Rheinische
Friedrich-Wilhelms-
Universität Bonn

Institut für Informatik
Abteilung VI
Autonome Intelligente Systeme

Rheinische
Friedrich-Wilhelms-Universität Bonn

Bachelorarbeit

Real-Time Dense 3D Scene Reconstruction: Fusing LiDAR Data into the TSDF

Author:
Moritz Miebach

Erstgutachter:
Prof. Dr. Sven Behnke

Zweitgutachter:
Dr. Jens Behley

Betreuer:
Malte Splietker

Datum: 28. Juli 2021

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen verwendet habe. Die Stellen der Arbeit sowie evtl. beigefügte Abbildungen, Zeichnungen oder Grafiken, die anderen Werken dem Wortlaut oder Sinn nach entnommen wurden, habe ich unter Angabe der Quelle kenntlich gemacht.

Ort, Datum	Unterschrift

Abstract

In robotics, real-time dense 3D scene reconstruction is an important task for a variety of applications, such as autonomous driving, grasping, or Augmented Reality (AR). An efficient and reliable representation of the environment enables us to implement these tasks.

RGB-D cameras have successfully been utilized to reconstruct indoor environments, using the dense Truncated Signed Distance Function (TSDF) to represent the scene. With LiDAR sensors becoming more affordable and gaining in popularity in autonomous driving and drones, 3D scene reconstruction using LiDAR data becomes a more important task. There exist few attempts to fuse LiDAR data into the TSDF in real-time.

In this thesis, I analyze the problem and present novel approaches for fusing LiDAR scans into the volumetric TSDF representation. The methods are evaluated on real-world datasets and compared to the related work.

Contents

1 Introduction

In robotics, an important task is to be able to perceive the environment, in order to interact with it. Ultrasonic sensors, radar, LiDAR sensors, and RGB-D cameras have been frequently used to perceive the environment in the field of robotics. Various concepts on how to represent a 3D environment have been proposed, such as point clouds, surfels, or volumetric representations, in particular, the voxel-based Truncated Signed Distance Function (TSDF). The voxel-based TSDF representation is a discretization of the 3D space into a regular grid of so-called voxels, where each voxel stores the distance to the closest surface of an object. It has gained popularity since the introduction of KinectFusion [20], the first real-time capable TSDF tracking and mapping method. This is mainly due to the availability of affordable sensors, such as the Microsoft Kinect and later the Intel RealSense.

A 3D representation of the environment can be useful for a variety of tasks, such as localization, Augmented Reality (AR), or interaction with the environment. The TSDF can be used for all these tasks, as it allows for direct wire mesh generation, scene rendering, and ICP localization.

Most applications that fuse data into a dense volumetric scene, however, are limited to the use of depth images as input, such as KinectFusion [20] or InfiniTAM [16]. These approaches are often designed to map smaller indoor scenes. LiDAR sensors, on the other hand, are more suited for outdoor scenes, as they have greater accuracy, a 360-degree horizontal view angle, and a significantly higher range. Autonomous driving and other robotics applications, such as simultaneous localization and mapping (SLAM), or planning often require large-scale 3D scene reconstruction. Employing LiDAR sensors to obtain measurements of the environment is suitable for such large-scale mapping.

There are multiple approaches to scene reconstruction using LiDAR data, the simplest being the accumulation of point clouds. However, the accumulation of point clouds scales linearly with the number of scans, which makes further processing inefficient. Other methods use surfels [26] or the TSDF to aggregate multiple input points and jointly represent parts of the environment.

A major task of dense scene reconstruction using the TSDF is to fuse a given stream of input scans into a dense representation. There are only a few approaches, that fuse LiDAR data into the TSDF representation. The sparsity of the LiDAR data, as compared to depth images from RGB-D sensors, makes it challenging to fuse the points into the TSDF: the representation will end up incomplete, which makes mesh extraction or tracking difficult.

In this thesis, I will present three novel approaches to fuse LiDAR-data into the TSDF and will compare them to existing approaches. In particular, I will present approaches that fuse point clouds, that are generated by a rotating LiDAR sensor, into the TSDF. The first approach will be based on casting a ray from the sensor to each detected point of the recorded LiDAR scan. The second approach will make use of the approximated local surface normals. Finally, an approach is proposed that employs an intermediate step, in which the LiDAR data is converted into a surfel map. This surfel map is used to reconstruct the scene in the TSDF representation.

2 Related Work

The first approaches for dense 3D scene reconstruction fusing depth images into the TSDF were proposed by Curless et al. [5]. The approaches by Curless et al. [5], Hilton et al. [14], and Wheeler et al. [29] convert depth maps into signed distance fields, which are then averaged into a voxel grid.

More recently, with Microsofts' KinectFusion [20], dense 3D scene reconstruction has had a breakthrough. Using a general purpose GPU (GPGPU), to parallelize tracking and mapping operations, a real-time dense 3D scene reconstruction, that makes use of a voxel-based TSDF representation, has emerged. However, KinectFusion employs a regular voxel grid, resulting in large memory consumption, as both empty space and surface are represented densely [21]. Therefore larger scenes cannot be represented.

This problem is addressed by Keller et al. [17] using a point-based representation. Roth [25] and Whelan et al. [30] stream voxels out of the GPU based on camera motion. Furthermore, Whelan et al. propose a dense mesh-based approach, Kintinuous [31], that extends the KinectFusion approach so that larger environments can be reconstructed. This is achieved by allowing the space used for mapping to vary dynamically, extracting dense point clouds where the volume is left, and incrementally adding the resulting points to the triangular mesh [31].

Nießner et al. [21] propose an approach of voxel hashing, thus only storing data densely in cells where measurements have been made. Data is integrated into the TSDF by using Raycasting, as proposed in [28].

Based on the work of Neißner et al. [21], InfiniTAM [16] provides an optimization of [21] and [20], resulting in a real-time 3D scene reconstruction that works on mobile devices. Furthermore, Dryanovski et al. [9] propose an approach for real-time indoor 3D scene reconstruction on mobile devices. However, [16], [20], [9], and [21] only consider depth images as input.

In recent years LiDAR sensors have become more popular, as they are frequently used in disciplines such as autonomous driving. Caminal et al. [3] provide an initial approach for fusing LiDAR data into the TSDF by converting the LiDAR data into a depth image and using interpolation to obtain a dense representation. Based on Nießner et al. [21], Kühner et al. [18] project the data of a rotating 360-degree-LiDAR sensor onto a cylindrical depth image, which is used to fuse LiDAR data into the TSDF.

A different approach is proposed by Roldão et al. [24] who make use of primitive local surfaces, that approximate the scene. Even though this approach achieves good results, it does not run in real-time.

A method to render a mesh from a 3D dense scene reconstruction using a voxel-based TSDF representation is the Marching Cubes Algorithm [19], which is provided in the implementation of [16]. Further, the TSDF is used by many SLAM systems, such as [16], [20] or [18], as its properties for localization may be used to track the sensor.

3 Fundamentals

3.1 LiDAR

LiDAR (Light Detection and Ranging or Light imaging, Detection and Ranging) is a method to collect depth data from a 3-dimensional scene. It is often used for autonomous driving and drones (UAV and MAV). There are two main types of methods to collect LiDAR data from a scene. (i) The Pulse-Based LiDAR and (ii) The Phase-Based LiDAR.

The functionality of a Pulse-Based LiDAR system is usually based on the Time of Flight (ToF) of laser impulses, that are emitted onto an object (Fig. 3.1(a)). Using the time the laser takes to arrive back at the sensor, the distance to the detected point in space can be calculated relative to the sensor

$$d = \frac{ct}{2}, \tag{3.1}$$

where d is the distance between the detected object and the sensor, t is ToF and c is the speed of light [15].

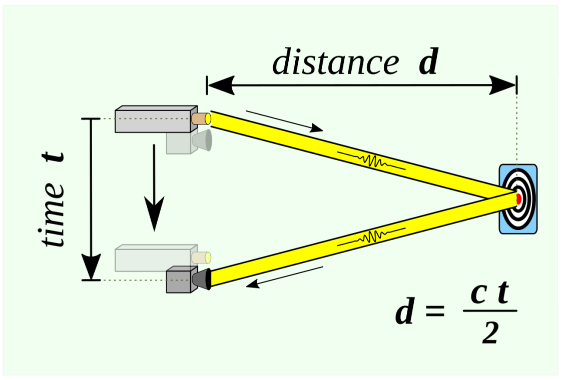
The Phase-Based LiDAR makes use of a continuous laser signal and modifies the amplitude of the laser signal [15], as shown in Fig. 3.1(b). The difference between the transmitted and received signal can be calculated by

$$\Delta\phi = 2\pi f t = 2\pi f \left(\frac{2r}{c}\right), \tag{3.2}$$

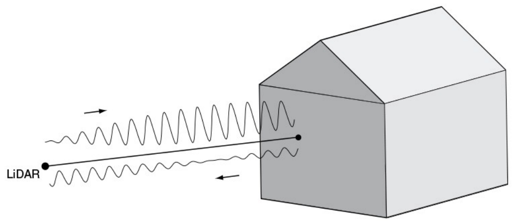
where f is the frequency. The distance to the measured point can therefore be calculated as

$$\frac{\Delta\phi c}{4\pi f} \tag{3.3}$$

However, the phase difference lies between 0 and 2π, causing ambiguity of the measured distance. Therefore multiple frequencies are employed: A higher frequency to increase the ranging and a lower frequency to increase maximum unambiguity distance [2].



(a)

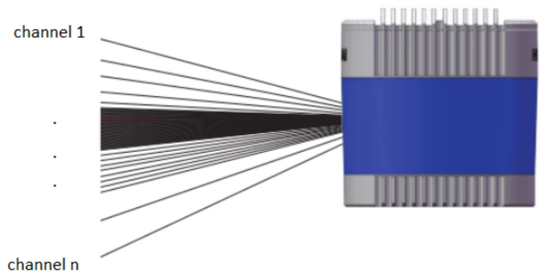


(b)

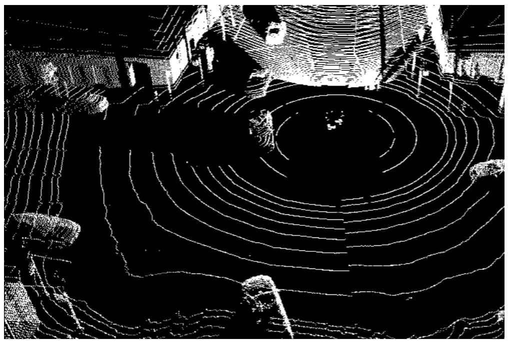
Figure 3.1: (a) Visualization of the Pulse-Based LiDAR procedure [32] and (b) visualization of the Phase-Based LiDAR procedure [27].

In recent years more affordable LiDAR technology has emerged, encouraging the use of LiDAR for various tasks.

LiDAR scanners used for autonomous driving and drones often consist of several sensors (Fig. 3.2(a)) and are rotating around an axis. This creates a point cloud, composed of several so-called scanlines or channels. Each scanline is created by a sensor that emits a laser onto the surface of the scene with a given angle relative to the heading of the sensor. The resulting scene can then be represented as a point cloud (Fig. 3.2(b)).



(a)



(b)

Figure 3.2: (a) LiDAR scanner with n channels, (b) resulting point cloud, taken from the PandaSet [1].

3.2 The TSDF representation

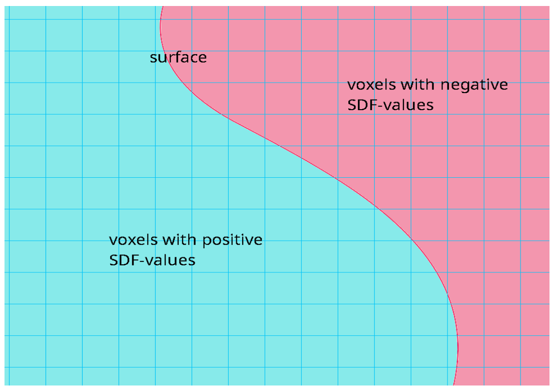
A Signed Distance Function (SDF) is an implicit function, often used in the context of scene reconstruction or the representation of isosurfaces. The SDF has positive values on the exterior of a surface and negative values on the interior of the object, as illustrated in Fig. 3.3(a). For a scene $S \subset \mathbb{R}^3$, with Euclidean metric, the SDF can be defined formally as follows:

Considering a subset $O \subset S$, O being the combination of the volume of all objects in the scene, the SDF-value, $F(x)$, for a point $x \in \mathbb{R}^3$, is defined by

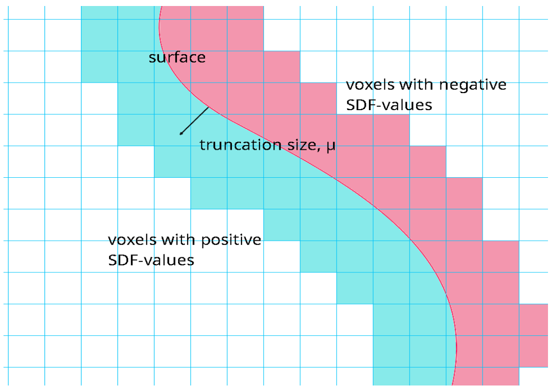
$$F(x) = \begin{cases} d(x, \partial O), \; x \in O^c \\ -d(x, \partial O), \; x \in O \end{cases}, \tag{3.4}$$

where ∂O denotes the boundary of O and $d(x, \partial O) := \inf_{y \in \partial O} d(x, y)$. $d(x, y) := \|x - y\|_2$ is the Euclidian distance between x and y. The SDF-value, therefore, defines the minimum distance from a point $x \in \mathbb{R}^3$ to a surface of the nearest object in the scene. The SDF is often combined with a voxel-based representation in order to discretize the continuous scene S.

The Truncated Signed Distance Function (TSDF) is a signed distance function that only considers the areas in a given truncation band around the surface, represented by the black arrow in Fig. 3.3(b).


surface
voxels with negative SDF-values
voxels with positive SDF-values

(a)


surface
voxels with negative SDF-values
truncation size, μ
voxels with positive SDF-values

(b)

Figure 3.3: (a) Voxel-based SDF representation of a scene, where the red line represents the surface of an object, the red voxels the inside of an object, and the blue voxels the outside of an object (b) Voxel-based TSDF representation, where the black arrow denotes the size of the truncation band.

This representation has the advantage, that the required memory only grows proportional to the surface area of O, rather than to the volume of the entire scene. As usually in scene reconstruction only the isosurface is required, the TSDF representation, therefore, is a more effective representation than the SDF in terms of memory consumption.

The SDF-value or distance value in the TSDF, $F(x)$, is normalized with respect to the truncation size, so that $F(x) \in [-1, 1]$ for all x. Likewise, the TSDF is often used in combination with a voxel representation of the scene, as shown in Fig. 3.3(b).

Furthermore, when using the TSDF representation it is common to allocate the SDF-value, $F(v)$ and corresponding weight, $W(v)$, to each voxel, v, that is used, as proposed in [5]. The SDF-value indicates the distance to the nearest surface, the weight indicates the likelihood that the SDF-value of the voxel is correct.

In scene reconstruction, a sequence of input data is read into the reconstruction system. Therefore it is necessary when using a voxel-based TSDF representation to update the voxels. As proposed by [5] and used in [20], [21] and [16], a common approach is to cumulatively average the SDF-value and the weight in each updated voxel.

$$F(v)_i = \frac{W(v)_{i-1}F(v)_{i-1} + W(v)F(v)}{W(v)_{i-1} + W(v)} \tag{3.5}$$

$$W(v)_i = W(v)_{i-1} + W(v) \tag{3.6}$$

We denote $F(v)_{i-1}$ as the current TSDF-value and $W(v)_{i-1}$ as the current weight for a voxel, v. $F(v)_i$ and $W(v)_i$ denote the SDF-value and the weight of the new data for v respectively.

In recent years, with parallelized GPU computation, the TSDF representation has often been used for real-time 3D scene reconstruction from depth images, as it can be used well for localization and mapping, [20],[21], [16].

Further [21] and [16] employ an additional technique, voxel block hashing, to further reduce memory consumption. InfiniTAM's [16] approach on voxel block hashing will be employed in this thesis and will be discussed in more detail in section 5.1.1.

3.3 Principal Component Analysis

Principal Component Analysis (PCA) is a data-driven procedure, that is commonly used to simplify a given set of correlated data by finding a set of orthogonal principal components, $\{u_1, \ldots, u_k\}$. For a point cloud of n data points in the $\mathbb{R}^n$, the PCA procedure can be summarized, as elaborated in [11], as follows:

We are first given a set of m data points, $x_i \in \mathbb{R}^n$. This can be organized in a $n \times m$-matrix, X. In a pre-processing step, the mean for each data component is calculated.

$$\bar{x}_j = \frac{1}{m} \sum_{i=1}^{n} X_{ij} \tag{3.7}$$

The given data points will be centered using the mean.

$$z_j = x_j - \bar{x}_j \tag{3.8}$$

The centered data is used to compute the centered data matrix.

$$Z = [z_1 \dots z_m] \tag{3.9}$$

The covariance matrix, C, can then be calculated as follows:

$$C = ZZ^T \tag{3.10}$$

The first principal component, u_1 is given by:

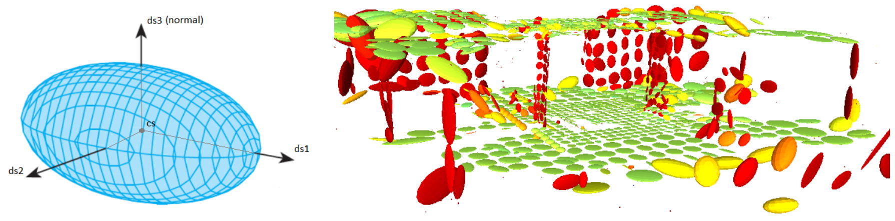
$$u_1 = \underset{\|u_1\|=1}{\arg\max}\, u_1^T ZZ^T u_1, \tag{3.11}$$

which is the eigenvector of C, which corresponds to the largest eigenvalue. The other principal components can be computed, by finding the eigenvectors of the matrix C and sorting them according to the magnitude of their eigenvalues. The eigenvectors with larger eigenvalues will portray data with larger variance. 3-dimensional PCA is often used to compute surfels from point clouds.

3.4 Surfel

In the scope of this thesis a surfel (surface element) is defined by an ellipsoid, s, with a center $c_s = (c_{s_1}, c_{s_2}, c_{s_3}) \in \mathbb{R}^3$, and three direction vectors, $d_{s_1}, d_{s_2}, d_{s_3} \in \mathbb{R}^3$ indicating the orientation of the surfel, as shown in Fig. 3.4(a). Fig. 3.4(b) visualizes a reconstructed scene using a surfel map.

A surfel is generated by combining all input points p_i of a given point cloud, $\mathcal{P}$, that lie in a coherent subset $V \subset \mathbb{R}^3$. This is done by applying a 3-dimensional PCA to the points $p_i \in V$, thus computing three orthogonal directions of maximum variance, which are defined by the three eigenvectors $d_{s_1}, d_{s_2}, d_{s_3}$ of the covariance matrix [8]. The direction of the smallest variance is used as the normal of the surfel. Sufels are commonly used for rendering the surface of volumetric data and for 3D scene reconstruction.



(a)

(b)

Figure 3.4: (a) Visualization of a surfel, which is composed of the three eigenvectors of a resulting PCA and (b) reconstructed scene using surfels [8].

3.5 MRSMaps

The MRSMap (Multi-Resolution Surfel Map) proposed by Droeschel et al. [8] is a 3D map, consisting of surfels, which are generated from a point cloud, $\mathcal{P}$. The point cloud is divided into different levels. These levels increase with increasing distance to the sensor origin. Each level consists of voxels, whose size increases with increasing level, as shown in Fig. 3.5, thus creating a multi-resolution voxel grid. In order to generate surfels, PCA is applied to all points within each voxel. Thus resulting in larger surfels farther away from the sensor and smaller surfels closer to the sensor.

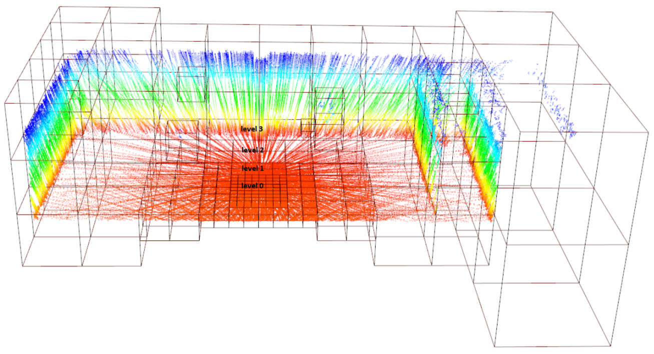


Figure 3.5: Visualization of the multiresolution grid on a point cloud[8].

4 Approaches for Fusing LiDAR into TSDF

4.1 Interpolation - Caminal et al.

One of the first approaches fusing LiDAR data into the TSDF representation was proposed by Caminal et al. [3]. The 3D-LiDAR data is first projected into a given camera, where it is converted into the PNG format. The respective depth values of each pixel are then calculated. Caminal et al. [3] consider two approaches to fuse LiDAR into the TSDF.

The first approach uses Kintinous [31], a successor of KinectFusion [20]. The converted LiDAR data is scaled by a factor of 0.05 as the maximum range of the LiDAR is theoretically 120m. Thus, the side length of the model that can be represented by Kintinous is set to 6m. The resulting depth images are then fused into the reconstruction pipeline of Kintinuous [31].

The second approach uses an RTAB-Map (Real-Time Appearance-Based Mapping) approach.

As the LiDAR data is sparse, Caminal et al. propose a post-processing step, where the ”holes” in the depth image are inpainted by using the 8- connectivity version of a morphological interpolation technique [4], thus creating infrared data, that is similar to a dense raster image. The morphological interpolation technique preserves the original infrared values of the projected points, as shown in Fig. 4.1(a). This approach, however, often results in an incomplete representation of the scene, as shown in Fig. 4.1(b).

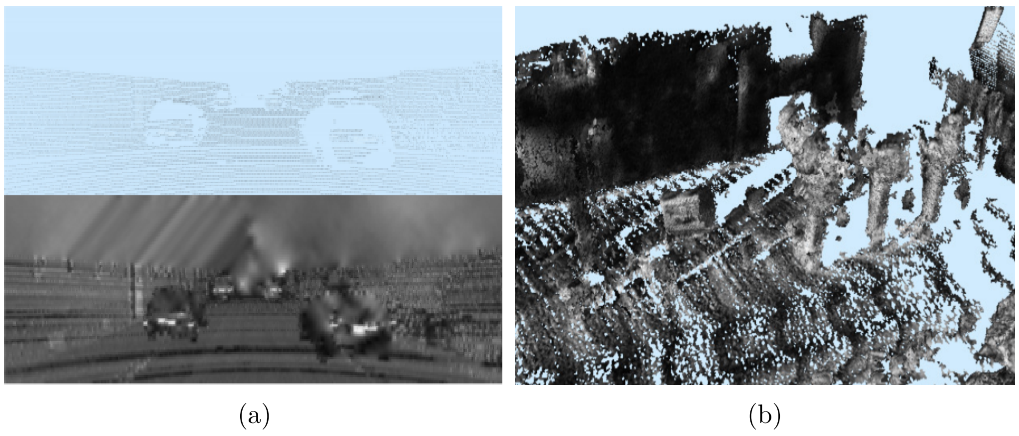


Figure 4.1: Visualization of (a) the conversion of LiDAR data (top) to a depth image (bottom) and (b) the resulting TSDF on the KITTI dataset [13].

4.2 Cylindrical Projection - Kühner et al.

Similar to the work of Caminal et al. [3], Kühner et al. [18] propose a method that converts the LiDAR data into a depth image format and then fuses the depth images into the TSDF. Furthermore, their work considers aspects when dealing with a rotating LiDAR sensor, as they are commonly found on vehicles.

For each timestep a new point cloud $\mathcal{P}_i = \{p_1, \ldots, p_p\}_i$ of LiDAR data is determined and converted into a depth image using a projection on a cylindrical depth image. This idea resembles the rotation of the LiDAR sensor. The plane of the depth image is wrapped around the rotation axis of the LiDAR sensor. For each point $p = [x, y, z]^T \in \mathcal{P}$ in the sensor coordinates, a pair of pixel coordinates, $u = [u, v]$ is computed as follows.

First the center of projection, c, is placed to the point on the rotation axis of the LiDAR sensor, which minimizes the squared distances to all rays, thus simulating the viewpoint. The offset from the center of the sensor to the center of projection is denoted by c_z. c_u and c_v denote the horizontal and vertical center of the depth image respectively. Now the pixel positions are calculated [18]:

$$\pi(x) = \left[c_u - n_u(1 - \frac{\varphi}{2\pi}), c_v - \frac{(z - c_z) f_r}{\rho} \right]^T \tag{4.1}$$

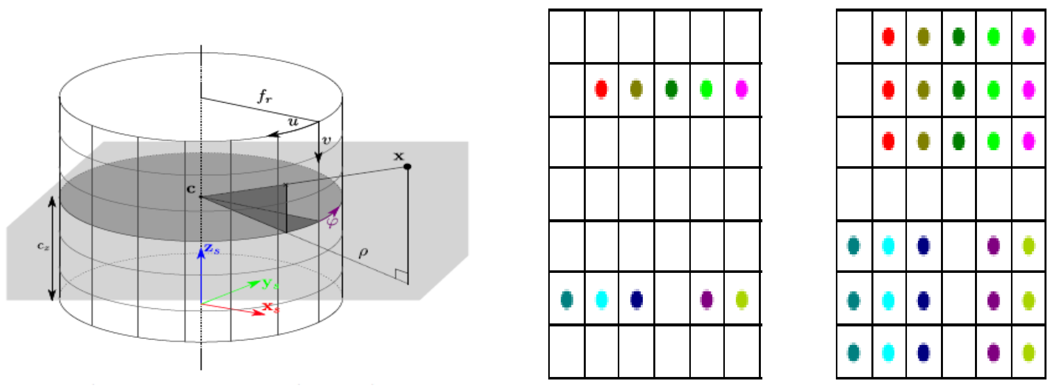
$$\rho = \sqrt{x^2 + y^2} \tag{4.2}$$

$$\varphi = \arctan 2(y, x), \;\; \varphi \in [0, 2\pi), \tag{4.3}$$

Here, ρ is the projection of the point onto the xy-plane, φ is the angle of the point on the xy-plane, and n_u is the number of columns of the depth image. The number of columns is the same as the horizontal resolution of the LiDAR sensor so that all points in the point cloud $\mathcal{P}_i$ approximately project into the horizontal center of a pixel. The number of rows is chosen according to the number of columns, in such a way that the pixels are approximately square [18].

Fig. 4.2(a) illustrates the process of projecting the point cloud onto a cylindrical image.

This method, however, leaves most pixels empty. Therefore the measurements of the filled pixels are assigned to the empty neighboring pixels in the same column within a certain range, which approximately corresponds to half of the gap between two neighboring projected scanlines, as shown in Fig. 4.2(b).



(a) (b)

Figure 4.2: Visualization of (a) the projection of a point cloud onto a cylindrical image and (b) the filling of neighboring pixels.

Kühner et al. [18] utilize voxel hashing from [21] for efficient memory consumption and parallel GPU implementation for a fast fusion of data.

To avoid missing voxels at a greater distance, all voxels within the viewing frustum and the truncation band of each pixel are allocated in the hash table.

The SDF-value and the weight of each allocated voxel in the hash table are computed to update the voxels in the TSDF. Here, he calculation of the weight, $W(x)$, for an allocated voxel follows the principles of [12], hence the weight is decreasing if the voxel is farther away from the sensor. All allocated voxels are then updated using the update rule (3.5) and (3.6), proposed by [6].

The approach proposed by Kühner et al. [18] shows good results, as it reconstructs the scene well and runs in real-time. However, a preprocessing step is needed to convert the LiDAR data into a cylindrical depth image. Furthermore, the vertical intersection of each point in a pixel of the depth image is not centeres due to rounding. This may lead to problems in the TSDF and is discussed in more detail in section 7.6.

4.3 Probablistic Scene Reconstrucion - Roldao et al.

A completely different approach was proposed by Roldão et al. [24]. Unlike the work of [3] and [18], Roldão et al. uses primitive local surfaces to approximate complex environments.

A voxel representation, with voxel size α is used to efficiently update the representation of the point cloud $\mathcal{P} \rightarrow \{V^1, V^2, \ldots, V^n\}$. Each voxel, V, stores the number of points, $|V|$, lying inside the voxel, the mean, V_μ and the covariance V_σ. Local planar surfaces are then computed from the LiDAR data.

To overcome the heterogeneous density of LiDAR data, an adaptive neighborhood is used, which both increases the statistical robustness and counterbalances the lack of local data due to low density or occlusion [24]. The adaptive neighborhood is defined as a multi-scale neighborhood at the location of vertices between voxels. For a vertex v the first neighborhood level is composed of the 8 neighboring voxels of the vertex and is denoted by $\mathcal{H}^1(v)$. The second level, $\mathcal{H}^2(v)$, is composed of the neighbors of the first level. One can now obtain the cardinal, $|\mathcal{H}|$, the statistical mean, $\mathcal{H}_\mu$, and the covariance, $\mathcal{H}_\sigma$ inside the neighborhood. To further obtain an estimation of a local plane of the voxels $\mathcal{H}$, the covariance $\mathcal{H}_\sigma$ in combination with Principal Component Analysis (PCA) is used, if a given number of LiDAR points are present:

$$|\mathcal{H}| \geq N_{min}, \tag{4.4}$$

where N_{min} is a hyperparameter [24].

The plane is calculated using the resulting eigenvectors, $(\overrightarrow{e_1}, \overrightarrow{e_2}, \overrightarrow{e_3})$, and eigenvalues, $(\lambda_1, \lambda_2, \lambda_3)$, with $\lambda_1 \geq \lambda_2 \geq \lambda_3$. As e_3 is the least dominant eigenvector, it is used to define the unoriented normal of the plane. The direction towards the sensor, s_p, will be used as orientation of the normal, $\overrightarrow{n}$:

$$\overrightarrow{n} = \begin{cases} \overrightarrow{e_3} & if\ \overrightarrow{e_3} \cdot (s_p - \mathcal{H}_\mu) > 0 \\ -\overrightarrow{e_3} & else \end{cases} \tag{4.5}$$

The resulting local plane is defined by the normal and the statistical mean: $\Pi = (\overrightarrow{n}, \mathcal{H}_\mu)$.

In order to reconstruct the global continuous surface, the TSDF is computed for each vertex $v \in V$. Hereby an optimal neighborhood is computed and the TSDF value is estimated for each vertex. This is done by calculating the projection, w^k, of the vertex, v, onto the local plane, $\Pi^k(v)$, and evaluating the likelihood of this projection belonging to the Gaussian, $\mathcal{N}^k$, where $\mathcal{N}^k$ is the 2D planar-Gaussian of the statistical distribution of $\mathcal{H}^k$ projected onto Π^k [24].

The optimal neighborhood-level k^* is defined as the smallest level for which the projection of v onto Π^k has a probability of belonging to the Gaussian $\mathcal{N}^k$ greater than a hyperparameter, τ [24]. Formally it is the smallest integer for which the probability density function $\mathcal{N}_{PDF}$ satisfies 4.6.

$$\mathcal{N}^k_{PDF}(w^k \mid \mathcal{H}^k_\mu, \Sigma) \geq \tau, \;\; \Sigma = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \tag{4.6}$$

The calculation is done iteratively, starting with level one and stopping when 4.6 is satisfied. Furthermore, the level size is bounded by k_{max} to avoid exponential computation time. The TSDF value is computed as follows:

$$TSDF(v) = \overrightarrow{n^{k'}}(v - \mathcal{H}^{k'}_\mu), \tag{4.7}$$

where $\overrightarrow{n^{k'}}$ is the approximated normal and $\mathcal{H}^{k'}_\mu$ is the statistical mean [24].

This approach gives good results in terms of accuracy, however, it is not able to perform in real-time.

5 Methodology

In this thesis, I will be making use of the software provided by InfiniTAM [16] to fuse LiDAR data into the TSDF. This software will serve as the framework for my implementation.

5.1 InfiniTAM as Framework

InfiniTAM [16] is a SLAM system, that fuses depth images into the TSDF representation. It is based on the KinectFusion [20] approach to fuse data into the TSDF. However, the InfiniTAM system employs the concept of voxel block hashing, similar to [21], thus making the data structure and memory allocation more efficient. Overall, the InfiniTAM framework is composed of three major steps: tracking, fusion, and rendering, as shown in Fig. 5.1.

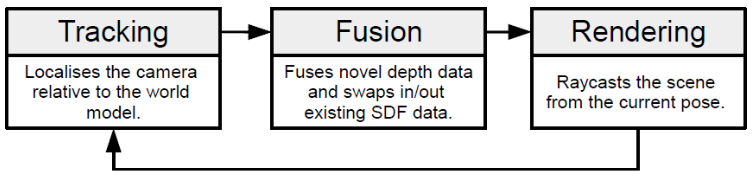

Tracking
Localises the camera relative to the world model.
Fusion
Fuses novel depth data and swaps in/out existing SDF data.
Rendering
Raycasts the scene from the current pose.

Figure 5.1: Processing steps of InfiniTAM [16].

In this thesis, I will only consider the fusion and use the rendering to visualize my results. The poses of the sensor will be provided by the used datasets. In order to fuse new data into the TSDF, I will use the software provided by InfiniTAM [16]. The main concepts of [16], that are employed in my work are the voxel block hashing and the rendering.

5.1.1 Voxel Block Hashing

The hashing operates using voxel blocks, that are composed of blocks of $8 \times 8 \times 8$ voxels. These voxel blocks have a certain position, (b_x, b_y, b_z), that is used to calculate the corresponding hash index as follows:

$$h = ((b_x) \times P_1) \oplus (b_y \times P_2) \oplus (b_z \times P_3)) \mod K, \tag{5.1}$$

where mod and $\oplus$ are the modulo and the logical XOR operators respectively. In this case (P_1, P_2, P_3) are the prime numbers $(73856093, 19349669, 83492791)$ and K is the number of buckets in the hash table, h [16]. The concept of closed hashing is employed to deal with resulting hash collisions. A so-called chained table with list head cells is employed.

The operations InfiniTAM employs on the hash table when new depth images are integrated into the TSDF are (i) the retrieval of needed voxel blocks, (ii) the allocation and insertion of new voxels blocks, and (iii) the deletion of hash table entries.

The retrieval returns the desired voxel block by computing the hash index, h, and iterating through its list until the voxel block is found. Therefore, the hash table lookup usually takes constant time.

When fusing a new depth image into the TSDF, the hash value of all voxel blocks, that lie on the camera's line of sight within the depths $d - \mu$ and $d + \mu$, is computed. Here, d is the depth of the pixel and μ is the truncation size of the TSDF. The hash table is updated to incorporate these voxel blocks. Thus new hash entries are allocated for each voxel block, where new data has been detected.

The allocation is implemented in parallel on a given number of threads. If multiple threads require an update in the same hash bucket, InfiniTAM chooses one at random for the update.

If there is no need to store parts of the data in the hash table, deletion of hash table entries can be used to swap out data, into long-term storage.

5.1.2 Fusion

New data of depth images is integrated in four steps: (i) Allocation, (ii) Visible list update, (iii) Camera data integration, and (iv) optionally swapping out data. An overview of the fusion-pipeline is given in Fig. 5.2.

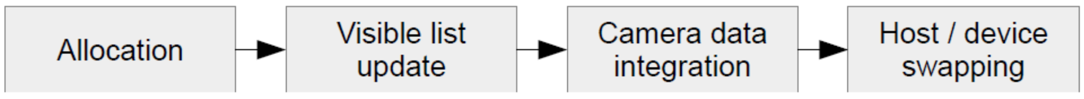


Figure 5.2: Overview of the fusion process [16].

In the allocation step, new hash and voxel block entries are created from the integrated data.

The visible list is used to keep track of all visible voxel blocks from the camera. This enables us to consider only the relevant parts of the scene in the hash table for data integration.

The camera data integration operates by transforming the location of each voxel into the camera coordinate system $X_d = R_d X + t_d$, using the rotation matrix $R_d \in \mathbb{R}^{3\times3}$ and the translation $t_d \in \mathbb{R}^3$ of the camera pose. The projection of the camera coordinates into a depth image, I_d, is computed by using its intrinsic parameters $K_d \in \mathbb{R}^{3\times3}$

$$\eta = I_d(\pi(K_d X_d)) - X_d^{(z)}. \tag{5.2}$$

Here, π determines the inhomogeneous 2D coordinates from the homogeneous ones and the superscript (z) selects the Z-component of X_d [16]. If $\eta \geq -\mu$, the SDF-value is updated as

$$F(x) \leftarrow \frac{w(X)F(x) + \min(1, \frac{\eta}{\mu})}{w(X) + 1}, \tag{5.3}$$

where w is a field counting the number of observations in the running average [16]. Therefore, $F(x)$ holds an average of all fused points inside a voxel.

Optionally one can swap out data into long-term storage, in order to speed up the retrieval operation in the hash table. If this data is required in later fusion steps it can simply be swapped back in.

5.2 LiDAR in InfiniTAM

As I will focus on fusing 360-degree-LiDAR data into the TSDF using known poses, some adaptation has to the InfiniTAM software has to be made, as it only supports depth images as input.

Considering a sequence of point clouds $\mathcal{P}_i$, that was retrieved from a LiDAR sensor, I will present three approaches of fusing LiDAR data into the TSDF:

1. A Raycasting approach.
2. An approach that locally approximates the surface normal at each point in the point cloud by making use of an organized point cloud.
3. An approach that incorporates an intermediate step, in which the points of the point cloud are combined to surfels.

In the approaches that I present in this work, the retrieval of voxel blocks and the computation of the hash index is analog to [16]. However, if multiple points of the point cloud require an update of the same voxel, due to dense regions in the point cloud, InfiniTAM only uses one point to update the voxel [16]. I will combine the computed values of all points by computing the weighted average of all updates. Furthermore, InfiniTAM employs a weighted running average of the SDF-value for each voxel, based on the distance to the sensor [12]. I will adapt the computation of the weight depending on the method used, in order to determine the confidence of a computed SDF-value more precisely.

Similar to [16], there are three main steps when fusing LiDAR data into the TSDF. First, the LiDAR data, consisting of a point cloud and a pose, is read by the CPU. The data is then copied into the GPU and the host is updated. All needed voxel blocks are computed and new entries in the hash table are allocated by applying the hash function. Finally, the registered voxel blocks are updated in two steps. First, all threads of the parallel implementation on GPU will update all registered voxel blocks. This is done by finding the necessary voxels in each voxel block and updating them according to (3.5) and (3.6). Second, the updates are combined in order to obtain a single update for each updated voxel block.

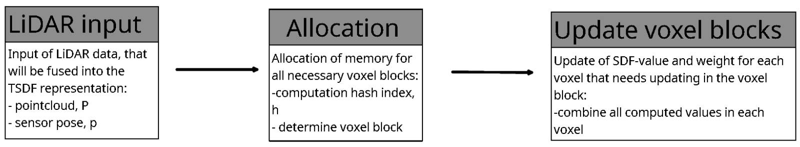

LiDAR input
Input of LiDAR data, that will be fused into the TSDF representation:
- pointcloud, P
- sensor pose, p
Allocation
Allocation of memory for all necessary voxel blocks:
-computation hash index, h
- determine voxel block
Update voxel blocks
Update of SDF-value and weight for each voxel that needs updating in the voxel block:
-combine all computed values in each voxel

Figure 5.3: Processing steps of LiDAR fusion in InfiniTAM.

The resulting TSDF can be visualized using the rendering component of InfiniTAM [16].

5.3 Raycasting

A naive approach to fuse LiDAR data into a voxel-based TSDF representation is by casting a ray, r, from the sensor pose to each point and calculating the respective TSDF-values for all voxels, that lie on this ray. In order to represent the TSDF realistically, only voxels that are within the truncation band and that the ray passes through should be allocated in the hash table.

Considering a point, p_{ij} of the point cloud, $\mathcal{P}_i = \{p_{i1}, \dots, p_{in_i}\}$, a cube ,$K$, around p_{ij}, of size $(\frac{2*\mu}{\kappa})^3$, can be defined. Hereby μ is the truncation band size and

κ is the voxel size. All voxels inside K, that r intersects, have to be considered and, therefore, all voxel blocks in which one of these voxels lie are allocated.

Whether a voxel is intersected by the ray can be calculated as follows. Given the sensor origin, o and a point, p_{ij}, the ray from o to p_{ij} is defined by $r(t) = o + t(p_{ij} - o)$, where t is a variable.

The distance from a point, p, to a line, L, with $L = r(t) = Q + t\overrightarrow{u}$ is given by

$$d(p, L) = \frac{\mid \overrightarrow{pQ} \times \overrightarrow{u} \mid}{\mid \overrightarrow{u} \mid}, \tag{5.4}$$

where $\times$ denotes the cross-product. The distance from a voxel center v to the ray, r, is therefore given by

$$dist(v_w, r) = \frac{\mid (o - v_w) \times (p_{ij} - o) \mid}{\mid (p_{ij} - o) \mid}, \tag{5.5}$$

where v_w denotes the world coordinates of the voxel center, v. A voxel lies on the ray from the sensor to the point, p_{ij} if

$$\sqrt{3\left(\frac{\kappa}{2}\right)^2} > dist(v_w, r). \tag{5.6}$$

$\sqrt{3(\frac{\kappa}{2})^2}$ is the maximum distance of a point in the voxel to its center. Therefore, all voxels that the ray passes through will be considered for allocation.

The calculation of the SDF-value for an allocated voxel is done by calculating the distance between the center of the given voxel, v, to the detected point, p_{ij}.

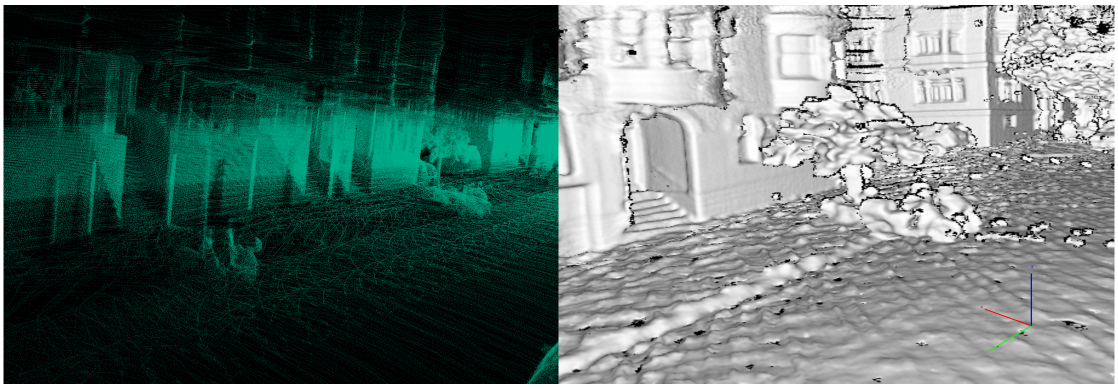
$$F_{RC}(v) = \|v_w - p_{ij}\| \tag{5.7}$$

The computation of the weight is chosen to decrease with increasing distance of p_{ij} to the sensor origin, o, as points that are detected farther away from the sensor are more likely to be less precise. The weight is computed by

$$W_{RC}(v) = \min\left(1, \frac{1}{d(o, p_{ij})}\right), \tag{5.8}$$

where $d(\cdot, \cdot)$ is the Euclidean distance.

This approach gives promising results, as shown in Fig. 5.4(b). However, the areas, in which the laser touches the surface with a very shallow angle may cause some issues with representation, such as holes or a noisy representation of the isosurface. As the sensor moves, allocated voxels may be overridden as the negative



(a) (b)

Figure 5.4: Visualization of (a) a combined point cloud of multiple LiDAR scans from PandaSet[1], (b) the rendering of the resulting TSDF of the PandaSet[1].

part of the previous LiDAR frame may be in the same location as the positive part of the current frame. This drawback will be discussed in section Section 7.3 in more detail.

To overcome this problem the normal of the local surface can be evaluated and the TSDF will be created using the approximation of the normal rather than using the ray from the sensor to the detected points.

5.4 Fusing into the TSDF using Local Surface Normals

As stated above the approach using Raycasting to compute the TSDF may cause some issues if the angle between the ray from the sensor to the detected point and the surface is very small. By computing the local normal at each point, one can utilize the voxels along the normal rather than those on the ray from the sensor to the detected point. This may lead to a more realistic representation of the TSDF, as can be seen in Fig. 5.5, where a model of the resulting voxel-based TSDF of the two approaches is visualized.

Fig. 5.5(a) represents the resulting TSDF when a raycast is used. Blue voxels denote voxels with positive SDF-values and red voxels the ones with negative SDF-values. In some cases, this may lead to an incorrect TSDF representation. Using the local surface normal, denoted as the blue arrow in Fig 5.5(b) may result in a more realistic representation of the TSDF, as shown in Fig 5.5(b).

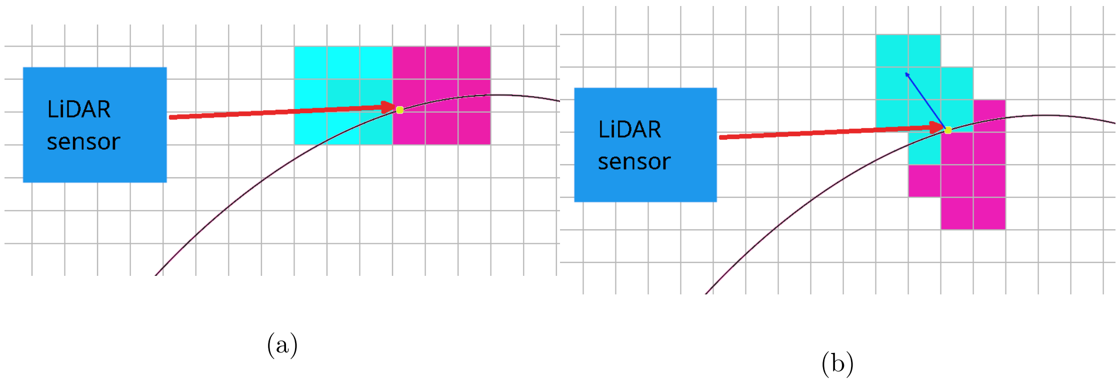


Figure 5.5: (a) Resulting TSDF when using Raycasting and (b) when using local surface normals.

5.4.1 Approximation of Local Surface Normals

A method to approximate the normal for each point in a point cloud is proposed by Holz [7]. The organized point cloud, $\mathcal{P}$, is traversed, in order to build a quad mesh by connecting every point, $p = \mathcal{P}(u, v)$, to its neighbours, $\mathcal{P}(u, v+1)$, $\mathcal{P}(u+1, v)$, and $\mathcal{P}(u+1, v+1)$, where $P(u, v)$ is the v-th point in the u-th scanline [7]. A new quad is only added to the mesh if the point $P(u, v)$ and its neighbors are valid and if the connecting edges between the neighboring points are not occluded [7]. After construction, the mesh is simplified by removing all vertices that are not used in any quad. To overcome the fact that one can often not deduce whether an edge is occluding due to sparsity of the point cloud, the validity depends on the length of the edge:

$$valid = (d_{i,j} \leq \epsilon_d^2), \tag{5.9}$$

$$with \;\; d_{i,j} = \|p_i - p_j\|^2, \tag{5.10}$$

where p_i and p_j are neighboring points in the quad mesh. The threshold ϵ_d is adapted to

$$\epsilon_d(d_i) = \begin{cases} \sqrt{2} d_i tan \Delta\theta & between\ the\ scan\ lines \\ \sqrt{2} d_i tan \Delta\phi & within\ the\ scan\ lines \end{cases}. \tag{5.11}$$

Here, $\Delta\phi$ is the difference in angle in the horizontal axis and $\Delta\theta$ is the difference in the vertical axis. Having calculated the quad mesh, the normal for each point can be determined by calculating the mean of all normals of the adjacent quads:

$$n_i = \frac{\sum_{j=0}^{N_T}(p_{j,a} - p_{j,b}) \times ((p_{j,a} - p_{j,c}))}{\| \sum_{j=0}^{N_T}(p_{j,a} - p_{j,b}) \times ((p_{j,a} - p_{j,c}))\|}, \tag{5.12}$$

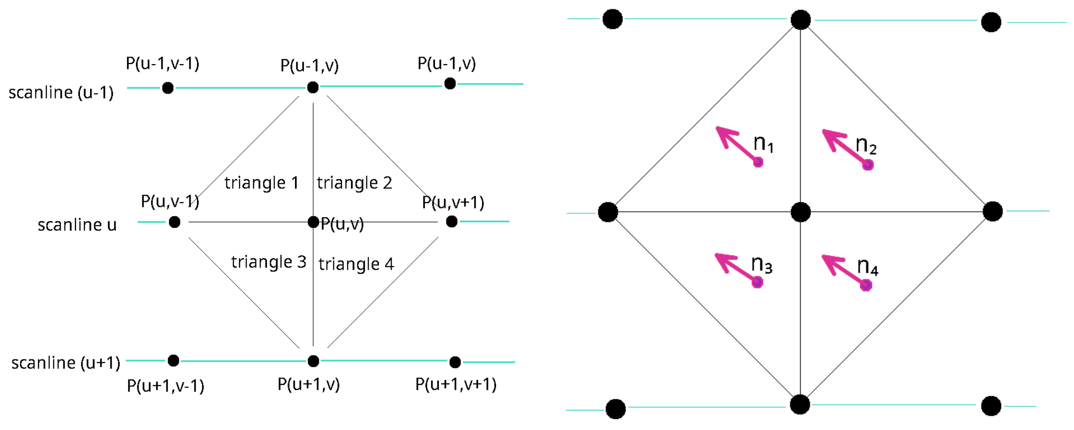
with face vertecies $p_{j,a}$, $p_{j,b}$, $p_{j,c}$ and neighborhood N_T.

5.4.2 Computation of Normal for the TSDF Fusion

Instead of computing a meshlike structure, I will consider the 4-Neighborhood of each point $p_i \in \mathcal{P}$ in an organized point cloud. I will use the adjacent triangles, shown in Fig. 5.6(a), to compute the normals, as shown in Fig. 5.6(b). The 4-Neighborhood for a point $p_i = P(u, v)$ is given by

$$N_T := \{p_1 = \mathcal{P}(u-1, v), p_2 = \mathcal{P}(u+1, v), p_3 = \mathcal{P}(u, v-1), p_4 = \mathcal{P}(u, v+1)\}$$

In order to compute the normal of $p_i \in \mathcal{P}$, we first have to check whether p_i is valid. p_i is valid if p_i and all four of its neighbours have been detected by the sensor. For each of the adjacent triangles of p_i the normal is computed, hence $n_{i_j} = (p_{i_j} - p_i) \times (p_{i_{j+1}} - p_i)$, where n_{i_j} is the normal of the j-th adjacent triangle of p_i, p_{i_j} is the j-th neighbor of p_i and $p_{i_{j+1}}$ is the next neighbor of p_i in clockwise direction. The normals are then normalized to obtain n_1, n_2, n_3, and n_4.



(a)

(b)

Figure 5.6: (a) Resulting triangles on neighboring points of the neighboring scanlines on which (b) the normals are computed.

Having computed the normals n_1, n_2, n_3, and n_4, it has to be checked whether the overall normal of the point makes sense.

If the angle between the normals of the adjacent triangles differs a lot, there is likely some sort of corner or edge present or the neighboring points of p_i belong to different objects. Therefore, one would only consider similar normals, as taking the average over all four normals could result in an unprecise approximation of the overall normal, leading to an inaccurate TSDF.

To avoid an unprecise approximation of the normal, the angle, $n_i \cdot n_j$, between all pairs (n_i, n_j) of normals are computed and it is checked whether these are similar. This is done by employing a threshold on $n_i \cdot n_j$. Tests have shown that 0.8, which allows approximately 36° of difference between two normals, is a suitable threshold for the difference in angle.

The subset of normals N_S, for which the angle between the pair of normals fulfills this criterion is used to calculate the overall normal of p_i. If none of the normals fulfill this criterion, the p_i is omitted.

$$\overrightarrow{n} := \sum_{j \in N_S} \frac{n_j}{\|n_j\|} \tag{5.13}$$

5.4.3 Allocation and Fusion of Points to the TSDF

Having computed the normals to each valid point, all voxels within the truncation band, whose distance to the normal is smaller than $\sqrt{\frac{3}{4}\kappa^2}$ should be considered for fusion. Thus, a voxel $v \in \mathbb{R}^3$ has to fulfill the two conditions

$$\frac{|(p_i - v_w) \times \overrightarrow{n}|}{|\overrightarrow{n}|} \leq \sqrt{\frac{3}{4}\kappa^2} \tag{5.14}$$

$$\frac{|p_i - v_w| \cdot \overrightarrow{n}}{|\overrightarrow{n}|} < \mu \tag{5.15}$$

to be updated. Here, v_w denotes the center of the voxel in world coordinates.

The SDF-value of a voxel, v is computed by finding the distance of the voxel to the plane, defined by the normal, $\overrightarrow{n}$ and scaling it according to the size of the truncation band, μ

$$F_{OP}(v) = \frac{|(p_i - v) \cdot \overrightarrow{n}|}{|\overrightarrow{n}|\mu} \tag{5.16}$$

The computation of the weight of each voxel, v, depends on how certain one is when allocating v. Therefore, I propose, that the weight depends on the difference in angle between the computed normals, $n_j \in N_S$, the distance of the detected point, p_i, to the sensor origin, and the angle at which the point is detected.

Unlike the method proposed in [7] the weight will not depend on the distance between the neighboring points, as this may result in the loss of information when the scanlines are farther apart. This for instance occurs when the street is detected when obtaining LiDAR data from a car.

For the first part of the weight the complement of the sum of the scaled angles between the normalized normals, n_j, and the overall normal, $\overrightarrow{n}$ is computed.

$$w_\theta := 1 - \sum_{j \in N_S} \left(\frac{\left| arccos \left(\frac{n_{i_j}}{\|n_{i_j}\|} \cdot \frac{\overrightarrow{n}}{\|\overrightarrow{n}\|} \right) \right|}{\pi} \right), \tag{5.17}$$

Here, N_S is the set of all valid neighbors of the detected point.

As the increasing distance to the sensor origin, o, of a detected point, p, increases the probability of p having a larger error, the normalized complementary distance is taken as another factor of the weight, similar to [12].

$$w_o := 1 - \frac{\|p - o\|}{\chi}, \tag{5.18}$$

Here, χ is the maximum range of the LiDAR sensor.

Furthermore, p_i is more likely to have been detected with less error if the angle between the normal, $\overrightarrow{n}$ and the ray from p to o is small. In terms of weighting, the normalized complementary angle is taken

$$w_{\theta_{\overrightarrow{n}}} := 1 - \frac{2 \left| \arccos \left(\frac{(o-p) \cdot \overrightarrow{n}}{\|o-p\| \|\overrightarrow{n}\|} \right) \right|}{\pi}, \tag{5.19}$$

as the angle between the normal and the ray will be in $[0, \frac{\pi}{2}]$. In total, we obtain a weighting for each voxel

$$W_{OP} := w_\theta w_o w_{\theta_{\overrightarrow{n}}} \tag{5.20}$$

This approach gives good results. However, especially in regions, where edges or other boundaries are present, the reconstruction of the scene shows some limitations. These limitations will be discussed in section 7.4.

5.5 Surfel Approach

As described in section 3.4, a surfel, S, is generated by accumulating points of a point cloud, $\mathcal{P}$, in a local neighborhood. The surfel is composed of a normal, n_S, and two other direction vectors, $d_{S,1}, d_{S,2}$, defining the plane, with a maximum variance of the points. The three direction vectors are the eigenvectors resulting from a 3-dimensional PCA and the corresponding eigenvalues determine the length of the corresponding eigenvector. The following approach will make use of MRSMaps, as described in section 3.5 as a preprocessing step.

Once the MRSMap is computed from a given point cloud, one can obtain the TSDF, by allocating values to the voxels around each surfel of the MRSMap.

5.5.1 Using a Cylinder

One approach to allocating values to the TSDF using a surfel map is to consider the voxels, that are located inside an imaginary cylinder around each surfel, S_i. The radius of the cylinder is chosen as the size of the vector pointing into the direction of maximum variance, $d_{S,1}$. The center of the cylinder is chosen as the center of the surfel, as shown in Fig. 5.7(a). When values are allocated to the TSDF, one only needs to consider voxels that are within the truncation band. The cylinder's height is, therefore, 2μ, where μ is the size of the truncation band of the TSDF (Fig. 5.7). To generate a realistic representation, the direction, in which the cylinder is tilted, is the same as the direction of the normal of the surfel (Fig. 5.7). Hence, the allocated voxels are aligned with the surfel's normal, representing the local surface normal of the scene.

A voxel, v, is updated if v lies within the cylinder of the surfel S_i. This means that the distance to the normal of the surfel has to be smaller than the radius of the cylinder and the distance to the plane, defined by the surfel is smaller than the truncation size.

$$\frac{(c_i - v_w) \times n_i}{\|n_i\|} \leq r \tag{5.21}$$

$$\frac{(c_i - v_w) \cdot n_i}{\|n_i\|} \leq \mu, \tag{5.22}$$

For each voxel, v, for which the conditions (5.21) and (5.22) hold, the corresponding distance to the surfel, $dist(v, S_i)$, can be calculated by projecting the centerpoint, c_i, of the voxel onto the normal, n_i, of the surfel. The result then has to be normalized in order to obtain the SDF-value, $F_{Surfel}(v)$.

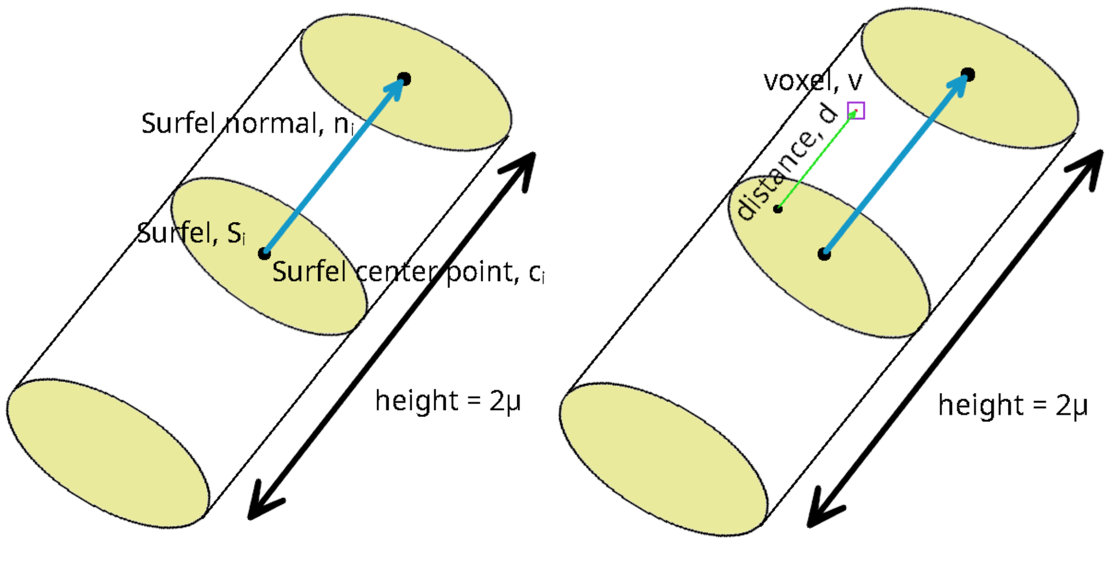


Figure 5.7: (a) Visualization of a generated cylinder around a surfel, (b) and the distance from a voxel to the surface of the surfel.

$$dist(v_w, S_i) = \frac{\langle c - v_w, n_i \rangle}{\|n_i\|} \tag{5.23}$$

$$F_{Surfel}(v) = \frac{dist(v_w, S_i)}{\mu} \tag{5.24}$$

Here, v_w again denotes the voxel center in world coordinates and n_i denotes the normal of surfel S_i in the MRSMap.

In order to obtain a suitable weight for each allocated voxel, one has to find a measure of how certain one is that a voxel is that the SDF-value of a voxel is correct. The error of the SDF-value is probably smallest close to the surfece of the surfel and gets larger if the voxel is farther away from the surfel. Therefore, the weight can be calculated by computing the inverse Euclidean distance from v_w to the surfel center, c_i.

$$w_\mu(v) := \frac{1}{1 + \frac{\|v_w - c_i\|}{\mu}}, \tag{5.25}$$

so that $w_\mu(v) \in (\frac{1}{2}, 1]$.

Furthermore, if surfels are generated by points that are farther away from the sensor, it is more likely that the surfel will be less accurate, as the points used to generate the surfel are less accurate. Analog to (5.18) we define a weight component based on the distance from the surfel center to the sensor origin, o.

$$w_{o_{Surfel}}(v) := 1 - \frac{\|o - c_i\|}{\chi} \tag{5.26}$$

Here, χ denotes the maximum range of the sensor and c_i denotes the center of the surfel. The weight components combined result in the overall weighting for a voxel, v.

$$W_{Surfel}(v) := w_{o_{Surfel}}(v) w_{\mu_{Surfel}}(v) \tag{5.27}$$

This approach does not give promising results, as there are many limitations when using surfels as intermediate step when fusing LiDAR data into the TSDF. The limitations and drawbacks of the use of surfels will be discussed in section 7.5 in more detail. As can be seen in Fig. 5.8, the scene is not reconstructed well by this approach. Only the rough structure of the scene is reconstructed. The detail in the reconstructed scene, however, is completely lost.

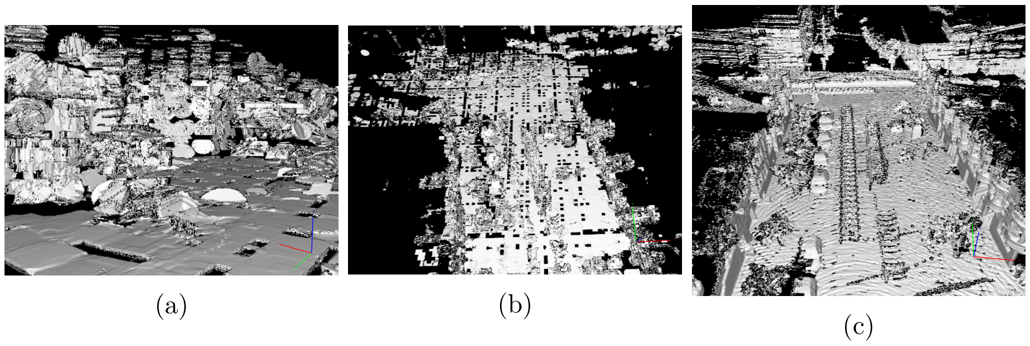


Figure 5.8: Visualization of the scene reconstruction using the surfel approach of (a) a small part of the scene and (b) the entire scene and (c) a comparison when using the Raycasting approach.

5.6 Cylindrical Projection

As mentioned in section 4.2, Kühner et al. [18] propose an approach, that projects the LiDAR data onto a cylindrical depth image. However, their software is not provided. Therefore I will reimplement this approach into the software provided by InfiniTAM [16] for evaluation reasons only.

5.6.1 Generating a Depth Image from a Point Cloud

As this approach requires the use of depth images, a given point cloud, $\mathcal{P}$, in sensor coordinates, is first transferred into a depth image, of size (vr, hr) in a preprocessing step. hr is the width and vr is the height of the image. The with of the depth image will be chosen to match the horizontal resolution of the LiDAR sensor. This is done so that the projection of the points onto the image falls onto the horizontal center of each pixel. The height of the image is chosen, so that the pixels are approximately square [18].

The pixel coordinates, $[u, v]$, of a point $p = [x, y, z]^T \in \mathcal{P}$, can be computed using the maximum possible elevation angle of the sensor, θ_1, and the minimum possible elevation angle of the sensor, θ_2, as follows

$$u = \frac{\varphi}{360} * w \tag{5.28}$$

$$v = \left\{ \begin{array}{cc} \delta_1(1 - \frac{z}{\rho * tan(\theta_1)}), & z \geq 0 \\ \delta_1 + \frac{\delta_2 z}{\rho * tan(\theta_2)}, & z < 0 \end{array} \right. , \tag{5.29}$$

where $\delta_1 = \left[\frac{\tan\theta_1 - \tan\theta_2}{\tan\theta_1} * vr\right]$ and $\delta_2 = \left[\frac{\tan\theta_1 - \tan\theta_2}{\tan\theta_2} * vr\right]$ is used to map the the points, $p_i \in \mathcal{P}$, so that the points with maximal angle are close to the top of the depth image and the points with minimal angle are close to the bottom of the depth image. The depth of each pixel is computed by computing the distance of the point, that is projected on the xy-plane, $\rho = \sqrt{x^2 + y^2}$. φ denotes the azimuth with range $(0, 2\pi)$:

$$\varphi = \left\{ \begin{array}{cc} atan2(x, y), & atan2(x, y) \geq 0 \\ atan2(x, y), +2\pi & atan2(x, y) \leq 0 \end{array} \right. \tag{5.30}$$

5.6.2 Fusing a Depth Image into the TSDF

Having obtained the depth images, they can be integrated into the TSDF. In each fusion step the (x, y, z)-coordinate of each depth value is computed in world coordinates.

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} \cos\left(\frac{u2\pi}{hr}\right)d \\ \sin\left(\frac{u2\pi}{hr}\right)d \\ \tan\left(\frac{v2\pi}{vr}\right)d \end{pmatrix}, \tag{5.31}$$

Here, u is the horizontal index of the depth image, v is the vertical index of the depth image, and d is the depth value. The computed points $x' = [x', y', z']^T \in \mathbb{R}^3$ are, however, only the points in sensor coordinates and have to be transformed into world coordinates. This is done by applying

$$x =^{World} M_{Sensor} x', \tag{5.32}$$

where ${}^{World}M_{Sensor}$ is the affine transformation from the sensor coordinate system to the world coordinate system.

Similar to the Single Ray approach, voxels in the TSDF representation, that lie on the ray from the sensor to the detected point and are within the truncation band, are regarded for allocation. Therefore, given a point $p_i = [x, y, z]^T$, all voxels, v, with

$$\|v_w - p_i\| \leq \mu \tag{5.33}$$

$$\frac{|(o - v_w) \times (p_i - o)|}{|p_i - o|} < \sqrt{3\left(\frac{\kappa}{2}\right)^2}, \tag{5.34}$$

where o is the sensor origin and v_w is the voxel center in world coordinates, are considered for fusion. The SDF-value and weight of each of these voxels can be computed analog to (5.7) and (5.8).

6 Implementation details

The implementation of all four approaches is based on the software provided by InfiniTAM [16], using voxel hashing for efficient memory consumption, as described in section 5.1.1. Both the allocation and fusion of new data entities is done by traversing a cube of voxels for each element in the data entity. The cube size depends on the method used and the size of the element in allocation and fusion. In order to obtain an efficient time, parallel implementation is employed, by executing 256 CUDA kernels in parallel.

6.1 Raycasting

As the approach employing Raycasting computes new values for voxels that lie on the ray between the sensor origin and each detected point in the point cloud, a cube of voxels of size

$$\left\lceil 2\sqrt{\frac{3}{4}\kappa^2+\mu^2} \right\rceil \times \left\lceil 2\sqrt{\frac{3}{4}\kappa^2+\mu^2} \right\rceil \times 2\left\lceil \sqrt{\frac{3}{4}\kappa^2+\mu^2} \right\rceil$$

around the detected point, is traversed. The size of the cube of voxels is chosen, as it is just large enough to guarantee that all needed voxels lie within the block, regardless of the direction of the ray to the sensor. For each voxel within the cube of voxels, for which the condition (5.6) holds, memory is allocated and the respective signed distance value and weight is computed with (5.7) and (5.8) respectively.

6.2 Approximated Local Normals

Similar to the Raycasting approach a cube of voxels of size

$$\left\lceil 2\sqrt{\frac{3}{4}\kappa^2+\mu^2} \right\rceil \times \left\lceil 2\sqrt{\frac{3}{4}\kappa^2+\mu^2} \right\rceil \times 2\left\lceil \sqrt{\frac{3}{4}\kappa^2+\mu^2} \right\rceil$$

around the detected point is traversed and each voxel in the block is updated, if the conditions (5.14) and (5.15) hold. The size of the cube of voxels ensures that all needed voxels, regardless of the direction of the normal, lie within it. The SDF-value and the weight of each voxel is updated by (5.16) and (5.20) respectively.

6.3 Surfel Approach

The surfel approach makes use of Multi-Resolution surfel Maps (MRSMaps), as proposed in [8]. In each iteration of the fusion, all surfels of the MRSMap are used to fuse the data into the TSDF. For each surfel a cube of voxels of size $2\lambda_1 \times 2\lambda_1 \times 2\lambda_1$ around the center of the surfel is traversed, where λ_1 denotes the length of the first principal component. For each voxel in the cube of voxels for which the conditions (5.21) and (5.22) hold, memory will be allocated and a corresponding SDF-value and weight is calculated using (5.24) and (5.27) respectively.

6.4 Cylindrical Projection

Having computed the points in world coordinates, the necessary space has to be allocated and SDF-values have to be computed for all voxels, for which 5.33 and 5.34 hold. Similar to the single ray approach for each detected point $x \in \mathbb{R}^3$ a cube of voxels with size

$$\left\lceil 2\sqrt{\frac{3}{4}\kappa^2 + \mu^2} \right\rceil \times \left\lceil 2\sqrt{\frac{3}{4}\kappa^2 + \mu^2} \right\rceil \times 2\left\lceil \sqrt{\frac{3}{4}\kappa^2 + \mu^2} \right\rceil$$

and center x is traversed and voxels are allocated if the conditions hold. The SDF-value and weight of each allocated voxel are computed in the same way as is done in the Raycasting approach with (5.7) and (5.8) respectively.

7 Evaluation

7.1 Datasets

The evaluation of the proposed methods will be done on the 32nd sequence of the PandaSet [1] and on the first 600 scans of the DRZ Living Lab data [10]. The used voxel size is 5cm and the size of the truncation band is 20cm. Testing showed that these parameters are a good tradeoff between memory consumption and accuracy of a reconstructed scene of this size.

7.1.1 PandaSet

The PandaSet uses Hensai's LiDAR sensors and Scale AI's data annotation. The scenes used in the Pandaset are generated from two routes in Silicon Valley using a car: San Francisco and El Camino Real from Palo Alto to San Mateo [1]. Regarding the LiDAR setup, the Pandaset incorporates a forward-facing image-like LiDAR sensor (PandarGT) and a mechanical spinning LiDAR (Pandar64) [1].

In this thesis, I will only be making use of the data obtained by the mechanical spinning LiDAR sensor. This sensor has 64 channels, a range of 200m with 10% reflectivity and a 360° horizontal Field Of View (FOV) and a 40° vertical FOV, with range $-25°$ to 15° [1]. Its horizontal angular resolution is 0.2°, with 10 Hz capture frequency and a vertical angular resolution of 0.167° [1].

The LiDAR-data in the original dataset is given in world coordinates. However, it does not incorporate data on the used laser-Id, the elevation angle, or the angle on the xy-plane, which is needed for the approach that approximates local surface normals.

Therefore, I will be using the raw data provided by [33] for the approach using the local approximation of the surface normals. Moreover, the PandaSet [1] only provides the vehicle pose, ${}^{World}M_{Car}$ in world coordinates for the LiDAR data. In order to be able to use the raw data, the pose of the sensor is needed. The offset between the sensor pose and the vehicle pose is obtained by applying the ICP-algorithm on the point cloud in vehicle coordinates and the point cloud, generated by the raw data, resulting in a transformation, ${}^{Car}M_{Sensor}$. To obtain the world coordinates from the raw data, one can apply the following transformations to each point, $p_{raw} = [x, y, z]^T$ in the raw point cloud:

$$p_{world} =^{World} M_{Car} \cdot^{Car} M_{Sensor} \cdot p_{raw} \tag{7.1}$$

7.1.2 DRZ Living Lab

This dataset includes the first 600 LiDAR scans of the DRZ (Deutschen Rettungsrobotik Zentrum) Living Lab [10]. The LiDAR scans have been collected using an Ouster OS-0 with 128 channels at 10 Hz attached to a DJI M210 v2. A Motion Capture (MoCAp) system is employed to generate ground truth poses [23]. The Ouster OS-0 has a range of 50m, a horizontal resolution of 1024 points per scanline, and a vertical resolution of 0.7°. The 128 channels are uniformly distributed so that we obtain a vertical Field Of View of -45° to +45° [22]. The poses of each scan are obtained by the method proposed in [23] and are used as given poses for the mapping into the TSDF.

7.1.3 Generating a Ground Truth

The given data of the PandaSet is composed of point clouds, $\mathcal{P}_i$ that are aligned in world coordinates. These can be combined to obtain a point cloud, that represents the entire scene, $\mathcal{P}_{all}$. This point cloud will be used as ground truth. Having fused the data of one sequence of the PandaSet into the TSDF representation, a mesh, M_{TSDF} can be generated using the Marching Cubes Algorithm [19], which will be used to determine the error of each proposed method. The mesh is converted into a point cloud $\mathcal{P}_{M_{TSDF}}$, in order to compute the Hausdorff Distance and the Average Distance (section 7.2) between $\mathcal{P}_{M_{TSDF}}$ and $\mathcal{P}_{all}$. This is done using open3d's sample points Poisson disk method, which is based on [34].

Similarly, the point clouds of the DRZ dataset will be accumulated to form a combined point cloud $\mathcal{P}_{all}$. This is done by transforming each point cloud with respect to the given pose. We will only consider every tenth scan, as $\mathcal{P}_{all}$ would otherwise be too large to work with. $\mathcal{P}_{M_{TSDF}}$ will be obtained from $\mathcal{P}_{all}$ in the same way as for the PandaSet.

7.2 Metrics for Evaluation

7.2.1 Hausdorff Distance

The Hausdorff Distance computes the maximal distance between two nearest neighbors of two point clouds and is defined by

$$d_H(X,Y) := \max\{\sup_{x\in X}\inf_{y\in Y} d(x,y), \sup_{y\in Y}\inf_{y\in X} d(x,y)\}, \tag{7.2}$$

where X and Y are point clouds. In the scope of the evaluation, only the one-sided Hausdorff distance, with the ground truth as reference is used. This is because using the point cloud, generated by the mesh, as reference will cause misleading results, as the $\mathcal{P}_{M_{TSDF}}$ is much smaller than $\mathcal{P}_{all}$. We therefore only compute

$$d_H(X,GT) := \sup_{x\in X}\inf_{y\in GT} d(x,y), \tag{7.3}$$

where X denotes the point cloud generated by the mesh and GT denotes the ground truth point cloud. The Hausdorff Distance is good for determining outliers of the error.

7.2.2 Average Distance

Furthermore, the Average Distance will be employed, thus including all points into the error. The Average Distance for a point cloud, $\mathcal{P}$, is defined by

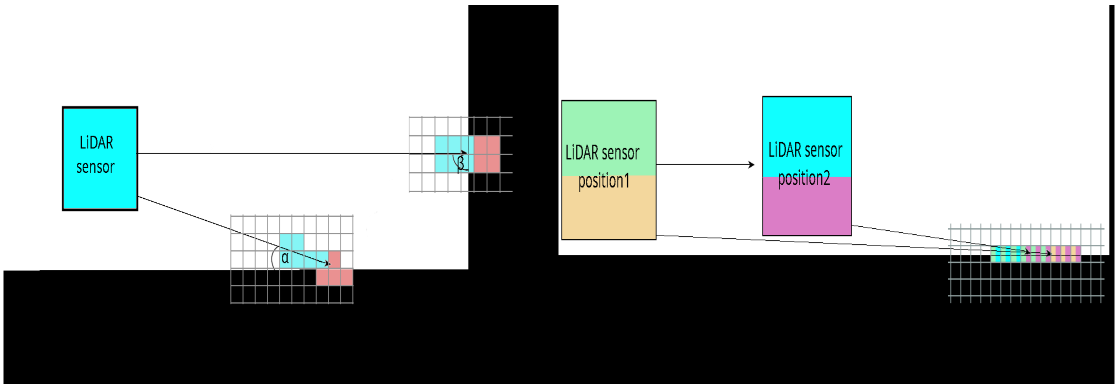
$$\frac{1}{|\mathcal{P}|}\sum_{a\in\mathcal{P}}\min_{b\in GT}|a-b|, \tag{7.4}$$

where GT is the ground truth.

7.3 Raycasting

Casting a ray from the sensor origin onto a registered point is a simple approach that shows promising results. However, only the parts of the TSDF, where the angle between the tangent of the local surface and the ray is large enough are represented realistically. In this case, the allocated voxels in the TSDF are approximately aligned to the direction of the local surface normal. When the angle between the observed surface and the ray from the sensor is small, however, the allocated voxels in the TSDF do not point towards the local normal of the surface.

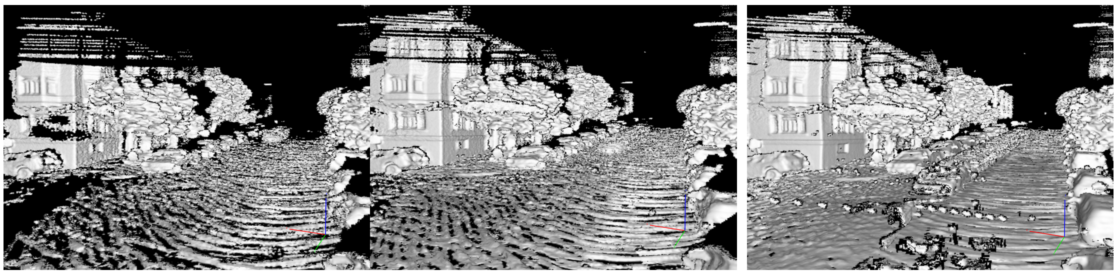
Fig 7.1(a) demonstrates the detection of two points, one of which has a small angle between the surface and the ray (with angle α), and the other with a large angle between the surface and the ray (with angle β). In Fig 7.1(a) only the ray with angle β allocates voxels that are approximately aligned to the local surface normal, thus generating a realistic TSDF. Fig. 7.1(b) shows the fusion of two consecutive scans. Points, that are detected by the LiDAR sensor in different scans and are close to each other may delete the resulting TSDF of one another if the angle between ray and surface is small, hence resulting in a TSDF with holes.



(a) (b)

Figure 7.1: (a) Blue denotes the positive parts of the TSDF, red the negative parts, black the occupied space, and white the empty space. (b) Two consecutive LiDAR scans, with the resulting fusion of the to the TSDF. Green denotes the voxels with positive SDF-values in the first LiDAR scan, orange the voxels with negative SDF-values, blue and purple for the second LiDAR scan respectively.

This sort of error mostly occurs when the LiDAR sensor is far away from the detected points, for instance when scanning the distant street using a sensor on a car. Here the angle between the street and the ray is very small. However, using a weighting that gives smaller weights to points with a larger distance to the sensor, partially resolves this problem. When approaching such areas with the sensor, the TSDF is overridden by the detected points there, as their weight is larger. Fig 7.2 illustrates the TSDF in the PandaSet [1] when the sensor is far away from the detected points on the left and when the detected points are close to the sensor on the right. As can be seen in Fig. 7.2(c) the areas in which the TSDF had holes in Fig. 7.2(a) and Fig. 7.2(b) are mostly smooth surfaces now.



(a) (b) (c)

Figure 7.2: (a) Detected points are approximately 100m away from the sensor, (b) detected points are approximately 50m away from the sensor, and (c) detected points are approximately 25m away from the sensor.

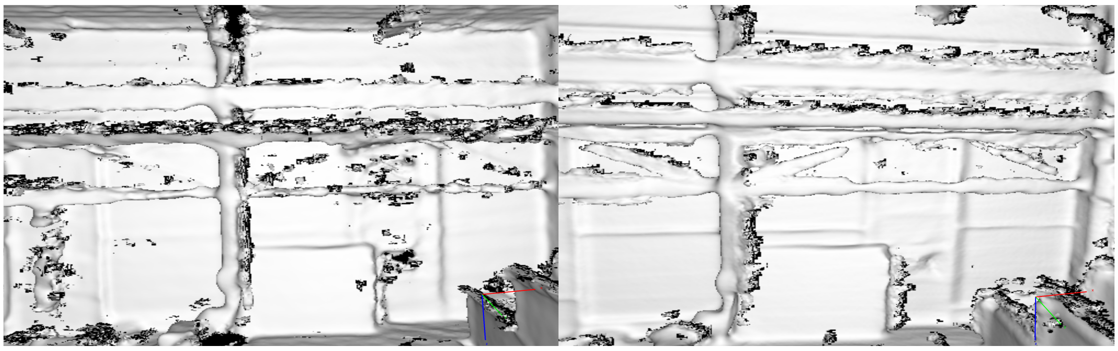
7.4 Approximation of Local Surface Normal

The approach using approximated local surface normals, which are computed by the local neighborhood of each point in the organized point cloud shows promising results. However, there are two main drawbacks when employing this approach. The first one being, that a large amount of data will be omitted, as one only considers the points, where the point itself and all of its four neighbors have been detected.

Furthermore, points, in which the normals of the adjacent triangles vary too much will be omitted as well. This especially happens at detected edges or at boundaries between objects, thus resulting in an incomplete or noisy TSDF in these regions. Fig. 7.3(a) illustrates the resulting incomplete edges in the TSDF when using the approach employing the approximated local surface normals. In contrast, Fig. 7.3 (b) visualizes the same part of the scene when using the Raycasting approach. It can clearly be seen that the edges in Fig. 7.3(a) are less complete and more noisy than in Fig. 7.3(b).

The second drawback of the approach using approximated local surface normals is the strong dependence on the correct sensor pose. In extreme cases, the computed normal rather reflects the negative surface normal, than the desired approximation of the surface normal. This for instance is observed when the sensor is positioned directly on the surface.

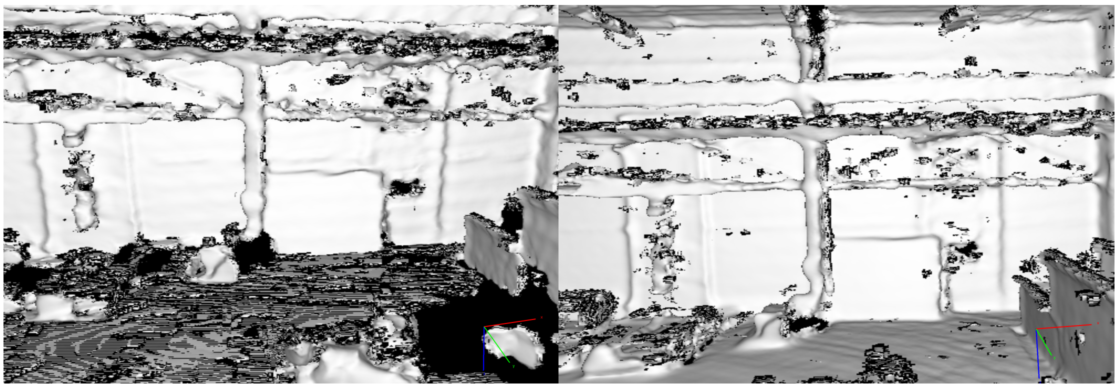
In the DRZ Living Lab data [10] this occurs in the first few LiDAR scans. Here the drone and the sensor are positioned on the floor of the lab before takeoff. Fig.7.4(a) illustrates how the TSDF after only a few amount of fusion steps, where the LiDAR sensor is very close to the floor. It can clearly be seen that the approximated normal is pointing in the opposite direction as one would want it



(a) (b)

Figure 7.3: Visualization of the resulting TSDF on part of the DRZ Living Lab data [10] when (a) using the approach that approximates the local surface normals and (b) using the Raycasting approach.

to. When more scans are fused into the TSDF this problem, however, is resolved, as more correct data is added to these areas.



(a) (b)

Figure 7.4: Visualization of the TSDF on a part of the DRZ data [10] after (a) 40 fusion steps (b) 100 fusion steps.

Another drawback when using this approach is the fact that one depends not only on the sensor pose and the positions of the points in the point cloud. Information on what laser-Id is used to detect each point is also required to form an organized point cloud. Even though this information is usually provided by the sensor, some datasets lack this information, for instance, the PandaSet [1].

Overall this approach still reconstructs the scene well, as can be seen in Fig.7.4(b).

7.5 Surfel approach

Having tested the surfel approach on the PandaSet [1] and the DRZ Living Lab [10] some limitations of the surfel approach using a voxel-based surfel generation such as MRSMaps has become clear, especially when approximating the normal of a surfel.

When computing a normal, n, in 3D using the plane of the surfel some ambiguity arises, as $-n$ is a valid normal for this plane as well. In order to choose the correct normal, one checks whether the angle between the normal and the ray from the surfel's center point to the sensor origin is larger than $90°$. If this is the case then the normal is flipped ($n = -n$).

However, in some cases, this does not make any sense. Considering a downward slope of a street for example this computation results in the wrong choice of normal, as illustrated in Fig 7.5. The yellow arrow denotes the correct normal and the red arrow denotes the computed normal. If in two different surfel maps, surfels with normals pointing in opposite directions and in similar positions can be found, they may cancel each other out. In practice this occurs very often, resulting in a TSDF with holes in problematic areas, as shown in Fig. 7.6(a).

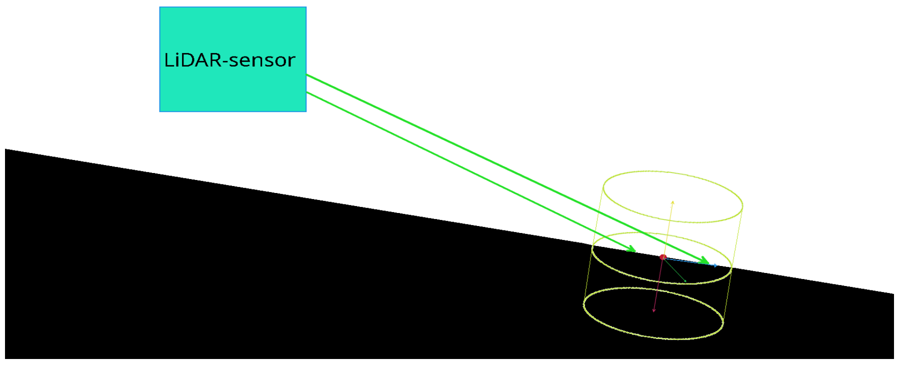


Figure 7.5: Visualization of the drawback of computing the normal using surfels.

A workaround to this specific problem is flip the normals of the surfels that are incorrect: First we rotate the surfels into the sensor coordinates. If the z-coordinate of the surfel center is smaller than 0 and the z-value of the normal is negative, the normal, n is flipped ($n = -n$).

This works well on the PandaSet [1], however, this method only considers regions below the sensor and is very limited to datasets where the data is obtained by a sensor on a car. Fig. 7.6 illustrates the resulting TSDF on the PandaSet when the normals are not flipped (Fig. 7.6(a)) and when using the proposed workaround

(Fig. 7.6(b)).

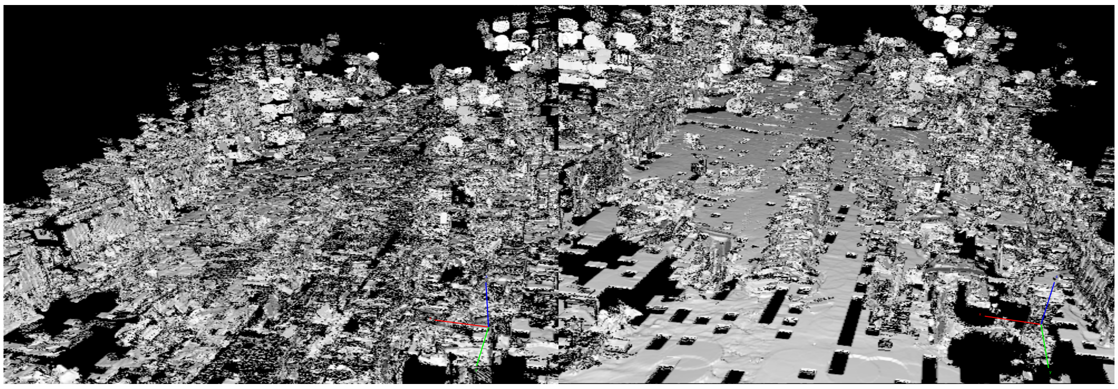


Figure 7.6: Rendering of the TSDF when (a) the incorrect normals of the surfels are not flipped and (b) the incorrect normals are flipped if needed.

Another drawback of the computation of the normals using a voxel-based surfel generation is that in many cases the normal of a surfel is computed incorrectly. As stated in section 3.5 the surfels in the MRSMap are generated by combining all detected points within a predefined voxel to a surfel using PCA. The incorrect normal approximation occurs when for instance a voxel includes detected points of only one scanline. Applying PCA to these points in order to generate a surfel often results in an incorrect approximation of the local surface normals, as shown in Fig. 7.7. Here, the distribution of points lying on a single scanline does not reflect on the distribution of the entirety of the points.

Having tested on the PandaSet, it has become clear, that this phenomenon often occurs in areas, where the scanlines are farther apart. Consequently, this leads to a TSDF, that does not represent the scene well, as shown in Fig. 7.7(c). The PandaSet incorporates larger angles for the first few and last few channels, thus resulting in scanlines that are far apart, as shown in Fig 7.7(a). This makes it likely that there are a lot of voxels that only include points of a single scanline.

Furthermore, the use of surfels limits the amount of detail that can be achieved by the scene reconstruction. Primarily this is due to the fact that all points within a voxel are combined to obtain a local surface. In addition, voxels that have too few points will not generate a surfel, thus resulting in holes in the TSDF. Fig. 7.8 compares the surfel approach on the left with the raycasting approach on the right on the DRZ Living Lab dataset. It can clearly be observed, that the surfel approach exhibits less detail.

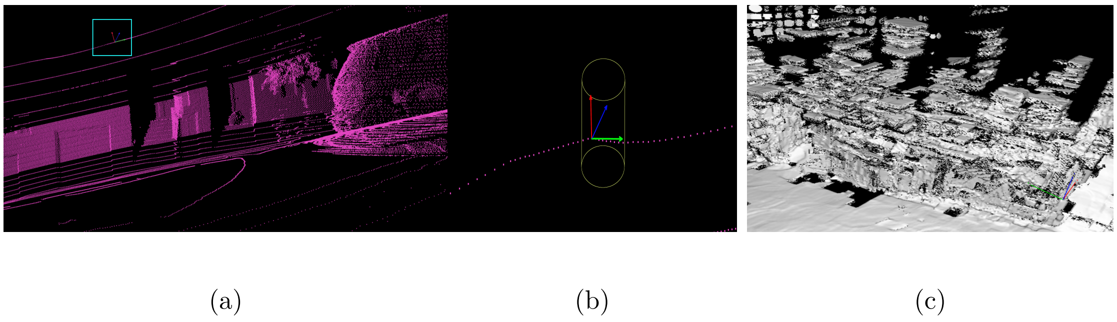


Figure 7.7: (a) Visualization of a point cloud and a voxel (cayan box) for surfel generation (b) cylinder around the generated surfel with normal (red arrow) (c) visualization of the resulting TSDF.

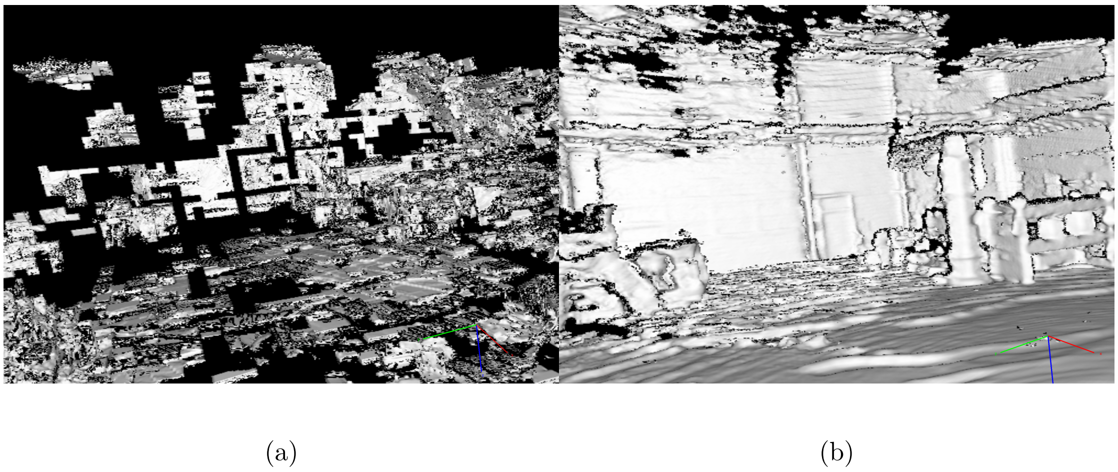


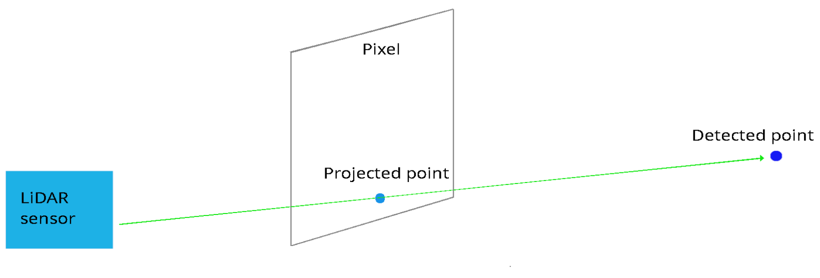
Figure 7.8: Rendering of the TSDF using the DRZ data when (a) using the surfel approach and (b) using the Raycasting approach.

7.6 Cylindrical Projection

The method proposed by Kühner et al. [18] works very well, however, there are some drawbacks to their method, as they discretize the given data when converting LiDAR scans into depth images.

In the preprocessing step, where the LiDAR data is converted into depth images, the pixel coordinates $[u, v]$ are computed according to section 5.6.1. In general, the horizontal value, u, of the computed depth image is aligned to the original LiDAR data, as the size of the depth image is chosen to reflect the number of points that are recorded in a scanline. Therefore, the ray that is cast through each pixel to the detected point will go through the center of the pixel in horizontal direction.

The ray, however, will not necessarily intersect the pixel center in vertical direction. Still, when fusing the depth image into the TSDF, it will be assumed, that the detected points are represented by the center of each pixel. This is especially problematic in areas where edges or boundaries between objects occur in the scene.



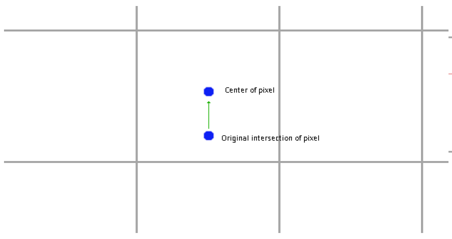
(a)

Figure 7.9: Projection of a LiDAR point onto a pixel of the depth image.

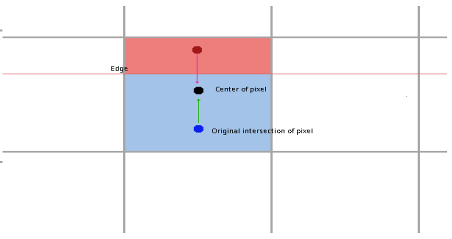
Fig. 7.9 demonstrates how a pixel of the depth image is assigned to a detected point, and how the position of the detected point varies through the rounding error, shown in Fig. 7.10(a). The points in the LiDAR data are therefore shifted slightly downward or upward in the each pixel of the depth image.

If in two different depth images two points in similar positions are detected and one is shifted downwards and the other is shifted upwards they may cause an overlap of different objects in the scene. Fig. 7.10(b) illustrates how two detected points may be shifted downwards (red point) and upwards (blue point), thus overlapping. The red line indicates an edge that separates one object (red) from the other (blue). The points may therefore have a different depth value, as they might belong to different objects.

When fusing both points consecutively into the TSDF the voxels with positive SDF-values, that corresponds to the rear point will be canceled out by the negative SDF-values of the front point.



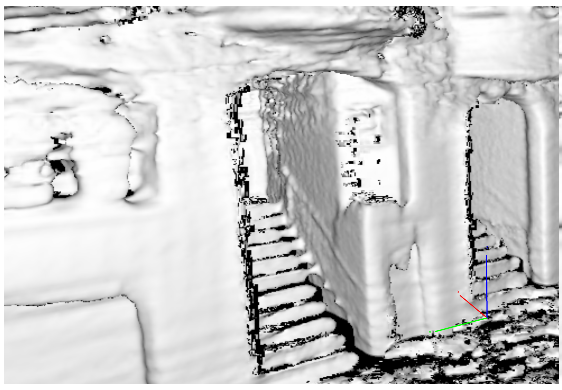
(a)



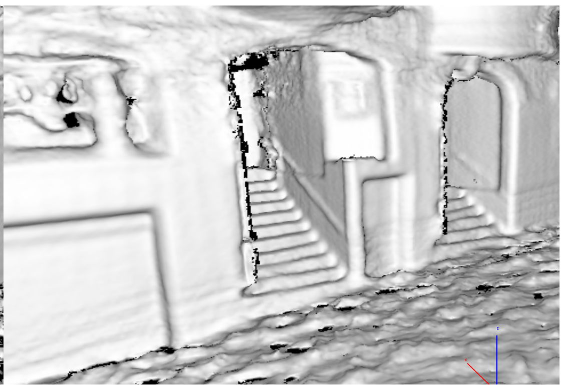
(b)

Figure 7.10: Rounding of the LiDAR data when obtaining the depth image in (a) a single data point and (b) two similar data points in presence of an edge.

These rounding errors result in holes in areas of the TSDF where edges are present. Fig. 7.11 shows how this affects the TSDF by comparing the Raycasting approach with the approach using cylindrical projection. On the stairs, in Fig. 7.11 the impact of the rounding error can be observed clearly.



(a)



(b)

Figure 7.11: Rendering of the TSDF on the PandaSet [1] (a) using the cylindrical projection approach (b) using the Raycasting approach.

7.7 Quantitative Evaluation

As stated in section 7.2 the Hausdorff Distance and the Average Distance will be used on the PandaSet [1] and the DRZ Living Lab data [10] to evaluate how well each method mapped the data into the TSDF. Furthermore, an analysis of the time taken for one fusion step will be evaluated.

7.7.1 Distance Metrics

Table 7.1 summarises the results of the Average Distance and corresponding standard deviation in meters. As expected, the approach using surfels has the largest Average Distance of all methods with approximately 20cm for the PandaSet and 28cm for the DRZ data. The other approaches have similar errors at around 11cm for the PandaSet and 15-16cm for the DRZ data.

	Method	Average Distance	Standard deviation
PandaSet	Raycasting	0.108	1.972
	Cylindrical Projection	0.108	1.205
	Approximated Normals	**0.106**	1.689
	Surfel pproach	0.198	**0.222**
DRZ	Raycasting	**0.147**	0.494
	Cylindrical Projection	0.161	0.607
	Approximated Normals	0.151	0.547
	Surfel Approach	0.280	**0.349**

Table 7.1: Overall Average Distance and standard deviation.

Table 7.2 shows the resulting Hausdorff Distances of the proposed methods.

	Method	Hausdorff Distance
PandaSet	Raycasting	4.761
	Cylindrical Projection	**4.478**
	Approximated Normals	5.329
	Surfel Approach	4.572
DRZ	Raycasting	5.06
	Cylindrical Projection	5.16
	Approximated Normals	**4.67**
	Surfel Approach	5.98

Table 7.2: Overall Hausdorff Distance.

These results, however, do not accurately reflect the effectiveness of the approaches, as large errors can occur when subsampling points from the generated mesh, $\mathcal{P}_{M_{TSDF}}$ in sparse regions. This can have a large impact on the used metrics. Fig. 7.12 illustrates how in regions where less data has been accumulated in the TSDF, the generated mesh is less accurate than in regions where the sensor has detected a variety of points.

The distance of each point to its nearest neighbor in ground truth, $\mathcal{P}_{all}$, is visualized on the PandaSet (Fig. 7.12(a)) and the DRZ data (Fig. 7.12(b)).

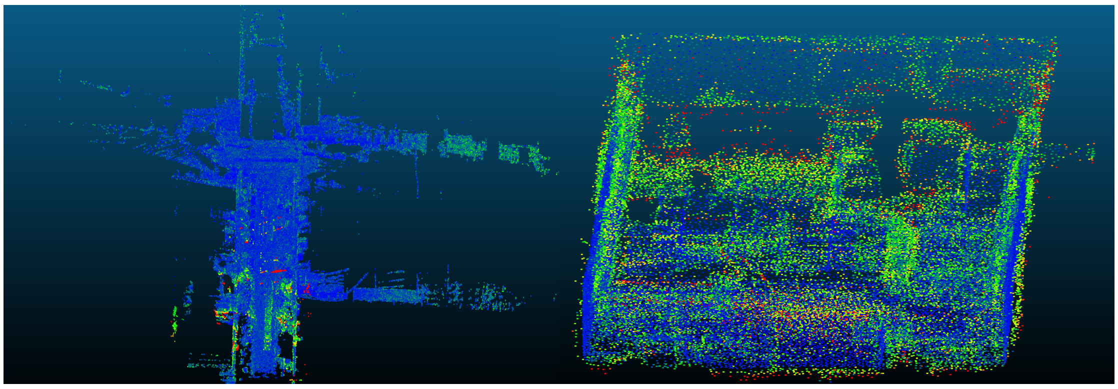


Figure 7.12: Visualization of the distance to the nearest neighbor of each point, subsampled from the mesh in (a) the PandaSet and (b) the DRZ data.

In order to compare the proposed methods more accurately, the Average Distance and the Hausdorff Distance will be computed on the TSDF in regions, that are more dense. Here, the generated mesh will represent the TSDF more accurately and therefore the subsampled points will reflect better on the resulting TSDF for each approach. Tables 7.3 and 7.4 summarise the average distance and the Hausdorff distance on regions that are densely mapped in the TSDF. It can clearly be seen that the values have improved. Especially, as pointed out by the standard deviation, there is not as much variation in the distance from one point to another.

	Method	Average Distance	Standard deviation
PandaSet	Raycasting	**0.0537**	**0.0791**
	Cylindrical Projection	0.0709	0.0965
	Approximated Normals	0.0655	0.0879
	Surfel Approach	0.135	0.180
DRZ	Raycasting	**0.108**	0.165
	Cylindrical Projection	0.120	0.172
	Approximated Normals	0.113	0.155
	Surfel Approach	0.236	**0.127**

Table 7.3: Average Distance and standard deviation of a dense region.

	Method	Hausdorff Distance
PandaSet	Raycasting	1.833
	Cylindrical Projection	1.689
	Approximated Normals	1.813
	Surfel Approach	**1.508**
DRZ	Raycasting	**1.221**
	Cylindrical Projection	1.589
	Approximated Normals	1.321
	Surfel Approach	1.537

Table 7.4: Hausdorff Distance of dense region.

As the Hausdorff Distance is strongly affected by outliers, the average distance, including its standard deviation provides a better insight into how well an approach has mapped the LiDAR data into the TSDF. Regarding the computations of tables 7.4 and 7.3 one can see that the Raycasting approach gives the best results. However, the approach using cylindrical projection and the approach using an approximation of the normals also work well. The approach employing surfels is as expected the least favorable approach.

7.7.2 Runtime

In order to evaluate the time taken to fuse a single LiDAR scan into the TSDF, the runtime analysis is divided into two parts: (i) the time taken to process a single LiDAR scan i.e. to allocate the necessary space in the hash table and to assign the SDF-value and weight to each voxel and (ii) the time taken to read the data into the CPU and transfer it to the GPU. All data on the time taken has been obtained by evaluating on an NVIDIA GeForce RTX 3090 and an Intel Core i7-8700K.

Fig. 7.13 shows the respective times taken to allocate space and compute the SDF-values and weights for the voxels (Fig. 7.13(a)) and the time taken to load the LiDAR data into the CPU (Fig. 7.13(b)). Here, CP denotes the method employing cylindrical projection, OP denotes the method using approximated local normals, RC denotes the Raycasting approach, and S denotes the surfel approach.

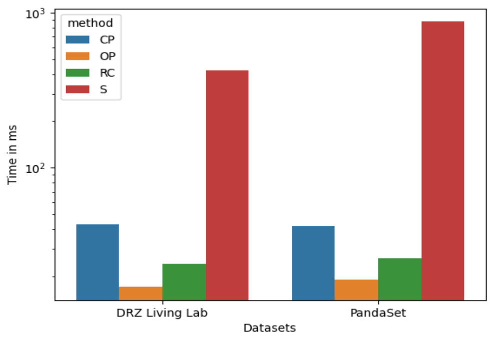
It can clearly be seen that the approach using surfels takes the longest time to both fuse the data to the TSDF on the GPU and to load the data to the CPU. This is due to the fact that the surfel approach applies PCA in a preprocessing step when loading the data into the CPU. Furthermore, in the allocation and computation step, a lot more voxels have to be traversed for a surfel than for a single point, as stated in section 6.3. Therefore the surfel approach can not be considered as a real-time application, taking up to a total of 1.9 seconds for a single fusion step.

The other approaches however can be considered to run in real-time with a possible rate of 18-35 fusion steps per second.

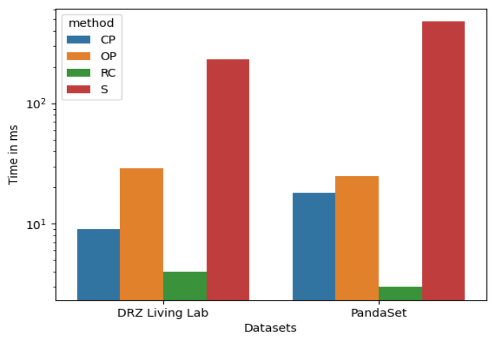
The approach using cylindrical projection, however, is a little slower than the other approaches with approximately 18-20 fusion steps per second, as often more pixels are present in the depth image than points in the LiDAR point cloud.

The approach using the approximated local normals also requires more time to load the data into the CPU, as all four neighbors of each point have to be considered as well. With an approximate fusion rate of 20-25 fusion steps per second, it still performs well.

The Raycasting approach with a fusion rate of 27-35 fusion steps per second performs best in the overall runtime.



(a)



(b)

Figure 7.13: Visualization of the average time taken to (a) allocate space and compute SDF-values and weights for all needed voxels and (b) to load the LiDAR data into the CPU for a single LiDAR scan on the DRZ Living Lab data [10] and the PandaSet [1].

Conclusion

In this work I have presented three novel approaches to fuse LiDAR data into the TSDF for large-scale outdoor and indoor 3D scene reconstruction: (i) a Raycasting approach, (ii) an approach using approximated local surface normals and (iii) an approach using surfels that are generated in a preprocessing step.

The approaches (i) and (ii) fulfill the requirements of real-time dense 3D scene reconstruction in terms of accuracy and time taken for each fusion step. Unlike the approach proposed by Caminal et al. [3] all approaches work well on data that is obtained by obtaining LiDAR data from a rotating 360-degree-LiDAR sensor.

An advantage over the approach proposed by Kühner et al. [18] is that in approaches (i) and (ii) there is no need for converting the original LiDAR data into a depth image in a preprocessing step. This makes them easier to implement, while at the same time fusing data accurately into the TSDF. Furthermore, the runtime of both the Raycasting approach and the approach using approximated local normals is better suited for real-time applications than the approach proposed by Kühner et al. [18].

Approach (iii), however, is not suited for real-time scene reconstruction as it neither runs in real-time nor does it reconstruct the scene accurately.

In conclusion, both the Raycasting approach and the approach using approximated local normals provide a slight improvement to the state-of-the-art approach ([18]) when fusing LiDAR data into the TSDF in real-time. Still, the Raycasting approach should be the preferred option, as it incorporates all data, is easiest to implement, only relies on the sensor pose and the recorded LiDAR data, and has the fewest drawbacks when enough data of the scene is given.

In future work, the presented mapping approaches can be combined with tracking, in order to build a Simultaneous Localization and Mapping (SLAM) system. Here, it may be interesting to see how the TSDF ICP tracking performs on the LiDAR data, as it performs well on depth images from RGB-D cameras.

List of Figures

List of Tables

Bibliography

[1] Scale AI. *PandaSet*. 2021. URL: `https://scale.com/open-datasets/pandaset` (visited on 05/30/2021).

[2] Paul J. Besl. “Active, optical range imaging sensors.” In: *Machine vision and applications* (1988).

[3] I. Caminal, J. Casas, and S. Royo. “Slam-based 3d outdoor reconstructions from lidar data.” In: *International conference on 3d immersion*. IEEE. Brussels, Belgium: IEEE, 2018.

[4] J. R. Casas, P. Salembier, and L. Torres. “Morphological interpolation for texture coding.” In: *International conference on image processing*. Vol. 1. 1995, pp. 526–529.

[5] Brian Curless and Marc Levoy. “A volumetric method for building complex models from range images.” In: *23rd annual conference on computer graphics and interactive techniques*. SIGGRAPH ’96. New York, NY, USA: Association for Computing Machinery, 1996, pp. 303–312.

[6] Brian Curless and Marc Levoy. “A volumetric method for building complex models from range images.” In: *SIGGRAPH*. Ed. by John Fujii. ACM, 1996, pp. 303–312.

[7] Dirk Holz. “Efficient 3d segmentation, registration and mapping for mobile robots.” PhD thesis. Rheinische Friedrich-Wilhelms-Universität Bonn, 2017.

[8] David Droeschel, Jörg Stückler, and Sven Behnke. “Local multi-resolution representation for 6d motion estimation and mapping with a continuously rotating 3d laser scanner.” In: *IEEE international conference on robotics and automation (ICRA)*. 2014, pp. 5221–5226.

[9] Ivan Dryanovski, Matthew Klingensmith, S. Srinivasa, and J. Xiao. “Large-scale, real-time 3d scene reconstruction on a mobile device.” In: *Autonomous robots* 41 (2017), pp. 1423–1445.

[10] DRZ. *Deutsches Robotik Zentrum*. 2021. URL: `https://rettungsrobotik.de/living-lab/` (visited on 07/10/2021).

[11] Karl Pearson F.R.S. “LIII. on lines and planes of closest fit to systems of points in space.” In: *The london, edinburgh, and dublin philosophical magazine and journal of science* 2.11 (1901), pp. 559–572.

[12] Simon Fuhrmann and Michael Goesele. “Floating scale surface reconstruction.” In: *ACM transactions on graphics* 33 (2014), pp. 1–11.

[13] Andreas Geiger, Philip Lenz, and Raquel Urtasun. “Are we ready for autonomous driving? the kitti vision benchmark suite.” In: *2012 IEEE conference on computer vision and pattern recognition*. 2012, pp. 3354–3361.

[14] Adrian Hilton, A. Stoddart, John Illingworth, and Terry Windeatt. “Reliable surface reconstruction from multiple range images.” In: vol. 1064. 2006, pp. 117–126.

[15] R. A. Jarvis. “A perspective on range finding techniques for computer vision.” In: *IEEE transactions on pattern analysis and machine intelligence* 2 (1983), pp. 122–139.

[16] O. Kahler, V. A. Prisacariu, C. Y. Ren, X. Sun, P. H. S Torr, and D. W. Murray. “Very High Frame Rate Volumetric Integration of Depth Images on Mobile Device.” In: *IEEE transactions on visualization and computer graphics* 22.11 (2015).

[17] Maik Keller, Damien Lefloch, Martin Lambers, Shahram Izadi, Tim Weyrich, and Andreas Kolb. “Real-time 3d reconstruction in dynamic scenes using point-based fusion.” In: *2013 international conference on 3d vision - 3dv 2013*. 2013, pp. 1–8.

[18] T. Kühner and J. Kümmerle. “Large-scale volumetric scene reconstruction using lidar.” In: *2020 IEEE international conference on robotics and automation (ICRA)*. 2020, pp. 6261–6267.

[19] William Lorensen and Harvey Cline. “Marching cubes: a high resolution 3d surface construction algorithm.” In: 21 (1987), pp. 163–.

[20] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohi, J. Shotton, S. Hodges, and A. Fitzgibbon. “Kinectfusion: real-time dense surface mapping and tracking.” In: *2011 10th IEEE international symposium on mixed and augmented reality*. 2011, pp. 127–136.

[21] Matthias Nießner, Michael Zollhöfer, Shahram Izadi, and Marc Stamminger. “Real-time 3d reconstruction at scale using voxel hashing.” In: *ACM transactions on graphics (tog)* 32 (2013).

[22] Ouster. *Ouster*. 2021. URL: `https://ouster.com/products/os0-lidar-sensor//` (visited on 07/10/2021).

[23] Jan Quenzel and Sven Behnke. “Real-time multi-adaptive-resolution-surfel 6d lidar odometryusing continuous-time trajectory optimization.” In: (2021).

[24] L. Roldão, R. de Charette, and A. Verroust-Blondet. “3d surface reconstruction from voxel-based lidar data.” In: *IEEE intelligent transportation systems conference (ITSC)*. 2019, pp. 2681–2686.

[25] Henry Roth. “Moving volume kinectfusion.” In: *Bmvc 2012 - british machine vision conference 2012* (2012).

[26] Thomas Schöps, Torsten Sattler, and Marc Pollefeys. “Surfelmeshing: online surfel-based mesh reconstruction.” In: *Computing research repository* (2018).

[27] *Towards data science.* `https://towardsdatascience.com/light-detection-and-ranging-lidar-8f22971eeaea`. visited 2021-07-05.

[28] A. Watt and M. Watt. “Advanced animation and rendering techniques - theory and practice.” In: 1992.

[29] Mark Wheeler, Yoichi Sato, and Katsushi Ikeuchi. “Consensus surfaces for modeling 3d objects from multiple range images.” In: 1998, pp. 917–924.

[30] Thomas Whelan, Hordur Johannsson, Michael Kaess, John Leonard, and John Mcdonald. “Robust tracking for real-time dense rgb-d mapping with kintinuous.” In: (2012), p. 10.

[31] Thomas Whelan, Michael Kaess, Maurice Fallon, Hordur Johannsson, John Leonard, and John Mcdonald. “Kintinuous: spatially extended kinectfusion.” In: (2012).

[32] *Wikipedia.* `https://en.wikipedia.org/wiki/Time-of-flight_camera`. visited 2021-07-05.

[33] Hensai Technologies xpchuan-95. *PandaSet Raw Data.* 2020. URL: `https://drive.google.com/drive/folders/1ou_HqaQq0rGR0UrGZkFYWrE3mCJDosX8` (visited on 07/08/2020).

[34] Cem Yuksel. “Sample elimination for generating poisson disk sample sets.” In: *Computer graphics forum (eurographics 2015)* 34.2 (2015), pp. 25–32.