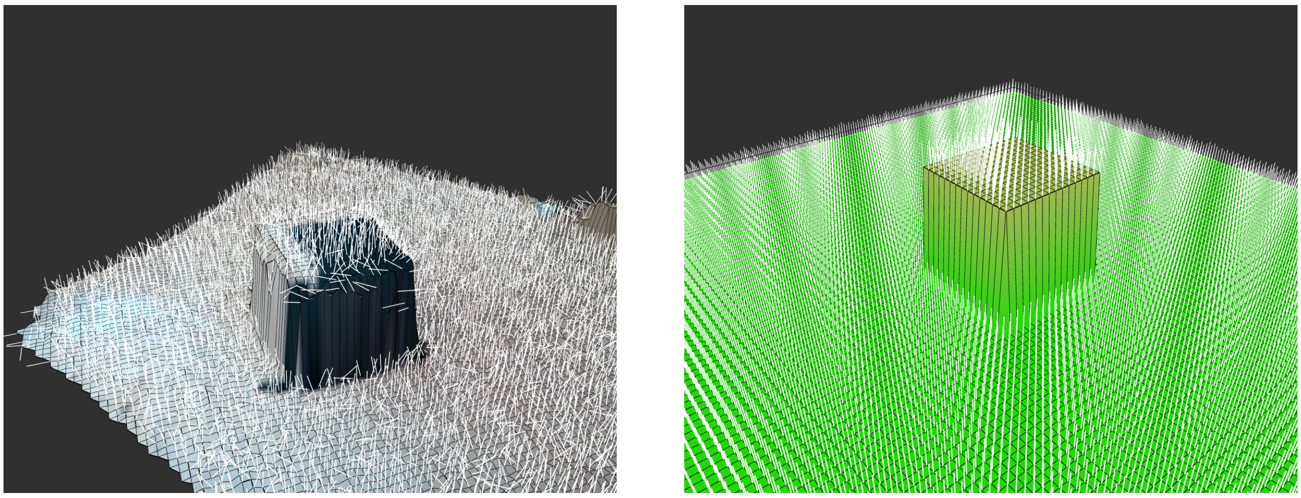
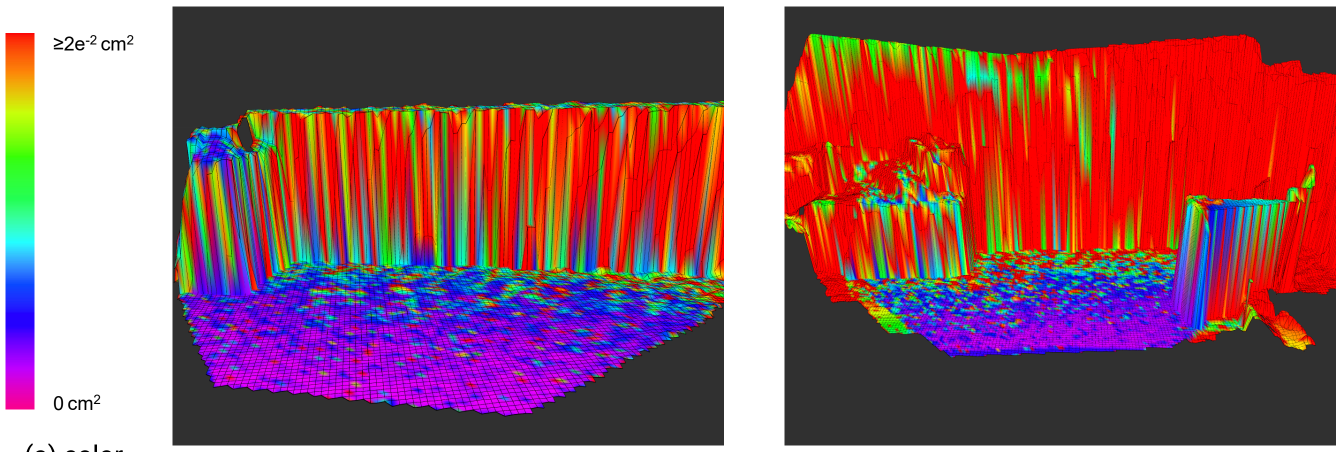

(a) Resulting map from real data of the box with normals computed from the map representation.
(b) Ground truth map of the box with normals computed from the map representation.

Figure 6.8.: Resulting map of the black box from captured real data with the mock-up and ground truth map for comparison.

6.2.3. Variance Computation

Two tests are performed to evaluate the computed variance. One uses a static camera setup, the other uses a moving camera setup. The current implementation of the framework sets all values of the covariance matrix of the robot pose to zero, so the variance of the pose estimation is not included in the computation.
The resulting variance is shown in figure 6.9. Even though the robot pose covariance is set to zero, the variance differs between the two setups as the rotation of the sensors in the map is used to compute the variance. This leads to a different variance when moving the camera, as the rotation constantly changes.
With the static camera setup, the variance is computed using only the noise model of the camera. As no noise parameters for the d455 camera were provided from either Fankhauser et al. [13] or Pan et al. [25], the noise parameters of the d435 provided from Fankhauser et al. [13] are used and can be seen in A.1. The pictures in figure 6.9 show that the variance increases when the points are further away from the camera. This resembles the noise of RGBD Cameras well, as they get noisier with larger distances to measured points.



(a) color scale

(b) Computed variance on map representation captured from a static camera setup.

(c) Computed variance on map representation captured from a moving camera setup.

Figure 6.9.: The computed variance with a static and a moving camera setup.

6.2.4. Traversable Score

For the static scene of the student room, the traversable score is computed using the same robot parameters as in 6.1.3. The resulting traversable score is shown in figure 6.10. Compared to the computed traversable score on the ground truth datasets see in figure 6.3, here the traversable score is not as often equal to zero or one. This is an expected outcome, because the map representation captured from a real system is not as smooth as the ground truth data. However, the traversable score still clearly shows a difference between the individual sections of the map. Walls and vertical surfaces are evaluated as untraversable, whereas the floor and even surfaces can easily be traversed by the robot.

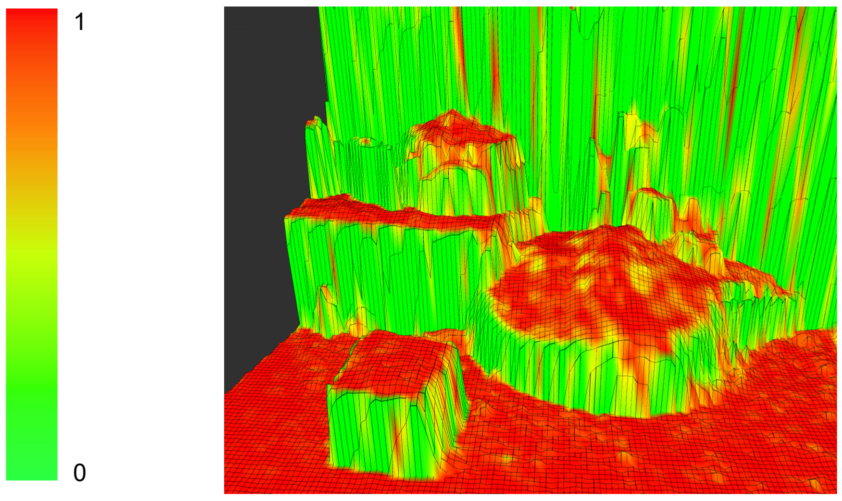
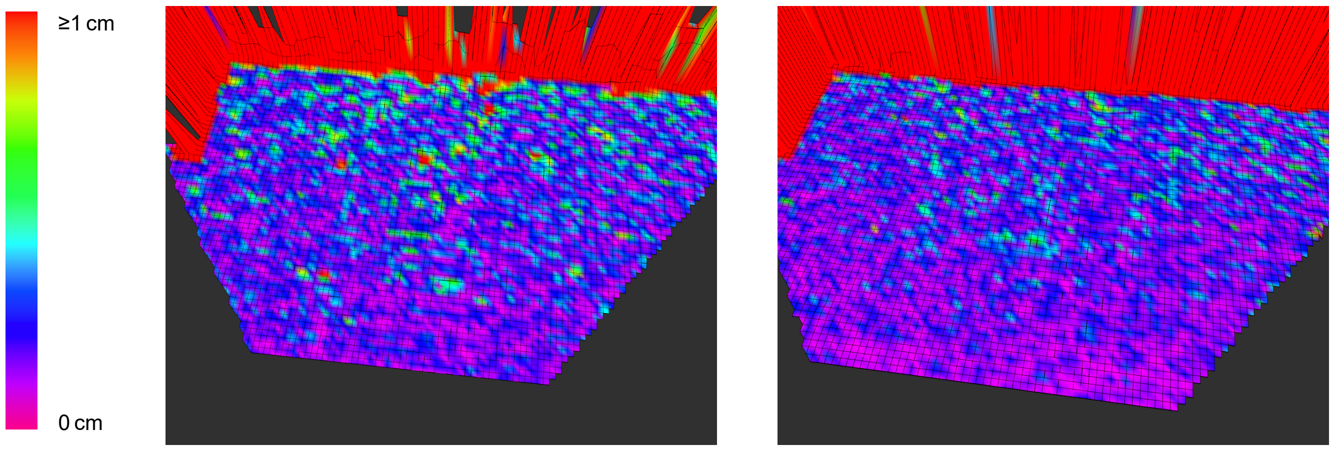


Figure 6.10.: The resulting traversable score for the static scene, which can be seen in picture 6.7a. A score of one describes a traversable area, whereas a score of zero denotes an untraversable area.

6.2.5. Comparison to the Current Mapping Method in the *L3 Terrain Model Generator*

In order to compare the smoothing effect of the mapping method, a plain floor is captured by the camera. Then, the data is used for both the currently used mapping method in the *L3 Terrain Model Generator*, as well as the new mapping method presented in this thesis. The roughness filter provided by the grid_map library [15] is used to evaluate the roughness of the captured scene. The maps after filtering can be seen in figure 6.11. Comparing the resulting maps of the two approaches shows that the probabilistic mapping approach is less rough than the mapping approach from the *L3 Terrain Model Generator*, which only uses a simple update weight to update the grid cell. The results show that sensor noise is handled better in the new approach presented in this thesis, which makes it more robust.

Additionally, vertical walls are better represented using the new approach, as points with lower heights do



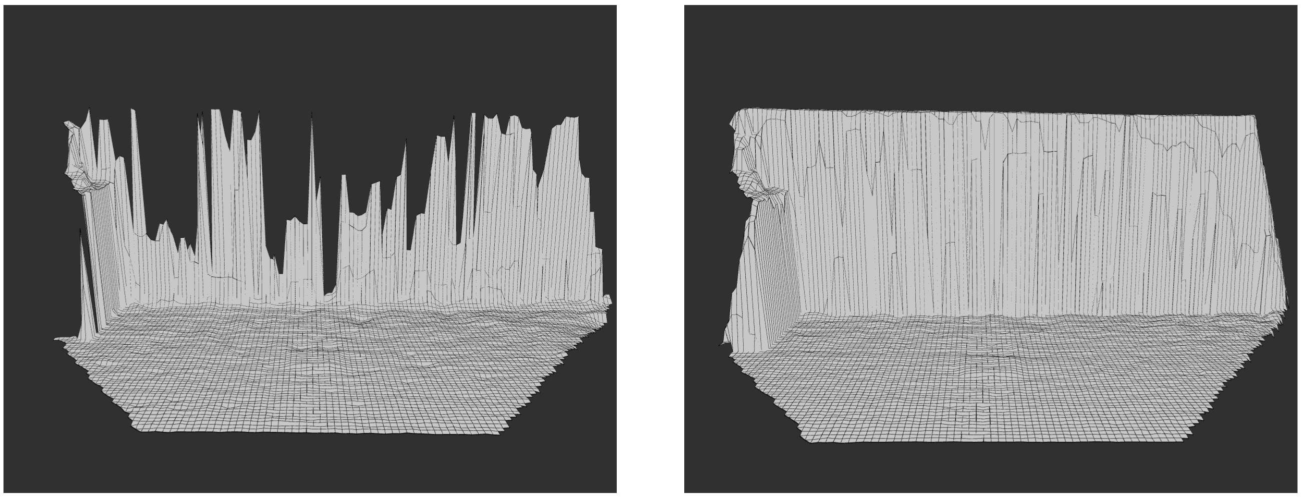
(a) color scale (b) Deviation map and smooth surface approach currently used in *L3 Terrain Model Generator*. (c) Deviation map and smooth surface using newly proposed approach in this thesis.

Figure 6.11.: On the left the representation of an even horizontal surface from the currently used approach in *L3 Terrain Model Generator*and on the right the representation from the proposed approach in this thesis. The color shows the deviation between the smoothed surface and the actual map representation.

not shrink the wall again, as is the case in figure 6.12a. In the mapping approach in the *L3 Terrain Model Generator* all points in a grid cell are used to update the height, which leads to incomplete vertical walls. The approach presented in this thesis rather tests whether the new points are within certain range of the currently stored height in the grid cell. If the points are too low, they will not be considered while updating the height. If they are higher, the grid cell will be reinitialized with a greater height. This leads to a good representation of vertical walls.

6.2.6. Discussion

The provided map representations show that static scenes can be captured well using the framework proposed in this work. The coloring of the grid cell is not very good, due to the fact that only one random point laying in the grid cell is used to color it. This shortcoming could be improved by using the most relevant point for coloring the grid cell. The 2.5D elevation map can not represent all details of the real world as only one height can be stored in each grid cell. The grid resolution plays a major role in determining how many details can



(a) Vertical surface representation from the old approach in *L3 Terrain Model Generator*.

(b) Vertical surface representation from the proposed approach in this thesis.

Figure 6.12.: On the left the representation of a vertical surface from the old framework and on the right the representation of a wall from the new framework.

be seen in the resulting map. Nevertheless, the framework delivers results for both the computation of the variance and the traversable area, which are in line with the expectations.

6.3. Comparison between CPU and GPU Performance for Map Update

For the purpose of evaluating the efficiency of the mapping approach, the performance of each individual updating method is tested. The hardware used for testing is the Nvidia Jetson AGX Xavier developer kit [22]. The framework is both tested with artificial data and real data from the camera setup explained in section 6.2.1.

6.3.1. Artificial Data

To test the framework with an artificial dataset, the terrain dataset from the grid_map library is used. Using a grid resolution of 2 cm, the resulting map has a size of 10 m × 10 m. The mono-colored image used to create the dataset can be seen in 6.1b and the resulting map is shown in 6.2b. The generated input cloud has 250,000 individual points, which resembles the size of a point cloud from a RGBD-Camera. The map size resembles the size of a map captured by a multiple RGBD-Camera setup, which covers a 360° field of view around the robot. In order to test the ray-tracing, the sensor position is placed into the middle of the map at a height of 1 m. To ensure that the same amount of grid cells are considered while ray-tracing for both the CPU and GPU approach, only the last two rows on the end of the map are used, which leads to 1000 cells being considered.
To measure the computation time needed by the mapping approach, the input cloud was fused into the system a hundred times. The average consumed time and its standard deviation for updating the map on the CPU and the GPU are denoted in table 6.1. All durations were measured on the Nvidia Jetson AGX Xavier [22]. The results show that the GPU implementation takes less time across all categories, being twice as faster for

both the map update and the map update with additional traversable area computation. Most noticeable is the difference between the CPU and the GPU implementation when ray tracing is used to handle dynamic objects. There, the GPU implementation is up to four times faster.
The CUDA API statistics show how much time the different operations consumed on the GPU. When updating the variance, height and the traversable score, it can be seen that 95% of the GPUs time is spent copying the data from the computer memory to the GPU memory and vice versa. This shows how time-consuming the copy of data is and that the mapping framework could be significantly faster, if fewer data were needed to be transferred.

Update Step on artificial data	CPU time in [s]	Standard deviation CPU in [s]	GPU time in [s]	Standard deviation GPU in [s]
point variance + height update	1.3822	0.0977	0.7877	0.0169
point variance + height update + traversable area	1.5999	0.0734	0.8588	0.0165
point variance + height update + traversable area + dynamic object handling	19.0262	0.4850	5.2468	0.0181

Table 6.1.: Computation time comparison between the CPU and GPU implementation on artificial data using the test terrain dataset.

6.3.2. Real Data

To test the implementations on real data, the dataset from the scene shown in 6.7 is used. The scene was captured as a ROS bag-file to make sure that the test data for both implementations is consistent. While the size of the input point cloud is nearly identical to the artificial dataset, the map size is significantly smaller having only a size of 5 m × 3 m. As a consequence, it is expected that the updating of the map runs faster than previously. Trough computing the traversable score identically for both implementations, it is ensured that the same amount of grid cells are considered while ray tracing. Only the height deviation is taken into account to compute the traversable score, as it is unclear whether the CPU and the GPU implementation of the 3F2N code produce the exact same normal estimation. The resulting average computation time and its standard deviation of the hundred taken measurements are shown in table 6.2. Again, the GPU version is significantly more efficient than the CPU implementation being between two and three times faster.
The CUDA API statistics show how much time the different operations consumed on the GPU. When updating the variance, height and the traversable score, it can be seen that 83% of the GPUs time is spent for copying the data from the computer memory to the GPU memory and vice versa. This shows why the GPU is so much faster using real data, as the map size is only around a fourth of the size of the map using artificial data. The consumed computation time is clearly depending mostly on how much data need to be transferred between the computer and the GPU memory.

Update Step on real data	CPU time in [s]	Standard deviation CPU in [s]	GPU time in [s]	Standard deviation GPU in [s]
point variance + height update	0.3633	0.028561	0.1098	0.0239
point variance + height update + traversable area	0.3844	0.0271	0.1625	0.0205
point variance + height update + traversable area + dynamic object handling	8.1946	1.9101	4.7074	0.7277

Table 6.2.: Computation time comparison between the CPU and GPU implementation on real data using the student room dataset.

6.3.3. Discussion

Overall, the GPU implementation outperforms the CPU implementation as it is at least two times faster across all categories. Using the GPU implementation on real data the map can be updated up to ten times a second, which can be interpreted as real time processing. However, the most time is lost while transferring the map data from the computer memory to the GPU memory, which shows that the data transferred should be decreased in order to achieve a significant speed-up. The map size is the most predominant factor influencing the used computation time as it determines how much data need to be transferred. This problem could be addressed by always holding a map of fixed size in the GPU memory as then, only the data of the input point cloud would need to be transferred.
Ray-tracing is a very computational demanding operation and can not be performed at the same rate taking a minimum of four seconds. Consequently, further research should address how the system could be improved to create a good map representation in a dynamic environment in real-time.

6.4. Dynamic Object Handling

As the comparison between the CPU and the GPU approach has shown, dynamic object handling is a very computational demanding operation for both approaches. Dynamic object handling is not needed in a static environment and should not be used to ensure good performance. However, in dynamic environments, it is essential to clear passed dynamic objects.

6.4.1. Comparison Dynamic Object Handling Toggled On and Off

A reference picture of the used scene to evaluate the dynamic object handling can be seen in 6.13. The resulting map representations in figure 6.14 show a scene before an object is added, while the object is in the scene and after the object is removed again. On the left side, the map representation without dynamic object

handling is shown. On the right side, the same scene is shown, however, this time dynamic object handling is toggled on. The advantage of having dynamic objects handling on becomes very apparent, because the object would otherwise congest the map, as is the case in picture 6.14e. This would lead to wrong information for other applications using the map, for example, path planning.
In the last picture on the right side 6.14f, a small spike can be seen, which is the last remnant of the obstacle. In the last picture on the left side 6.14e, a big cylinder, which is even bigger than the object that should have been captured, remains in the map representation. This is due to the fact that the arm of the person placing the object is captured as well.

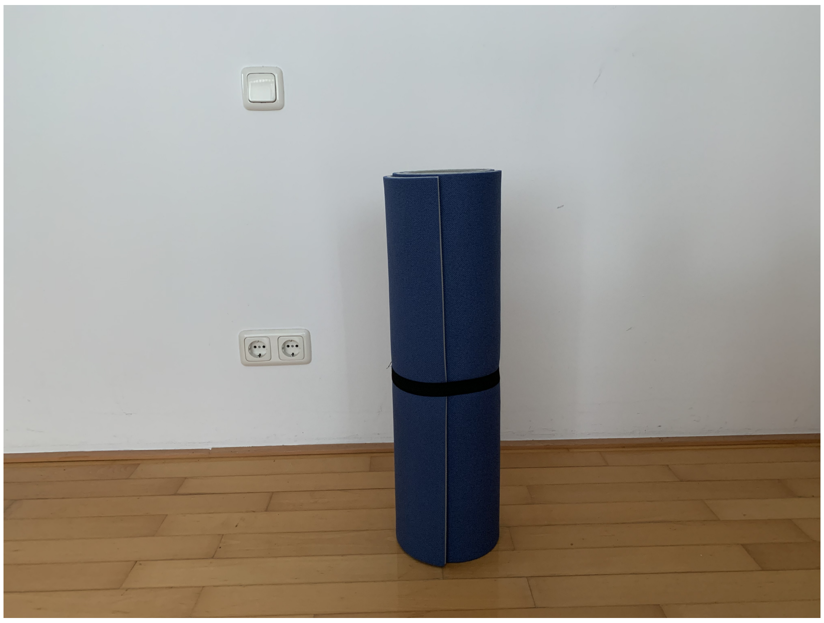
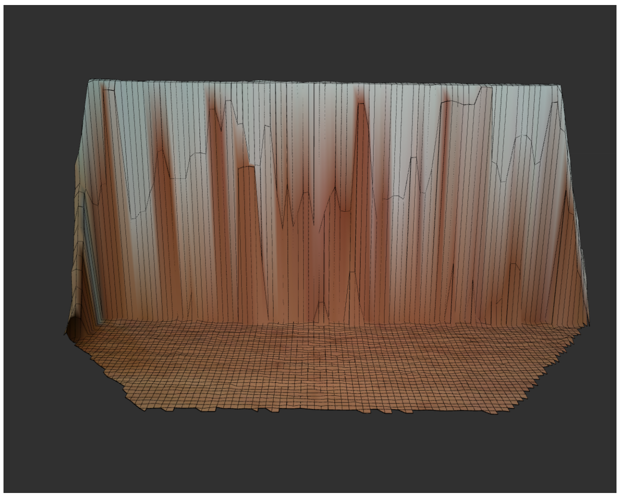
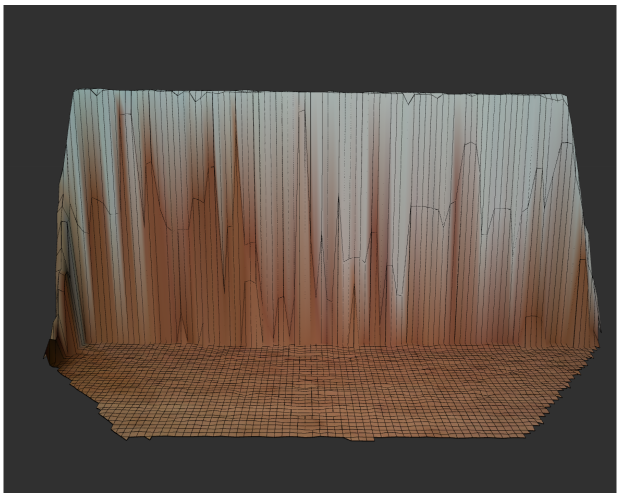


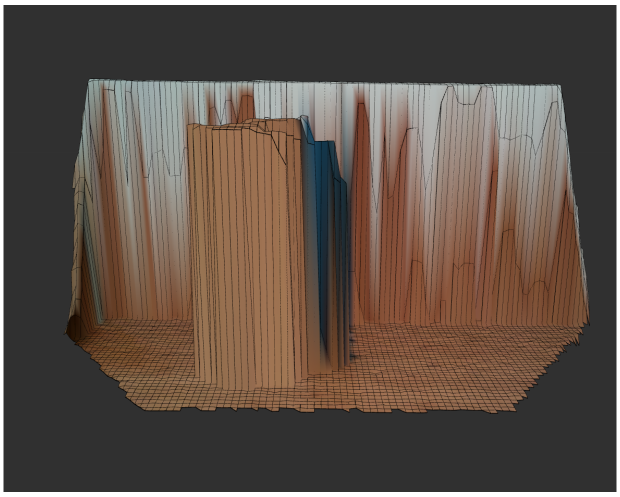
Figure 6.13.: The scene used for testing the dynamic object handling. While testing the blue object is placed into the empty scene and later is removed again.

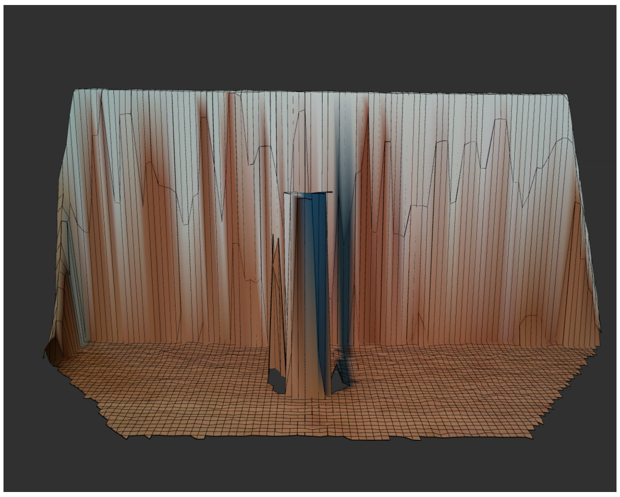
6.4.2. Discussion

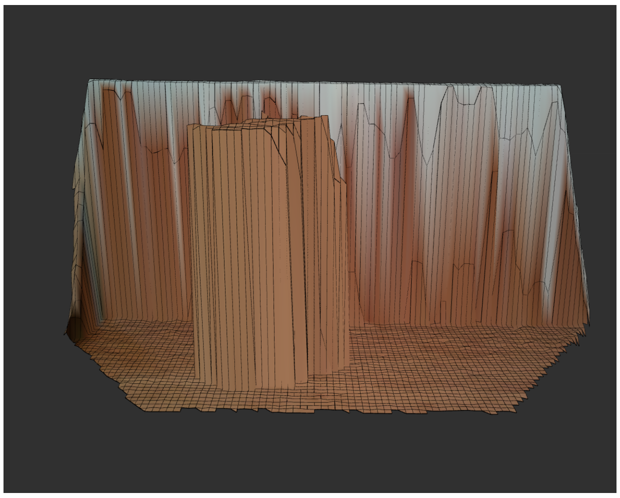
Looking at the results in the tables 6.1 and 6.2, it becomes clear that the dynamic object handling can not be performed in real-time from the system, because the duration needed for the computation is too long. However, it could be used at a lower rate in order to clear dynamic objects from the map representation.
Additionally, dynamic objects are not fully cleared, as shown in picture 6.14e. Small spikes on the floor remain while handling dynamic objects, because the proposed ray-tracing algorithm stops a few voxels early in order for vertical surfaces to not clear themselves.


(a) Before the object is in frame: Without dynamic object handling.


(b) Before the object is in frame: With dynamic object handling.


(c) While object is in frame: Without dynamic object handling.


(d) While object is in frame: With dynamic object handling.


(e) After the object was removed from the frame: Without dynamic object handling.

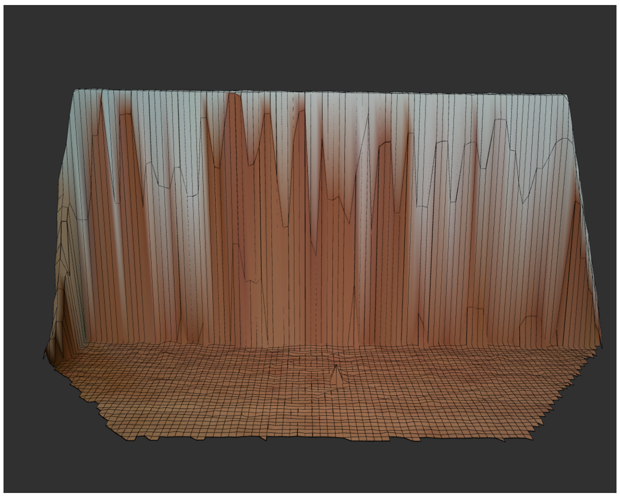

(f) After the object was removed from the frame: With dynamic object handling.

Figure 6.14.: The map before, while and after a dynamic object entered and left the frame, with dynamic object handling toggled on (right side) and off (left side).

6.5. Surface Normal Estimator

In this section, the presented surface normal estimator is tested and compared against the currently used surface normal estimator in the *L3 Terrain Model Generator*. For this, both the accuracy and the performance are evaluated.

6.5.1. Estimation from Input Point Cloud or Map

In this thesis, two approaches to retrieve the surface normal estimation are proposed. The first one uses the 3F2N code directly on the input point cloud from the RGBD-Camera and then fuses the estimated normals into the map. The other approach uses the updated map representation to compute the normals from the stored data in the map. This approach was proposed under the assumption that the map update can be seen as a filtering mechanism. As already discussed in section 6.2.5, the probabilistic mapping approach handles sensor noise better than the previously used mapping approach in the *L3 Terrain Model Generator*.
Aiming to test the two proposed surface normal estimation approaches, a dataset of an even horizontal surface is created by pointing the camera setup towards the floor. Figure 6.15 shows the resulting normals from both approaches, whereas figure 6.16 shows the deviation of the computed and the expected normals of the surface. The shown deviation is computed as the angle between the expected and the computed normals. The average measured error angle between the expected normals and the ones computed from the approaches at twenty different time steps can be seen in table 6.3. The resulting deviation shown in figure 6.16 and the average error angle clearly show that the normals from the input cloud are more accurate than the normals computed from the map representation.

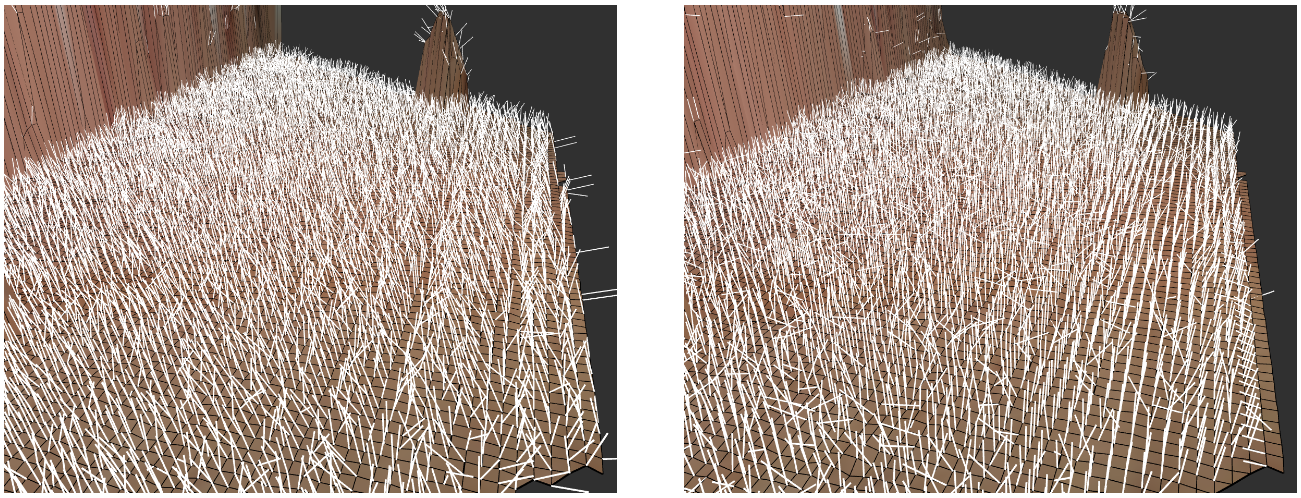
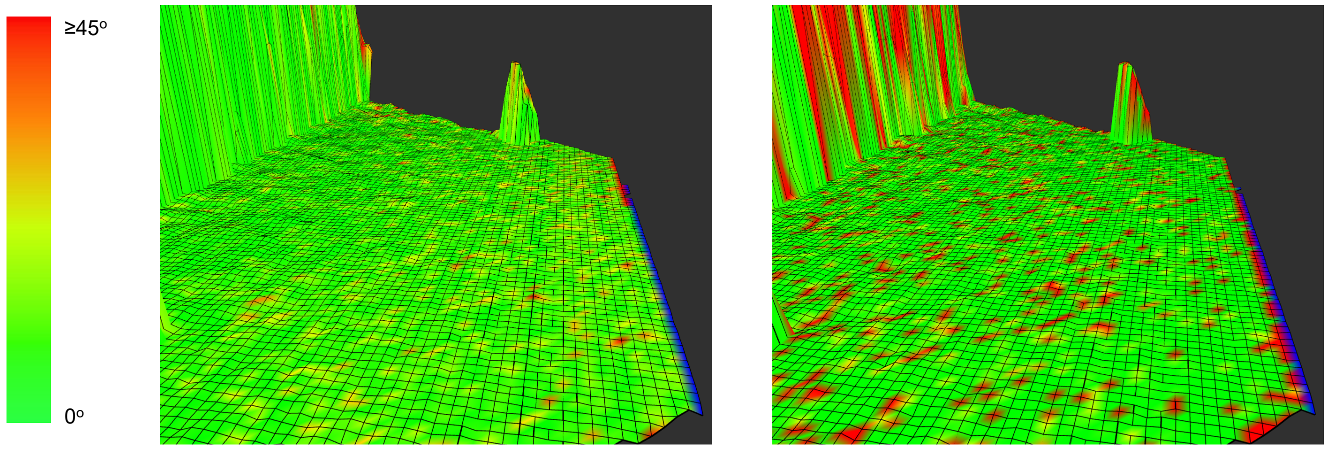

(a) normals from input cloud (b) normals from map

Figure 6.15.: On the left the normals computed directly from the input cloud and on the right the normals computed from the map.

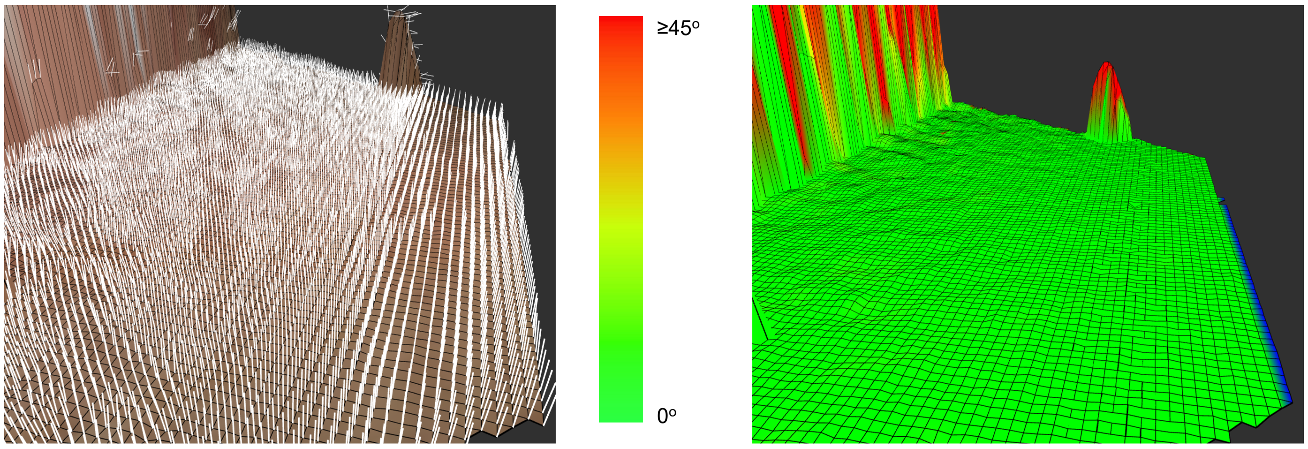


(a) color scale (b) Deviation from expected normals computed from input cloud (c) Deviation from expected normals computed from map

Figure 6.16.: The deviation between the computed and the expected normals for both proposed approaches.

6.5.2. Comparison with Currently Used Estimator in *L3 Terrain Model Generator*

The same scene explained above is used to compute the normals with the current approach used in the *L3 Terrain Model Generator*. The resulting normals and their deviation from the expected normals can be seen in figure 6.17. Furthermore, the average error angle is denoted in table 6.3. It is computed trough comparing the computed normals with the expected normals for the surface at twenty different time steps. As the results show, the surface normals are much more accurate compared to the normals of the two proposed approaches in this thesis.



(a) Normals computed by currently used approach in *L3 Terrain Model Generator* (b) color scale (c) Deviation from expected normals

Figure 6.17.: On the left the estimated normals computed from the currently used approach in the *L3 Terrain Model Generator*. On the right the deviation of these normals from the expected ones for the given surface.

Approach	New approach from input cloud	New approach from map	Old approach in *L3 Terrain Model Generator*
Average error angle in [degrees]	6.98	11.34	1.521

Table 6.3.: The table shows the average error angle between the estimated surface normals from the three aproaches and the expected normals of the surface.

6.5.3. Performance Comparison

Due to not being able to run the 3F2N [12] code on the ARM processor of the Nvidia Jetson AGX Xavier, the CPU implementation got tested using another PC with an Intel Core i5-4670 CPU with four cores. To keep consistency while testing, the current surface normal estimator of the *L3 Terrain Model Generator* is also tested on the other PC. The GPU implementation is tested on the Nvidia Jetson AGX Xavier.

Artificial Data

The following tables (6.4 and 6.5) show the average consumed time of the different approaches to compute the surface normals. To measure the time, the input cloud was fused into the system a hundred times. The average consumed time and its standard deviation are denoted in the tables. The table 6.4 shows the measured durations using the test terrain map presented at the beginning of the chapter, which creates an input cloud with 250,000 points. The table 6.5 shows the measured durations using the stairs dataset presented at the beginning of this chapter. With a grid resolution of 2 cm, the stairs map has a size of 2.4 m × 4.8 m, which creates an input cloud of 28,800 points. Due to the long duration needed of the currently used normal estimator in the *L3 Terrain Model Generator*, only the CPU and the GPU implementation of the surface normal estimator developed in this thesis are tested on the larger dataset.
The results show that the CPU approach presented in this thesis is the fastest of the three approaches, even being faster than the GPU approach. The CUDA API statistics show that most of the time is spent on transferring data between the computer memory and the GPU memory and vice versa. While testing with the terrain dataset, it can be seen that 75% of the time is lost transferring data. The computation on the CPU is already so fast that the additional transfer of data reduces the overall performance.
The currently used approach in the *L3 Terrain Model Generator* is very slow and can not produce surface normals in real-time. The new proposed approach is over a thousand times faster, thus being able to produce surface normal estimation in real-time.

Real Data

In a next step, the surface normal estimator's performances are tested using real data by using the dataset of the student room scene, which can be seen in 6.7. The average consumed computational time and the standard deviation of the hundred taken measurements are shown in table 6.6.
Again, the new proposed approach drastically outperformed the currently used one in the *L3 Terrain Model Generator*. Further, the CPU version is two times faster than the GPU version.

Normals on terrain dataset	New approach CPU time in [s]	Standard deviation new approach CPU in [s]	New approach GPU time in [s]	Standard deviation new approach GPU in [s]
Surface normal estimation	0.0922	0.0003	0.1263	0.0093

Table 6.4.: Computation time comparison between the CPU and GPU implementation of the surface normal estimator using the ground truth terrain dataset.

Normals on stairs dataset	Old approach CPU time in [s]	Standard deviation old approach CPU in [s]	New approach CPU time in [s]	Standard deviation new approach CPU in [s]	New approach GPU time in [s]	Standard deviation new approach GPU in [s]
Surface normal estimation on artificial data	26.1946	2.7409	0.0098	0.0007	0.0173	0.0016

Table 6.5.: Computation time comparison between the old surface normal estimator and the new one with CPU and GPU implementation using the ground truth stairs dataset.

Normals on student room dataset	Old approach CPU time in [s]	Standard deviation old approach CPU in [s]	New approach CPU time in [s]	Standard deviation new approach CPU in [s]	New approach GPU time in [s]	Standard deviation new approach GPU in [s]
Surface normal estimation on real data	17.7052	11.6658	0.0053	0.0009	0.0108	0.0019

Table 6.6.: Computation time comparison between the old surface normal estimator and the new one with CPU and GPU implementation using the student room dataset.

6.5.4. Discussion

The results in the previous section show that the currently used approach in the *L3 Terrain Model Generator* can not produce normals in real-time, whereas the presented approaches in this thesis are more than capable to do so. The CPU version outperformed the GPU version suggesting that the additional data transfer between

GPU and computer memory has an negative effect on the performance.
However, the accuracy of the computed surface normals of the proposed approaches is not as good as the accuracy of the current approach in the *L3 Terrain Model Generator*. With a speed-up of over a thousand times, the CPU approach demonstrates a significant performance boost. If the application requires real-time calculation, the newly introduced approaches in this work should be used. The accuracy however should be improved in future works.
Comparing only the two approaches developed in this thesis, the one using the input point cloud to compute the surface normals shows significantly higher accuracy.

6.6. Summary

To conclude this chapter, an overview of the benefits and the remaining challenges of the proposed approach is given.

Benefits: Using ground truth data, the system is able to produce the map representation, traversable score and surface normals according to the expectations from an artificial input cloud. Further, using real data from a camera setup, the system produces consistent map representations. The variance of the height measurement and the traversable area annotation produce meaningful results. Additionally, the system demonstrates its capability of clearing dynamic objects from the map. Using high parallelization shows to be a big advantage, especially using the GPU, as the map is able to update two times faster than the CPU. Furthermore, the surface normal estimator proves to be very efficient, being over a thousand times faster than the approach currently used in the *L3 Terrain Model Generator*. Using real data, the system is capable of updating the map, the surface normals, and the traversable score of up to ten times a second.

Remaining Challenges: The surface normals are computed very fast, but their accuracy is not very good as they are especially susceptible to sensor noise. Additionally, the system shows that it will not be able to produce a consistent map representation in dynamic environments, because the computation time used for ray-tracing is too high. Even though the system is capable of updating the map up to ten times a second, the normal update rate of RGBD-Cameras, which is as high as thirty times a second, can not be reached. The GPU approach would especially benefit from a robot-centric map representation, as then a map of consistent size could always be held in the GPU memory. This in turn would drastically reduce the data that needs to be transferred between computer and GPU memory. The coloring of grid cells, which is especially fragile around the edges of objects, should be looked at in future research.

7. Conclusion and Future Work

In the following chapter, the results achieved in this thesis are summarized.
Furthermore, potential work based on the shown results is discussed.

7.1. Conclusion

This thesis gives an overview of different mapping and surface normal estimation approaches. Furthermore, common open-source programs are introduced and it is evaluated, whether they are suitable for the proposed method.
Based on the discussed mapping approaches, a mapping system is proposed, which uses a probabilistic elevation map representation with added color, traversable area computation and dynamic object handling. Additionally, the 3F2N approach from [12] is used and enhanced, such that normals can be computed from an organized point cloud and a grid-based elevation map.
The mapping approach computes the variance for the sensor data using the robot's pose covariance matrix and a sensor model. The corresponding height and variance are then stored in a probabilistic elevation map using the grid_map library [15]. In a next step, the deviation between the height at each grid cell and its neighbors and the computed surface normals are used to calculate the traversable area on the map. Finally, the system tackles the problem of dynamic objects congesting the map representation by using a ray-tracing based approach to clear off passed dynamic objects.
The main goal of the proposed approach is to maximize efficiency, as the computational resources of mobile robots are limited. Using massive parallelization at every step of updating the map, the proposed approach works efficiently on both multi-threaded CPUs and GPUs.
The system is evaluated using both ground truth and real RGBD-Camera data. The ground truth datasets are generated using the available functionality of the grid_map library to transform mono-colored images into a map representation. The real data are captured using a camera system consisting of a RealSense D455 and a RealSense T265 camera.
While the proposed surface normal estimation approaches show to be significantly more efficient than the one currently used in the *L3 Terrain Model Generator*, they are prone to exhibit a lower accuracy. From the proposed two approaches to compute the normals, the accuracy of the one estimating the normals from the input cloud instead of the map representation is significantly higher.
The proposed mapping system shows clear advantages over the one used in the *L3 Terrain Model Generator*, especially at reducing the impact of sensor noise and capturing vertical surfaces. This gives reason to believe that the developed system is more robust. Furthermore, the uncertainty is captured and stored for every grid cell of the map. Computing the traversable area on both the ground truth and the real data produces meaningful results.
The evaluation shows that the ray-tracing based approach used is capable of clearing dynamic objects well, though it also becomes apparent that the operation is too computationally intensive to be used in real-time

applications.
Finally, the CPU and GPU implementations were benchmarked to find out which of the two methods is most efficient. As part of this, it is found that the GPU approach is already significantly faster than computing the map update using the CPU, despite the fact that the majority of computation time is spent transferring data between the computer and the GPU memory. Given no dynamic object handling is done and the data come from a single RGBD-Camera, the proposed approach can easily handle updating the map with a rate of up to ten Hz. Once additional cameras are added, which in turn increases the data load, updating the map is only possible at a lower rate, thus being less suitable for real-time applications.

7.2. Outlook

Based on the results illustrated above, future research could address the following points to further increase both the efficiency as well as the robustness with which map representations can be produced.
First, the color mapping could be improved, as currently, the last fused point determines the color of the grid cell, which shows to lead to wrong coloring especially around the edges. Using only the most relevant point, which is the point closest to the height of the grid cell, to color the cell could be an improvement. However, this detail does not have any implications for the robot, but rather is only of relevance to the human eye.
Second, the point variance is computed using a sensor model, therefore, the specific sensor model should be selected for the respective camera. Due to fact that the D455 was not yet on the market when [13] was published, there were no parameters for the sensor model of the camera provided by Fankhauser et al.. As a consequence, the parameters for the D455 should be determined. It is unclear to what extent the variance would differ if an updated sensor model for the D455 camera was used.
Third, while the proposed surface normal estimators are already very time efficient, their accuracy could be further increased. Future work could achieve this by applying different filtering methods for the normals. Additionally, the size of the used filter masks from 3F2N could be increased, which might further improve the accuracy of the computed normals. This, however, could also harm the efficiency of the system, as additional computational steps need to be carried out.
Fourth, for the dynamic object handling to work in real-time, future work should seek to reduce the number of points used for ray-tracing, thus lowering the computational effort substantially. As often multiple points fall into the same grid cell, one strategy might be considering only the lowest point at each grid cell. Furthermore, the problem of small artifacts remaining in the map after an object was cleared should be tackled to improve the consistency of the map.
Lastly, although it is shown that it is more favorable to use the GPU instead of the CPU for mapping, this difference in efficiency can even be extended, if one could reduce the amount of data that need to be transferred between computer and GPU memory. This could be achieved by storing a map representation of fixed size on the GPU memory at all times, thus only transferring the data of the input point cloud to the GPU. Furthermore, this could be enhanced by using a robot-centric mapping approach, where only a map of fixed size around the robot is maintained, instead of a world-centric mapping approach. Such an approach is already used in [25]. While still capitalizing on the high core count of the GPU to compute the different update steps in parallel, fewer data need to be transferred when applying these suggestions. Therefore, a significant speed-up of the GPU version is expected, which could make it feasible to handle data from multiple RGBD-Cameras sufficiently well, thus potentially allowing the system to be suitable in real-time applications.

Bibliography

[1] H. Badino, D. Huber, Y. Park, and T. Kanade, "Fast and accurate computation of surface normals from range images", in *2011 IEEE International Conference on Robotics and Automation*, 2011, pp. 3084–3091. DOI: 10.1109/ICRA.2011.5980275. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/5980275.

[2] M. Bloesch, H. Sommer, T. Laidlow, *et al.*, *A primer on the differential calculus of 3d orientations*, 2016. arXiv: 1606.05285 [cs.RO].

[3] G. Bradski, "The OpenCV Library", *Dr. Dobb's Journal of Software Tools*, 2000.

[4] G. Bradski and A. Kaehler, *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, Inc., 2008, ISBN: 9780596554040. [Online]. Available: https://books.google.de/books?hl=de&lr=&id=seAgiOfu2EIC&oi=fnd&pg=PR3&dq=opencv&ots=hVG1dlfHPh&sig=0wGaZShKr5LzW0ELnIeH5jsNeMM#v=onepage&q=opencv&f=false (visited on 09/16/2021).

[5] J. E. Bresenham, "Algorithm for computer control of a digital plotter", *IBM Systems Journal*, vol. 4, no. 1, pp. 25–30, 1965. DOI: 10.1147/sj.41.0025.

[6] A. Broggi, P. Grisleri, and P. Zani, "Sensors technologies for intelligent vehicles perception systems: A comparison between vision and 3d-lidar", in *16th International IEEE Conference on Intelligent Transportation Systems (ITSC 2013)*, 2013, pp. 887–892. DOI: 10.1109/ITSC.2013.6728344.

[7] L. Chiang, "3-d cnc trajectory interpolation using bresenham's algorithm", in *Proceedings of 1994 IEEE International Symposium on Industrial Electronics (ISIE'94)*, 1994, pp. 264–268. DOI: 10.1109/ISIE.1994.333107.

[8] "Cuda". (), [Online]. Available: https://developer.nvidia.com/cuda-toolkit (visited on 10/04/2021).

[9] "D455". (), [Online]. Available: https://www.intelrealsense.com/depth-camera-d455/ (visited on 10/04/2021).

[10] A. Dai, A. X. Chang, M. Savva, M. Halber, T. Funkhouser, and M. Niessner, "Scannet: Richly-annotated 3d reconstructions of indoor scenes", in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jul. 2017. [Online]. Available: https://openaccess.thecvf.com/content_cvpr_2017/html/Dai_ScanNet_Richly-Annotated_3D_CVPR_2017_paper.html.

[11] R. De Maesschalck, D. Jouan-Rimbaud, and D. Massart, "The mahalanobis distance", *Chemometrics and Intelligent Laboratory Systems*, vol. 50, no. 1, pp. 1–18, 2000, ISSN: 0169-7439. DOI: https://doi.org/10.1016/S0169-7439(99)00047-7. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0169743999000477.

[12] R. Fan, H. Wang, B. Xue, *et al.*, "Three-filters-to-normal: An accurate and ultrafast surface normal estimator", *IEEE Robotics and Automation Letters*, vol. 6, no. 3, pp. 5405–5412, 2021. DOI: 10.1109/LRA.2021.3067308. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9381580.

[13] P. Fankhauser, M. Bloesch, and M. Hutter, “Probabilistic terrain mapping for mobile robots with uncertain localization”, *IEEE Robotics and Automation Letters*, vol. 3, no. 4, pp. 3019–3026, 2018. DOI: `10.1109/LRA.2018.2849506`. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/8392399`.

[14] P. Fankhauser, M. Bloesch, D. Rodriguez, R. Kaestner, M. Hutter, and R. Siegwart, “Kinect v2 for mobile robot navigation: Evaluation and modeling”, in *2015 International Conference on Advanced Robotics (ICAR)*, 2015, pp. 388–394. DOI: `10.1109/ICAR.2015.7251485`.

[15] P. Fankhauser and M. Hutter, “A universal grid map library: Implementation and use case for rough terrain navigation”, in *Robot Operating System (ROS)*, A. Koubaa, Ed., ser. Studies in Computational Intelligence. Cham: Springer International Publishing, 2016, vol. 625, pp. 99–120, ISBN: 978-3-319-26052-5. DOI: `10.1007/978-3-319-26054-9{\textunderscore}5`. [Online]. Available: `https://link.springer.com/chapter/10.1007/978-3-319-26054-9_5`.

[16] F. Govaers, *Introduction and Implementations of the Kalman Filter*. IntechOpen, 2019, ISBN: 9781838805364. [Online]. Available: `https://books.google.de/books?id=YhT8DwAAQBAJ`.

[17] G. Guennebaud, B. Jacob, *et al.*, *Eigen v3*, http://eigen.tuxfamily.org, 2010.

[18] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle, “Surface reconstruction from unorganized points”, in *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH ’92, New York, NY, USA: Association for Computing Machinery, 1992, pp. 71–78, ISBN: 0897914791. DOI: `10.1145/133994.134011`. [Online]. Available: `https://doi.org/10.1145/133994.134011`.

[19] K. Jordan and P. Mordohai, “A quantitative evaluation of surface normal estimation in point clouds”, in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2014, pp. 4220–4226. DOI: `10.1109/IROS.2014.6943157`. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/6943157`.

[20] G. Klein and D. Murray, “Parallel tracking and mapping for small ar workspaces”, in *2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality*, 2007, pp. 225–234. DOI: `10.1109/ISMAR.2007.4538852`.

[21] C. Kolhatkar and K. Wagle, “Review of slam algorithms for indoor mobile robot with lidar and rgb-d camera technology”, in *Innovations in Electrical and Electronic Engineering*, M. N. Favorskaya, S. Mekhilef, R. K. Pandey, and N. Singh, Eds., Singapore: Springer Singapore, 2021, pp. 397–409, ISBN: 978-981-15-4692-1.

[22] “Nvidia jetson agx xavier”. (), [Online]. Available: `https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit` (visited on 10/04/2021).

[23] H. Oleynikova, Z. Taylor, M. Fehr, R. Siegwart, and J. Nieto, “Voxblox: Incremental 3d euclidean signed distance fields for on-board mav planning”, in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017, pp. 1366–1373. DOI: `10.1109/IROS.2017.8202315`.

[24] “Opencv about page”. (), [Online]. Available: `https://opencv.org/about/` (visited on 09/16/2021).

[25] Y. Pan, X. Xu, X. Ding, S. Huang, Y. Wang, and R. Xiong, “Gem: Online globally consistent dense elevation mapping for unstructured terrain”, *IEEE Transactions on Instrumentation and Measurement*, vol. 70, pp. 1–13, 2021. DOI: `10.1109/TIM.2020.3044338`. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/9293017`.

[26] Y. Pan, X. Xu, Y. Wang, X. Ding, and R. Xiong, “Gpu accelerated real-time traversability mapping”, in *2019 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, 2019, pp. 734–740. DOI: `10.1109/ROBIO49542.2019.8961816`. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/8961816`.

[27] *Robotic operating system*. [Online]. Available: `https://www.ros.org` (visited on 09/24/2021).

[28] “Roscomponentens”. (), [Online]. Available: `https://www.roscomponents.com/en/` (visited on 07/26/2021).

[29] R. B. Rusu and S. Cousins, “3D is here: Point Cloud Library (PCL)”, in *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 2011. [Online]. Available: `https://pointclouds.org/`.

[30] T. Schops, T. Sattler, and M. Pollefeys, “Bad slam: Bundle adjusted direct rgb-d slam”, in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2019. [Online]. Available: `https://openaccess.thecvf.com/content_CVPR_2019/html/Schops_BAD_SLAM_Bundle_Adjusted_Direct_RGB-D_SLAM_CVPR_2019_paper.html`.

[31] C. Suo, W. Xu, Y. Guan, L. He, and Y. Yang, “Real time obstacle avoidance and navigation with mobile robot via local elevation information”, in *2019 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, 2019, pp. 2896–2901. DOI: `10.1109/ROBIO49542.2019.8961624`. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/8961624`.

[32] “T265”. (), [Online]. Available: `https://www.intelrealsense.com/tracking-camera-t265/` (visited on 10/04/2021).

[33] I. Vasiljevic, N. I. Kolkin, S. Zhang, *et al.*, “DIODE: A dense indoor and outdoor depth dataset”, *CoRR*, vol. abs/1908.00463, 2019. arXiv: `1908.00463`. [Online]. Available: `http://arxiv.org/abs/1908.00463`.

[34] B. Younis and N. Salim, “Hardware implementation of 3d-bresenham’s algorithm using fpga”, pp. 37–47, Mar. 2013.

[35] J. Zeng, Y. Tong, Y. Huang, *et al.*, “Deep surface normal estimation with hierarchical rgb-d fusion”, in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 6153–6162.

[36] S. Zhang, L. Zheng, and W. Tao, “Survey and evaluation of rgb-d slam”, *IEEE Access*, vol. PP, pp. 1–1, Jan. 2021. DOI: `10.1109/ACCESS.2021.3053188`. [Online]. Available: `https://www.researchgate.net/publication/348670873_Survey_and_Evaluation_of_RGB-D_SLAM`.

[37] Q.-Y. Zhou, J. Park, and V. Koltun, “Open3D: A modern library for 3D data processing”, *arXiv:1801.09847*, 2018. [Online]. Available: `http://www.open3d.org/`.

A. Appendix

List of Figures

List of Tables

A.1. Used Noise Parameters

- normal_factor_a: 0.000611
- normal_factor_b: 0.003587
- normal_factor_c: 0.3515
- normal_factor_d: 0
- normal_factor_e: 1
- lateral_factor: 0.01576