Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Magnus Stray Schmidt

Underwater Incremental Surface Reconstruction from Dynamic 3D Point Clouds

Master's thesis in Cybernetics and Robotics
Supervisor: Annette Stahl
Co-supervisor: Mauhing Yip and Rudolf Mester
June 2022

NTNU
Norwegian University of
Science and Technology

Magnus Stray Schmidt

Underwater Incremental Surface Reconstruction from Dynamic 3D Point Clouds

Master's thesis in Cybernetics and Robotics
Supervisor: Annette Stahl
Co-supervisor: Mauhing Yip and Rudolf Mester
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

Department of Engineering
Cybernetics

TTK4900

Engineering Cybernetics
Master's Thesis

Underwater Incremental Surface Reconstruction from Dynamic 3D Point Clouds

Magnus Stray Schmidt

Supervisor:
Annette Stahl, Department of Engineering Cybernetics
Co-supervisors:
Mauhing Yip, Department of Engineering Cybernetics
Rudolf Mester, Department of Computer Science

6th June 2022
Trondheim, Norway

Preface

This report constitutes the Master's Thesis of the Master's Degree Program in Cybernetics and Robotics at the Norwegian University of Science and Technology, and was supported by the Autonomous Robots for Ocean Sustainability (AROS) project[1]. The AROS project is funded by the Research Council of Norway and aims to advance the technology in marine robotics. The final goal is to achieve full underwater autonomy in order to perform demanding subsea operations in a safer, greener, and more cost-efficient manner. This entails the need for autonomy in underwater sensing, situational awareness, motion planning, and energy consumption. Realizing this goal will improve our ability to explore and sustainably develop the oceans. This report is a part of work package two of project AROS: *Next-best-view and 3D reconstruction for AIAUVs*.

Throughout the project, I have received helpful guidance from my supervisor, Annette Stahl, as well as my co-supervisors, Rudolf Mester and Mauhing Yip. I am deeply grateful for all their valuable input.

[1] https://prosjektbanken.forskningsradet.no/en/project/FORISS/304667?Kilde=FORISS&distribution=Ar&chart=bar&calcType=funding&Sprak=no&sortBy=score&sortOrder=desc&resultCount=30&offset=0&Fritekst=aros. Project number 304667.

Abstract

Autonomous Underwater Vehicles (AUVs) can significantly extend our access to the oceans. For an AUV, having a map of its surroundings may deem necessary to perform certain tasks, such as path planning and collision avoidance. Alas, the performance of some sensors used on land is somewhat impaired underwater. Simultaneously, Visual Simultaneous Location and Mapping (VSLAM) methods based purely on RGB cameras are becoming increasingly accurate. In this thesis we present a surface reconstruction system that uses the output from a VSLAM system to reconstruct a dense 3D model of the scene. Having access to a synthetic underwater dataset, we also present a novel method of emulating VSLAM data on this dataset and similar datasets. In addition, as a step towards reconstructing the surface, we implement a per-frame depth interpolation method, based on the sparse depth samples obtained from the VSLAM data. The final reconstruction is then performed by a third-party software, presented along with a selection of relevant literature. The resulting reconstruction system thus consists of three separate modules, in which one is a third-party software, and all modules can be exchanged for other methods. Utilizing a synthetic dataset, the modular system stands as an example for testing surface reconstruction methods on simulated underwater environments.

Sammendrag

Autonome undervannsfarkoster (AUV-er) kan utvide vår tilgang til havene betydelig. For en AUV kan det være nødvendig å ha et kart omgivelsene for å gjennomføre noen typer oppgaver, som ruteplanlegging og kollisjonsunngåelse. Dessverre blir kvaliteten på noen typer sensorer brukt over vann betydelig redusert under vann. Samtidig har Visual Simultaneous Localization and Mapping (VSLAM) metoder basert kun på RGB kameraer blitt mer stadig nøyaktige. I denne oppgaven presenterer vi et system for å rekonstruere omgivelser basert kun på VSLAM output data. Med tilgang på et syntetisk undervanns datasett presenterer vi også en ny metode for å produsere realistisk VSLAM data fra dette datasettet og liknende datasett. I tillegg, som et steg mot å rekonstruere omgivelsene, implementerer vi et en metode for å interpolere dybden i et bilde fra noen få datapunkter gitt fra VSLAM dataen. Den siste delen av rekonstruksjonen blir gjennomført av en tredjeparts programvare, som blir presentert sammen med et utvalg av relevant litteratur. Det resulterende samlede modulære systemet står som et eksempel for å teste rekonstruksjonsmetoder på syntetisk data.

Acronyms

AIAUV Articulated Intervention Autonomous Underwater Vehicle. 1

AROS Autonomous Robots for Ocean Sustainability. 1

AUV Autonomous Underwater Vehicle. 2, 59

ESDF Euclidean Signed Distance Function. 2, 3, 7, 11, 13, 14, 39, 54, 63, 71, 74, 75, 77, 78

FOV Field of View. 78

IMU Inertial Measurement Unit. 43

JSON JavaScript Object Notation. 24

LiDAR Light Detection and Ranging. 7, 52, 53, 59, 80

SDF Signed Distance Function. 7, 9

SLAM Simultaneous Location and Mapping. 1–4, 16–18, 20, 27, 28, 30, 33, 36, 39, 40, 43, 52, 54, 55, 58–61, 67, 68, 79, 80

TSDF Truncated Signed Distance Function. 3, 7–11, 13, 14, 52, 55, 62, 63, 74

VSLAM Visual Simultaneous Location and Mapping. 2, 5, 43

Table of Contents

1 Introduction

In this thesis, we aim to provide a method for reconstructing a dense 3D model of an underwater scene. The main idea guiding the work in this thesis is to utilize point clouds and pose estimates from Simultaneous Location and Mapping (SLAM) as a basis for the reconstruction. Hence, building on previous work, we provide a method for emulating SLAM output on a synthetic dataset. Furthermore, we provide a system for reconstructing a surface from this data, using RGB images for applying texture. In this chapter, we provide some motivation behind this goal and define some objectives for this thesis. Then, we summarize the main contributions and briefly present the structure of this report.

1.1 Motivation

The world's population is increasing, and several of the United Nations' sustainable development goals entail an increasing need for energy, minerals, and food. Simultaneously, the world faces a grave climate change challenge, which threatens the livelihood of many people, animals, and other organisms on earth. Other forms of pollution, such as plastic waste and particles in the air, make the challenges even more important to overcome.

Conservation and sustainable use of the oceans will play a key role in the development of a sustainable future. Companies are already taking on this challenge in the oceans. Many are looking into new ways of producing energy, either by offshore wind, floating solar, wave energy, or other forms of ocean-based energy installations. The oceans will also be important to provide food, such as fish and seaweed, and companies are also exploring mining for minerals on the seabed.

Despite the increasing need to develop our ocean space, our current access to the oceans through robots is somewhat limited. There is typically a decisive trade-off between autonomy and maneuverability, limiting the use-cases of the robots and, in turn, the use-case of the ocean space. Increasingly, the robots need to operate in unconventional environments with challenging conditions. Many of these environments are human-made, such as a subsea oil installation and pipes, an ocean floor power transformer and cables from offshore wind, and other permanent ocean installations.

Researchers at project AROS have recently developed a highly maneuverable robot, with the asserted goal to make the robot autonomous, resulting in a so-called Articulated Intervention Autonomous Underwater Vehicle (AIAUV).

Full underwater autonomy has proved to be a big challenge. Partly, this can be blamed on difficult maneuvering in an environment filled with disturbances, making

intervention tasks demanding. Perhaps more importantly, underwater autonomy is challenging because of the constraints set by the underwater sensors, which typically have reduced or no functionality compared to the comparable sensors above water. Some sensors may be available for underwater use, but it may not be feasible to install them on an AUV because of cost, weight, size, or other factors.

A camera is typically feasible to install on an AUV, and Visual Simultaneous Location and Mapping (VSLAM) algorithms based purely on an RGB camera are becoming increasingly fast, lightweight, and accurate. Lighting conditions, disturbances, and other aspects of underwater VSLAM can, however, degrade the performance of a VSLAM algorithm; it may even fail altogether. In our case, the most relevant output of feature-based VSLAM algorithms will be a pose and a point cloud, both of which will be incrementally adjusted as the robot moves. This is typically the same as with any feature-based SLAM method. For that reason, we will in this thesis refer to this output as SLAM output data.

A point cloud will often be insufficient for the robot to be able to plan ahead and interact with its environment. For that reason, one may want to *reconstruct* a dense map of the environment in some form. Such a map can be represented in various ways.

One typical representation which has gained a lot of attention is a mesh, which will be presented in section 2.4. There are many reasons for wanting a mesh. One reason is for human interpretability of a scene. If the autonomous robot is sending a video stream to a human operator, a mesh is an intuitive representation for the human. For example, the robot could survey an area and send the structure and texture, represented as a mesh, to the human inspector. If the mesh and its texture are precise enough, one may even be able to perform inspection directly by looking at the mesh. Likewise, a robot may want to represent the surface as a mesh in order to interact with the environment, such as identifying and operating valves. This also hinges on the mesh being precise.

Another useful representation of a scene is the Euclidean Signed Distance Function (ESDF), on which the robot can plan trajectories and perform collision avoidance. The ESDF will be introduced in the selected literature. The surface reconstruction method described in this thesis is able to construct both an ESDF and a textured mesh, based on depth maps, pose information, and RGB images for texture.

1.2 Objective and Contribution

The high-level ambition forming the foundation of this thesis is for a robot to produce a dense map of an underwater environment, using only an RGB camera, in real-time.

Following this aim, we define some objectives for this thesis.

The objective of this thesis is fourfold. First, we return to the specialization project for a more in-depth study of the visual artifacts appearing in the results and a decision on whether to disregard the utilize third-party software. Second, we seek to improve the SLAM emulation method developed in the same project. The emulated SLAM data in the project had several sub-optimalities, such as not performing data association of the points across frames. Also, we construct a sparsification method to emulate a natural distribution and sparseness of detected features. The third objective is to create a simple, per frame, depth interpolation technique, only using information from the image and the points from the SLAM data. The fourth objective is then to combine these two methods – the improved SLAM emulation and the depth interpolation – and, through an identified surface reconstructor, construct a 3D mesh and an ESDF.

The main contributions of this thesis are:

1. **An improved SLAM emulation technique**. The method is improved from the specialization project to include data association, as well as a bucketing-based sparsification scheme, outlined in section 4. The SLAM data is meant to be generated on a synthetic dataset, with ground truth depth maps and pose information. In this thesis, we will utilize a synthetic underwater dataset developed in previous work.

2. **Implementation of a per-frame depth interpolation method based on sparse depth samples**. We will utilize our produced SLAM data on the synthetic dataset to produce per-frame interpolated depth maps of the scene based on the sparse samples of the depth map obtained by projecting the SLAM points with known locations onto the current frame with a known pose. An existing version of such a depth map interpolation technique is presented and implemented in this thesis.

3. **Combining SLAM data with depth interpolation as input for a surface reconstructor**. In this thesis, we will identify a suitable real-time surface reconstructor, which utilizes per-frame produced point clouds and pose data to reconstruct a mesh, a TSDF and an ESDF, as introduced in section 2. The output from the depth map interpolation, along with color information from RGB images and ground truth poses from the dataset, is given as input to the selected surface reconstructor, and tested for variations in important reconstruction parameters. The SLAM emulation and depth map interpolation techniques are also tested separately and discussed.

1.3 Structure of Report

First, in section 2, we review some of the most relevant literature forming the basis of this thesis. This will mainly be related to the surface reconstruction technique, as both the depth interpolation and SLAM emulation are more methodically presented in their respective sections. Then, in section 4, we present the improved SLAM emulation method, along with some techniques that it builds on. In section 5, we present both the per-frame dense depth map interpolation method before showing how the entire system is connected and used as input for the selected surface reconstruction method. In section 6, we present and discuss the results from the surface reconstruction. Finally, in section 7, we add some conclusive remarks and suggest some areas for future work.

2 Selected Literature

In this section, we review some relevant literature related to surface reconstruction from point clouds. The goal is to build our way to understanding the presented surface reconstructor, Voxblox (Oleynikova et al., 2017), and the surface representation it uses.

First, we will present some different ways of classifying surface representations. Then we will study one class of surface representations based on Signed Distance Functions (SDFs), and their discrete counterpart, Volumetric Truncated SDFs. Building this, we present Voxblox, a state-of-the-art surface reconstructor based on these techniques, which will be used to produce the final results in this thesis.

2.1 Surface Representations

There are several different ways of representing surfaces, and also several ways of classifying surfaces based on their representation and properties. Many VSLAM algorithms utilize point clouds for representing the map, as they are usually lightweight and accurate. An example of a state-of-the-art VSLAM system that produces point clouds is ORB-SLAM3 (Campos et al., 2020), which uses the ORB feature detector (Rublee et al., 2011). We aim to utilize pose and sparse point cloud data, as produced by a VSLAM algorithm, to reconstruct a denser map of the environment. In this subsection, we will thus focus on representations of surfaces that are constructed *from point clouds*.[2]

In (Khatamian & Arabnia, 2016), they present three main ways of classifying surfaces constructed from point clouds: (i) explicit vs. implicit, (ii) interpolated vs. approximated, and (iii) anisotropic vs. isotropic. The first classification is of most importance to us, but all three classifications will be summarized. The descriptions are largely based on (Khatamian & Arabnia, 2016).

(i) Explicit vs. Implicit. The difference between whether a surface is explicit or implicit is based on whether the surface is represented with a function that defines the location of each point of the surface. If so, we have an *explicit* surface; otherwise, we have an *implicit* surface.

An explicit surface can, for example, be a height field, i.e. defined as

$$z = f(x, y), \tag{2.1}$$

[2]The material covered in this subsection is collected and rewritten from the author's specialization project (Schmidt, 2021).

where $x, y, x \in \mathbb{R}$.

An implicit surface is usually defined as the collection of points where a function has a particular value, i.e. the points

$$\mathbf{x} \in \mathbb{R}^3 \text{ s.t. } f(\mathbf{x}) = \gamma, \tag{2.2}$$

where γ is a constant value in $\mathbb{R}$.

(Khatamian & Arabnia, 2016) also distinguishes between different subcategories of explicit and implicit surfaces. Explicit surfaces can typically be divided into *parametric* and *triangulated* surfaces. Quoting the paper, "a parametric surface is the deformation of a primitive model that covers an arbitrary portion of the points". This typically involves blending together shapes such as spheres and cylinders to approximate the surface. A triangulated surface, on the other hand, connects points in the point cloud into triangles or tetrahedra. One example of a surface reconstruction method that yields an explicit, triangulated surface is the Ball Pivoting Algorithm (Bernardini et al., 1999).

Implicit surfaces can be divided into *variational* and *typical* surfaces, where the difference lies in the basis functions used to represent the function f in eq. (2.2): Typical surfaces use radially symmetric basis functions, whereas variational surfaces use a variety of basis functions. One important example of a typical implicit surface reconstruction method is Poisson Surface Reconstruction (Kazhdan et al., 2006).

(ii) Interpolated vs. Approximated. Another classification of surface representations in (Khatamian & Arabnia, 2016) is based on whether the surface must intersect points in the point cloud. *Interpolated* surfaces must intersect at least a subset of the points, but approximated surfaces *must* not intersect any points. The distinction is quite similar to the explicit vs. implicit distinction: Interpolated surfaces are typically explicit surfaces, and approximated surfaces are typically implicit surfaces.

(iii) Anisotropic vs. Isotropic. The last distinction presented in (Khatamian & Arabnia, 2016) relates to how "smooth" the surface is. Informally, isotropy regards whether something is similar, or uniform, in all directions. For a surface, this would mean that measurements coming from all directions are the same, implying no sharp edges, corners, instantaneous jumps, or other features where measurements from two angles would not be equal. If the surface has any such features, it is anisotropic. Otherwise, it is isotropic.

2.2 Signed Distance Functions

A distance function, or distance field, is a function that for each point, x, calculates the distance, $\phi(x)$, to the boundary of a set Ω, $\partial\Omega$ (Chan & Zhu, 2005). In robotics applications, this set is usually a physical object in the metric space $\mathbb{R}^3$, but the distance measure ϕ varies from application to application. The sign illustrates whether the point is inside or outside the set. An SDF is illustrated in fig. 1. Here, the set Ω is a 2D shape (blue), and the distance measure is the Euclidean distance. The SDF is illustrated as a function $\phi : \mathbb{R}^2 \to \mathbb{R}$, with negative values inside the set and positive values outside the set. The border of the set can be extracted from the isosurface at $\phi(x) = 0$, which is why the SDF methods can be classified as implicit surfaces.

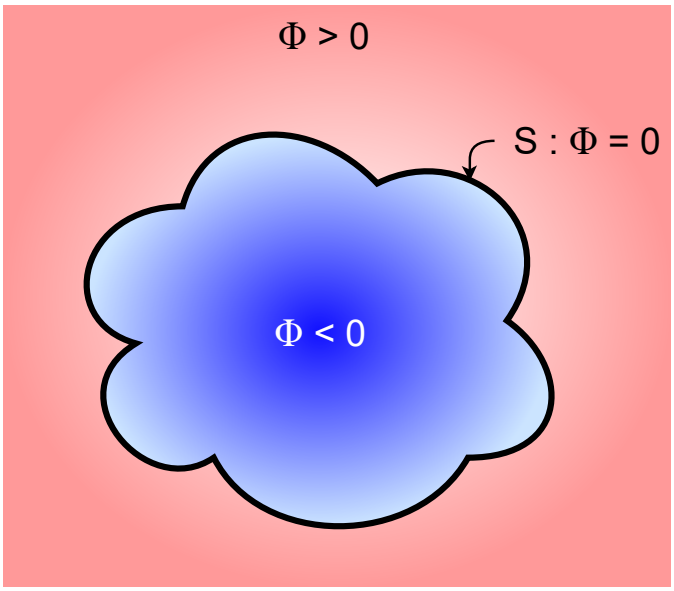


Figure 1: Signed Distance Function (SDF), Φ. Blue colors indicated negative values, red indicates positive values. An isosurface, S, is extracted at $\Phi = 0$.

In this thesis, we will consider two different kinds of SDFs: Euclidean SDFs and Truncated SDFs. In a Euclidean Signed Distance Function (ESDF), the distance measure is the Euclidean distance, as in fig. 1. Discretely sampled ESDFs are commonly used for tasks such as path planning and collision avoidance (Zucker et al., 2013).

In a Truncated Signed Distance Function (TSDF), the distance measure is the distance from to the surface of the model "along the ray direction from the center of the sensor" (Oleynikova et al., 2016). This is illustrated in fig. 2. The TSDF representation arose with the advent of modern scanner sensors, such as LiDARs and RGB-D cameras. Furthermore, only points near the surface have an associated distance value, i.e. the function is *truncated* at a specified distance from the surface.

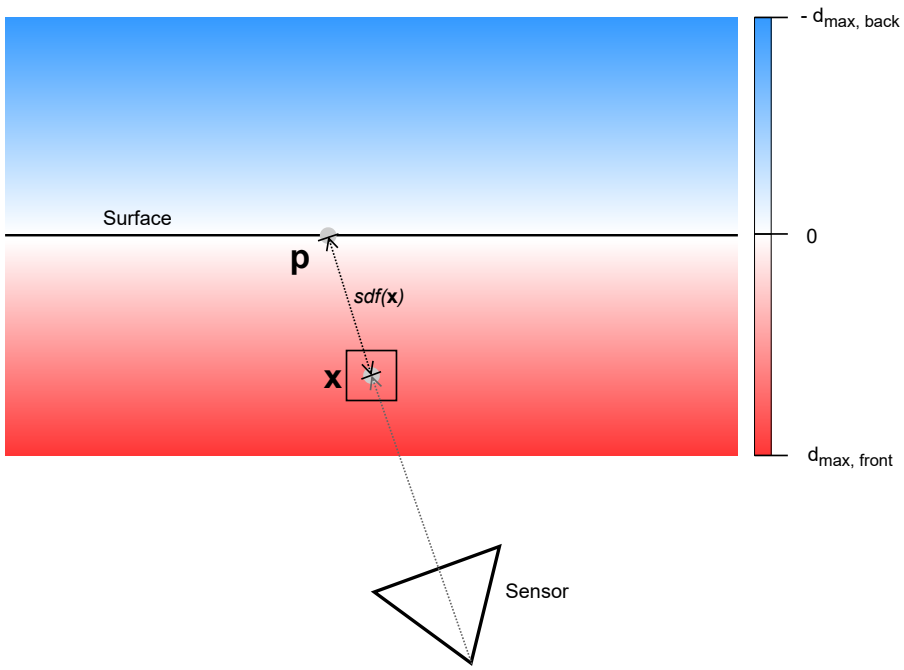


Figure 2: Truncated SDF (TSDF) illustration. The distance from a voxel to the surface is measured along the ray cast from the camera. The function is truncated at a specified distance both in front of and behind the surface.

2.3 Volumetric Approaches to Surface Reconstruction

The *volumetric* methods in surface reconstructions are closely related to the implicit surface representation. Whereas the implicit methods used a continuous function, f, the volumetric approaches represent f as a discrete collection of samples on a 3D-grid (Bærentzen et al., 2012). The 3D grid-cells, or samples, are typically called *voxels*. Figure 3 shows how the volumetric approaches relate to the implicit approaches. As in fig. 1, the function Φ measures some distance metric to the surface of the model, here denoted S. The significant change is that we in fig. 3 sample the function in a grid, discretizing the function. This way, changing the samples will implicitly change the surface, S in a small area around the sample.

Some noteworthy volumetric methods in surface reconstruction are Poisson reconstruction (Kazhdan et al., 2006), OctoMap (Hornung et al., 2013), and the TSDF method presented in (Curless & Levoy, 1996). In section 2.4, we will present Voxblox (Oleynikova et al., 2017), which is the main volumetric method this paper builds on.

2.3.1 Volumetric TSDF

(Curless & Levoy, 1996) presents a method for creating a TSDF from a series of range scans with a volumetric approach. In the paper, the volume is divided into a

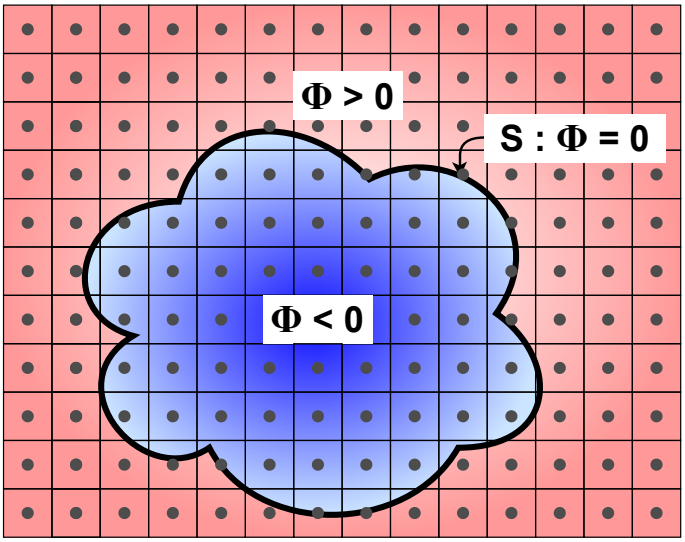


Figure 3: Volumetric SDF. The SDF from fig. 1 is sampled at the center of the voxels, yielding a discrete SDF.

voxel grid, and for each range scan, they update the associated distance value of each voxel. There are two important values associated to each voxel: a distance, $D(\mathbf{x})$, and a weight, $W(\mathbf{x})$. With each scan, we update these values, where $\mathbf{x}$ denotes the center of the voxel.

For range scan i, we obtain a measurement of the distance value of the voxel, $d_i(\mathbf{x})$, and we assign a weight $w_i(\mathbf{x})$ to this measurement. The weight should typically reflect the uncertainty of the measurement. One example of such a weight function is presented in section 2.4. The distance is measured as the distance from $\mathbf{x}$ to the surface along the line of sight of the sensor, i.e. *not* the Euclidean distance from $\mathbf{x}$ to the nearest point of the surface. However, the idea is that with enough such samples, the value can approximate this Euclidean distance. To improve the distance measure, $D(\mathbf{x})$, we use the weights to weigh the measurements, as explained below.

The problem with the TSDF, is that we cannot know whether the projective distance we assign to a voxel is correct. In fig. 4, we are viewing a surface that is almost perpendicular to the line of sight. However, for the illustrated voxel, the surface is actually far closer than the distance along the ray would suggest. If we do not get samples from enough viewpoints, the voxel distance value might end up completely incorrect. This is partially mitigated by truncating the SDF so that voxels far away from the surface are not assigned their actual distance value.

There is information we can utilize to obtain a more accurate distance approximation. In fig. 5, we have illustrated an example where the surface is not perpendicular to the line of sight ray. Intuitively, as the angle between the ray and the surface normal, θ grows larger, we assume that the measurement is more

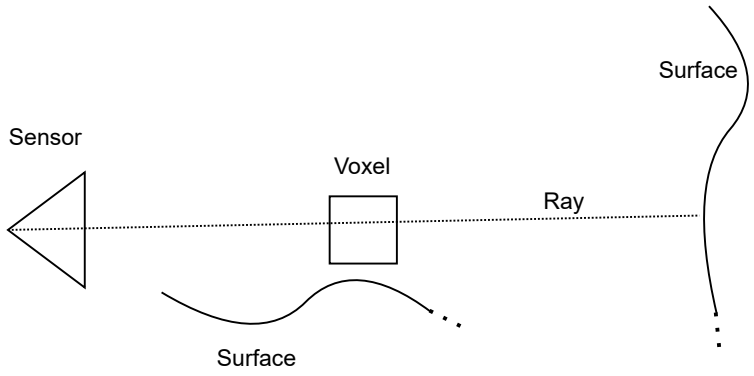


Figure 4: Possible TSDF error. The distance from the voxel to the surface should, in the Euclidean sense, be measured straight down, but is rather measured along the ray, giving a larger distance value.

uncertain. The Euclidean distance from the voxel to the surface is likely smaller compared to the measurement if θ is larger. For example, if θ approaches 90°, the surface is almost tangential to the line of sight, and chances are the viewing ray is close to the surface long before it actually hits the surface, and voxel distance values should take this into consideration.

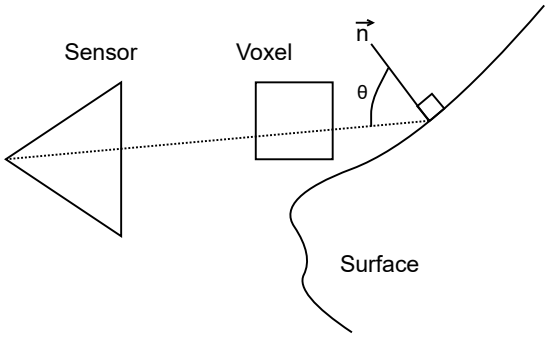


Figure 5: Possible TSDF error because of the angle of the surface relative to the camera. If this angle is large, then the resulting difference between the Euclidean and TSDF distance may be large.

Another point is that the sensor model gives information about which measurements are more certain than others. For most depth sensors, depth information for points far away is more uncertain than for points closer to the sensor (Nguyen et al., 2012).

The weight value of each measurement is a way for the methods to mitigate these uncertainties. These values are used for merging several scans together, forming the TSDF. As we will see, this can be done incrementally.

(Newcombe et al., 2011) discusses taking both of these uncertainties into consideration when weighting and merging the different scans. The weight associated with each measurement can be proportional to $\cos(\theta)$ (taking into consideration the

angle between the ray and surface normal) and inversely proportional to the measurement distance value (taking into consideration the sensor uncertainty model). However, they argue that the use cases for their sensor allow for enough variety in the sensor scans that these weights do not notably affect the reconstruction quality, so instead, they end up with a simple, constant weight assignment.

Having discussed both the distance value and weight value, we can now illustrate how several measurements are merged into a single TSDF. For each voxel, we can take the weighted average of each measurement, i.e. weighting each distance measurement with its corresponding weight. Furthermore, since these methods are usually designed to work in real-time, one can perform this procedure *incrementally*, taking the running weighted average of the measurements so far. For the voxel centered at $\mathbf{x}$:

$$D_{i+1}(\mathbf{x}) = \frac{W_i(\mathbf{x})D_i(\mathbf{x}) + w_{i+1}(\mathbf{x})d_{i+1}(\mathbf{x})}{W_i(\mathbf{x}) + w_{i+1}(\mathbf{x})} \tag{2.3}$$

$$W_{i+1}(\mathbf{x}) = W_i(\mathbf{x}) + w_{i+1}(\mathbf{x}) \tag{2.4}$$

(Curless & Levoy, 1996) also suggest three different states for a voxel: Unseen, Empty, and Near surface. Initially, all voxels are assigned the Unseen state. Then, as we get visibility information by rays passing through voxels and hitting the surface, we can assign voxels as Empty and Near surface. Both Empty and Unseen voxels will have truncated values and no associated weight, but Near surface voxels will have both a distance and a weight.

2.4 Voxblox

(Oleynikova et al., 2017) presents work that partially builds on (Curless & Levoy, 1996), and they have named their method *Voxblox*. There are several contributions in the work: (i) they utilize a new weighting scheme, (ii) they present a new merging strategy, and (iii) they incorporate a way to build an ESDF from the TSDF in real-time, allowing for planning, collision avoidance, and more. In addition, the method makes use of the *voxel hashing* approach presented in (Nießner et al., 2013) to extend the volume (and hence the map) incrementally as more measurements arrive (iv).

(i) The new weighting scheme is inspired partly by (Nguyen et al., 2012), where they find that the noise of a distance measurement varies predominantly with the square of the distance. Incorporating this with the strategy used in (Curless & Levoy, 1996), they arrive at the following weight function:

$$w_i(\mathbf{x}, \mathbf{p}) = \begin{cases} \frac{1}{z^2} & \text{if } -\epsilon < d \\ \frac{1}{z^2}\frac{1}{\delta-\epsilon}(d+\delta) & \text{if } -\delta < d < -\epsilon \\ 0 & \text{if } d < -\delta, \end{cases} \tag{2.5}$$

where z is the measurement depth and δ and ϵ are truncation distances. In Voxblox, they use $\delta = 4v$ and $\epsilon = v$, where v is the voxel size. The function is visualized in fig. 6.

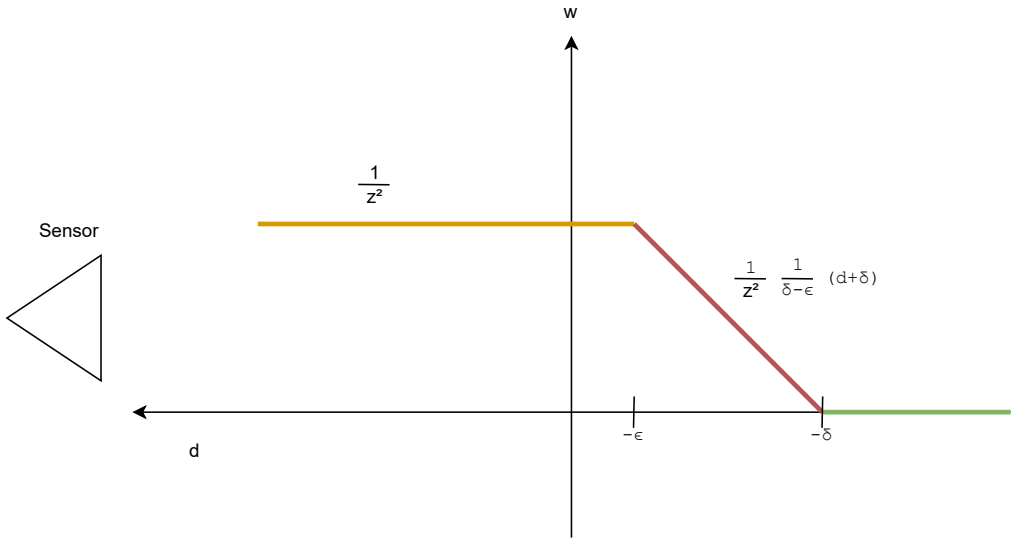


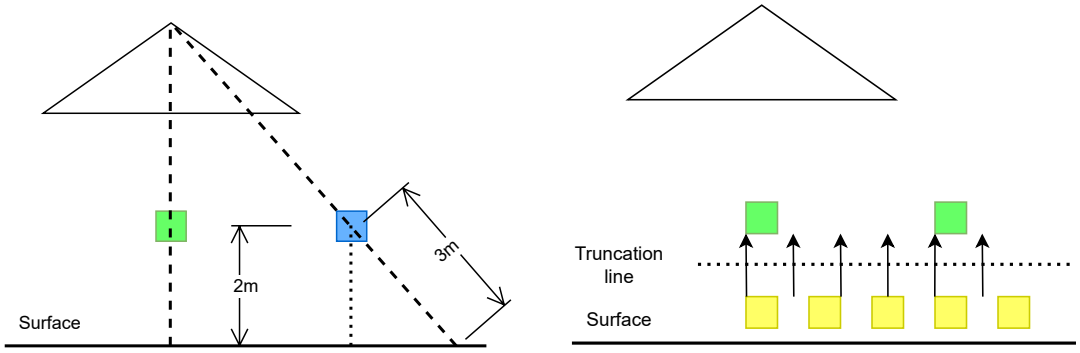
Figure 6: Weight function in eq. (2.5). The value of the weight drops to zero linearly behind the surface.

The idea behind this is to have a constant weight between the sensor and the surface which is proportional to the inverse squared depth of the measurement, then, behind the truncation region, drop off the weight from $1/z^2$ to 0 in a linear fashion, before having 0 weight behind the surface in order not to interfere with other possible surfaces behind the measured one.

(ii) The new merging strategy is designed to speed up the process of raycasting. Raycasting, in this sense, refers to the process of using the depth measurement from a range scanner to cast a ray from the sensor to the measured depth point. The voxels the ray intersects must then be updated with the new measurement. They have called their approach *grouped raycasting*, because all points in the sensor scan that project to the same voxel are averaged and then treated as one ray. That is, both their distance values and color values are averaged together. They state that this "leads to a very similar reconstruction result while being up to 20 times faster than the naive raycasting approach".

(iii) The ESDF construction is based on (Lau et al., 2010), where they update an ESDF based on an *occupancy map*. An occupancy map is, conceptually, a volume where each voxel is marked as occupied or free. They do this by propagating values in *wavefronts*, where such a wave propagation is started when a voxel is either newly occupied or freed. For each neighbor of the changed voxel, one updates the distance value of the neighbors and does this recursively for all neighbors, i.e. each neighbor is added to the wavefront queue. One can either (1) raise the values, which happens when a previously occupied voxel is freed such that the distances will be larger, or (2) lower the values, which happens when a previously free voxel is occupied and the surrounding voxels should resultingly get lower values.

(Oleynikova et al., 2017) extend this to utilize the TSDF as input to the ESDF construction: Instead of starting by propagating values from the occupied/freed voxels, some ESDF voxels get their distance values from their co-located TSDF voxels. These are the voxels where the TSDF distance is below some threshold, γ, and these ESDF voxels cannot have their values changed by a raise or lower wavefront: They are only changed when the TSDF changes. When the distance value in a TSDF is increased or decreased, the ESDF voxels start a raise or lower wavefront, respectively. The wavefront is propagated to all 26-neighboring voxels, i.e. all voxels in the cube around the current voxel, $3 \times 3 \times 3 - 1 = 26$.



(a) Potential error (b)

Figure 7: The ESDF is created directly from the TSDF. Since the TSDF is truncated near the surface, where voxels typically have quite accurate estimates, the ESDF will likely rule out most potential sources of measurement errors.

(iv) (Nießner et al., 2013) hashes the position of a voxel in world coordinates (x, y, z) to a hash value through the hash function

$$H(x, y, z) = (xp_1 + yp_2 + zp_3) \mod n, \tag{2.6}$$

where n is the hash table size, and p_1, p_2, p_3 are three large prime numbers.

Hence, the method can look up voxels in O(1) time (Oleynikova et al., 2017). They also handle hash collisions by inserting pointers between the hash entries with colliding values. When a new scan is arriving, they allocate voxels where the rays traverse, as well as around the new parts of the TSDF, inside a truncation radius that varies with the variance of the measurement.

They then extract a mesh of the surface directly from the TSDF. A mesh is a set of finite elements that are combined to explicitly represent the surface. Typically, the finite elements are two-dimensional polygons. This is also the case in Voxblox.

The mesh is extracted using the *Marching cubes* algorithm (Lorensen & Cline, 1987). The Marching cubes algorithm can utilize the positive and negative distance values in the TSDF voxels to determine whether each voxel is inside or outside the model. Based on these samples, one can determine how polygons intersect through the collection of voxels. The mesh is only built on demand, and is created mainly for human interpretability of the scene.

The entire Voxblox system diagram is shown in fig. 8. Sensor scans and pose estimates are needed for the TSDF to be built in global coordinates. The system then keeps three representations of the map: a TSDF, an ESDF, and a mesh. The latter two are both built directly from the TSDF. Both the TSDF and ESDF are built and updated incrementally as more sensor data arrives in real-time, while the mesh is built on demand. For simplicity, a feature of the program is to set a constant request frequency for the mesh, for example requesting a mesh every second. The mesh is then created from the newest version of the TSDF. The main output from a robotics perspective is the ESDF, which can readily be used by planning algorithms and for collision avoidance purposes.

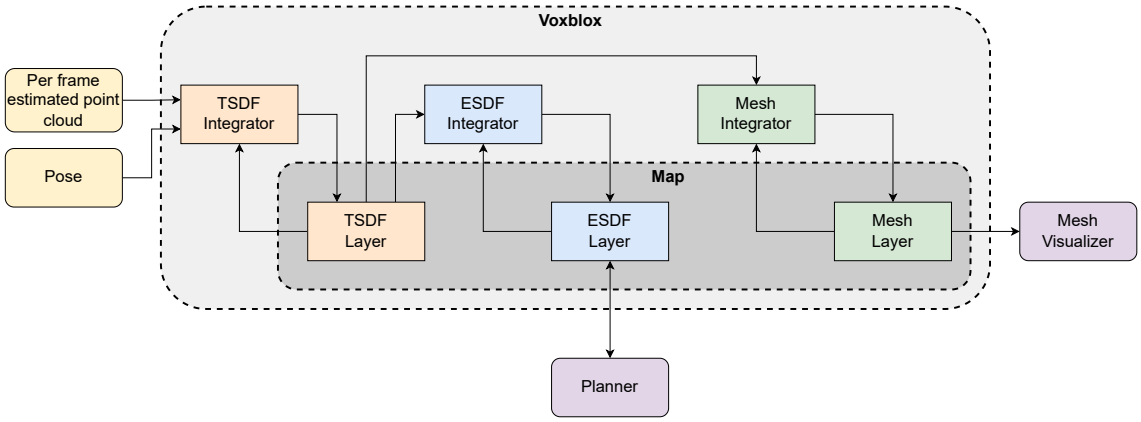


Figure 8: Voxblox system diagram. A per-frame depth map (represented as a point cloud) and pose are given as input, and ESDF and mesh are given as output. Both the mesh and the ESDF are built directly from the TSDF.

2.5 The VAROS Dataset

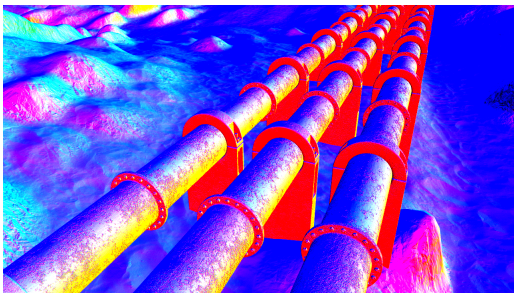
Since the surface reconstruction technique presented in this thesis is meant to be applied by an underwater robot, it is preferable to test the method on realistic underwater data. However, it is difficult to test surface reconstruction in isolation from other errors when using actual underwater footage and measurements.

In previous work, (Zwilgmeyer et al., 2021) has created a simulation of an underwater environment. In this underwater environment, one can lay out trajectories for the simulated underwater robot to follow, and it will generate a video sequence with four types of images: (i) Underwater monocular RGB images, (ii) Uniformly illuminated monocular RGB images, (iii) Surface normal images, and (iv) Depth images. An illustration of these four images is shown in fig. 9. In addition, the dataset provides ground truth pose data, IMU measurements, and depth gauge measurements recorded in combination with the video sequence.

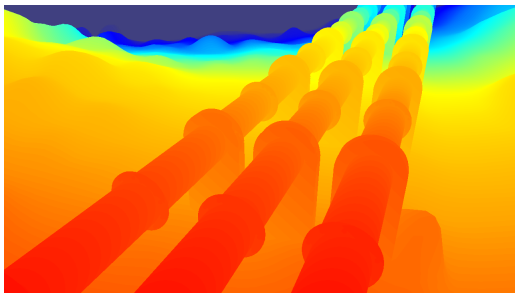
A sequence from the dataset was made available along with the publication of VAROS dataset version 1. It is this sequence we perform the experiments on. We will only utilize underwater RGB images, depth images, and ground truth pose to perform our experiments in section 6.

(a) Underwater RGB image

(b) Uniform lighting RGB image



(c) Surface normal map



(d) Depth image

Figure 9: Sample images from VAROS dataset. Depicts frame 1200 of sequence 01, published with version 1 of the dataset.

3 Revisiting the Specialization Project

In the specialization project, we utilized the third-party surface reconstructor presented in (Piazza et al., 2018a) to attempt to reconstruct a scene of a car and a cube. In the project, we had severe issues relating to visual artifacts. Therefore, the first step of the work towards this thesis was to study these artifacts further and determine whether they were possible to fix or if we should consider disregarding the reconstructor altogether. First, we provide a summary of the specialization project, before delving into the study of the visual artifacts. The results from this study are used to argue for the change of reconstructor.

3.1 Summary of the Specialization Project

In the specialization project, we developed a method for emulating SLAM data on a synthetic dataset with associated depth maps at each frame. The method has two working modes that can be used in combination. The first mode works by randomly sampling a given number of pixel coordinate points per frame, projecting these points into space using the ground truth depth, and describing them in world coordinates. Here, no data association is done on the detected points between frames. On the other hand, one usually obtains a well-distributed set of sampled points of the scene. The second working mode works by detecting ORB features (Rublee et al., 2011) in each frame while still using the ground truth to project the points into space. The use of ORB feature detection allows for simple data association of points by means of matching the descriptors provided by ORB. On the downside, we noted that points detected by ORB tended to be quite concentrated around small patches in the image. A result of these cost-benefit analyses of the two working modes was that we developed the possibility of using the two modes in combination. That means that one can both detect ORB features, with corresponding data association, and randomly sample points, with corresponding point sample disparity.

A second part of the specialization report consisted of identifying a real-time surface reconstructor, and applying this surface reconstructor to the emulated SLAM data provided by our developed method. We argued that a suitable reconstructor was presented in (Piazza et al., 2018a). The required input to the reconstructor was, for each frame: (i) a list of detected points from this view; (ii) the world coordinates of these points; (iii) the pose at the current view. Also, the intrinsic camera matrix was provided, which for our purposes remained the same throughout all frames. A high-level workflow of the entire system is shown in fig. 10.

The result obtained from using our constructed SLAM data as input to the reconstructor was somewhat surprising. For this thesis, we restrict ourselves to a

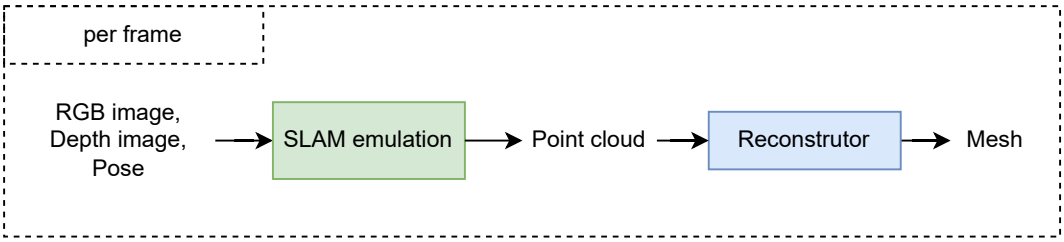


Figure 10: Workflow of the system in the specialization project. The SLAM emulation was built in the project, and the reconstructor was third-party software.

brief description, as more details are given in the specialization project. In general, we observed that the scenes were in fact reconstructed quite well, depending on parameters of the reconstruction and SLAM emulation. A dense sample, along with a reasonable set of reconstruction parameters, gave, in general, quite accurate results. However, the results were completely spoiled by what we called *visual artifacts*, an example of which is shown in fig. 11.

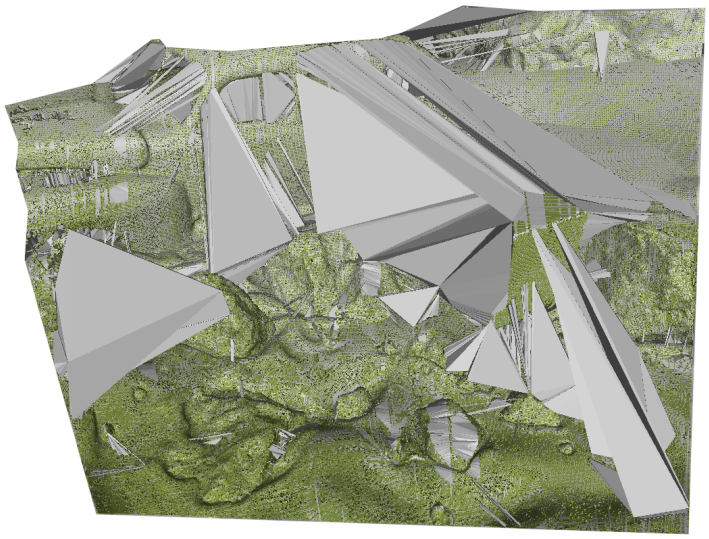


Figure 11: Example of a reconstruction cluttered by artifacts.

Alas, in the specialization report, we did not conduct a proper analysis as to the root of these artifacts. Hence, we will conduct such a study here, in the following subsection.

3.2 Issues and Future Work: Visual Artifacts

One line of future work mentioned in the specialization project was to study the appearance of the aforementioned visual artifacts. In this chapter, we seek to thor-

oughly investigate the causes of the artifacts and justify our reason for abandoning the surface reconstructor used in the specialization project.

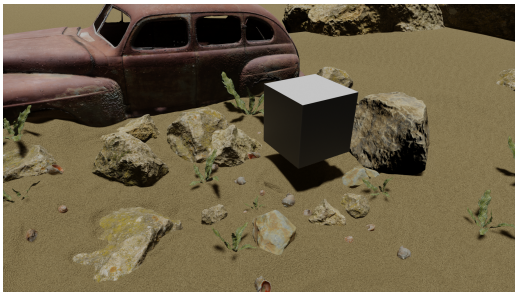
We started by investigating the code for emulating the SLAM data and adding some visualizations. First, recognizing that, at each frame, a set of points is projected into space, a visualization was added for the 3D position of this set of points. A dense sample could then provide a sanity check by a qualitative comparison between the visualization and the imagined 3D structure of the scene. This sanity check could be repeated for each frame, a tedious but somewhat informative procedure. Building on this, we added a visualization that included all points added to the model for each frame to visually check that the points were merged together sensibly, indirectly also providing *some* confirmation of the correctness of the pose transformations, albeit not being sufficient for a proof. Indeed, none of these visualizations were proofs that the method was correct, but they provided some more basis for not doubting so. In addition, we noted the largest and smallest coordinates of the points to make sure we did not have strange far-away outliers. After inspecting "many" of these models, we increasingly gained confidence that, at least, the SLAM emulation was working properly. Some visualizations are shown in fig. 12, which also depicts two RGB images of the scene. The dataset was named the Car-Cube dataset in the specialization project, and is a precursor to the VAROS dataset, and depicts a uniformly lighted scene of a car and a cube, each frame accompanied by a ground truth depth map and pose.

In the following paragraphs, we perform three tests. The results from these tests constitute our argument for abandoning the reconstructor. Each of the tests will give a preliminary conclusion and a foundation for further tests. Although there still may be some undetected error in our SLAM data, we will show that the reconstructor is fundamentally unreliable and unstable. We summarize the tests and results in advance:

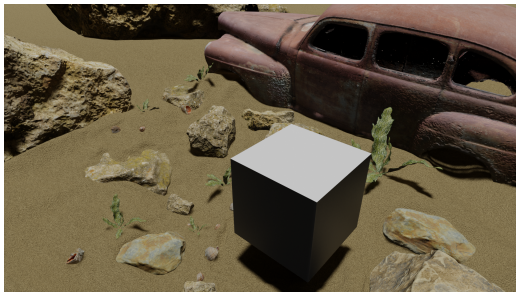
Test (A): Adding one point at a time, from stationary pose.
Conclusions: Reconstructor inexplicably removes points; artifacts appear despite no new points added to the area; arbitrary failures in reconstructor.

Test (B): Adjusting pose given to reconstructor.
Conclusions: All other pose configurations tested decreased performance; cannot definitively assert what units the reconstructor uses.

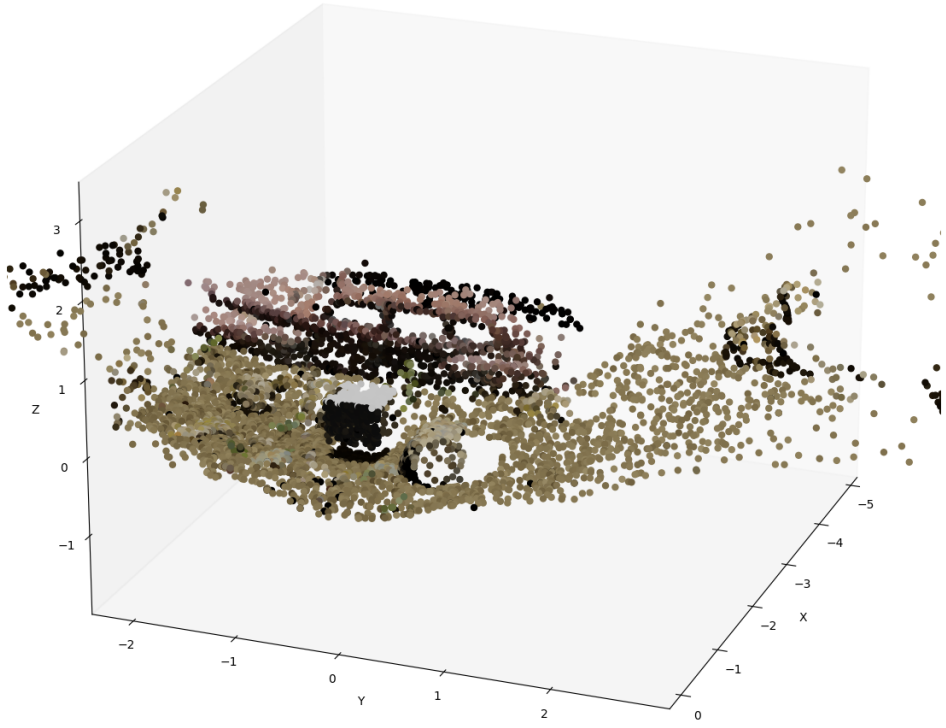
Test (C): Adjusting reconstruction parameters.
Conclusions: Key elements of the reconstruction were inexplicably lost in some reconstructions.



(a) Frame 200 of the Car-Cube dataset



(b) Frame 259 of the Car-Cube dataset

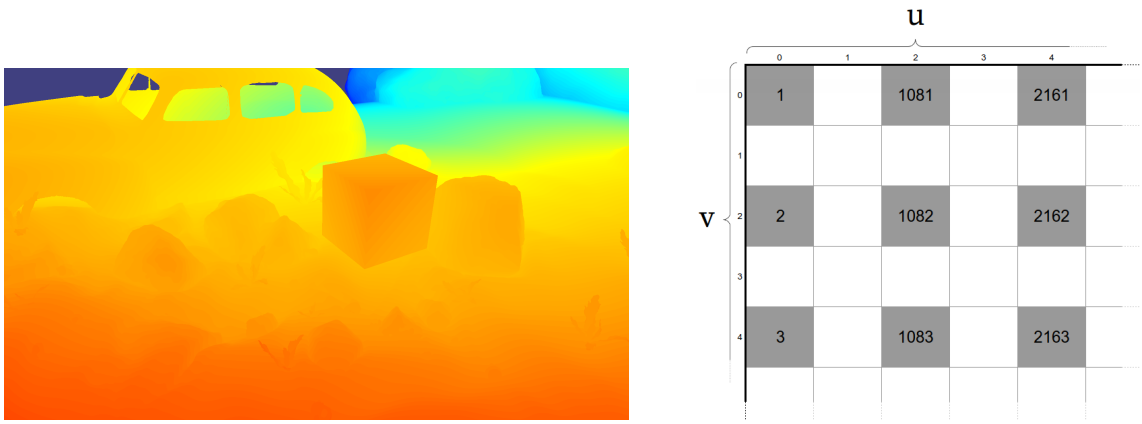


(c) Resulting point cloud from 60 frames

Figure 12: This reconstruction sampled 100 points randomly in each frame over 60 frames, i.e. frames 200 to 259.

Test (A): Adding one point at a time, from stationary pose. We reasoned that one reason why the triangulation behaved strangely in the reconstruction might be that we added an abundance of points simultaneously, deteriorating the reconstructor. Therefore, we designed a test in which one point is added to the reconstructor at a time, allowing it to update the surface after every point is added. For this test, we chose to keep the view, i.e. pose, stationary, implying that the depth image was the same for each frame. This made it easy to spot exactly when and where artifacts occurred, as well as assessing the correctness of the pose. At the same time, the large artifacts should still not arise.

The test was conducted by simulating the following SLAM output. Viewing the simulated scene from a stationary pose, we sample exactly one point per frame. The points are sampled top-to-down and left-to-right, acting as a scan from left to right in the depth image. We thus ended up with the same number of frames as number of points: 508 905. The depth image and the sampling pattern used is shown in fig. 13.



(a) Depth image (b) Sample pattern

Figure 13: Viewpoint and sampled pixels pattern for test 2: Adding one point to the model at a time from a stationary view.

The resulting 3D point model is shown in fig. 14. Using the python library *matplotlib* for visualizing the model is not very versatile and does not let us directly compare to the depth image by adjusting to the correct pose, as many points would then be out of view for the matplotlib-visualizer. However, it does let us conduct some sanity checks. For example, we see that the scale of the model looks appropriate. Furthermore, we confirm that the position from which the scene was viewed is at least negative in the y-direction and positive for the z-direction, as indicated in fig. 14.

In fig. 15a, the result of the full reconstruction is shown. As we can see, the result is again completely spoiled by the visual artifacts. In fig. 15b, we make

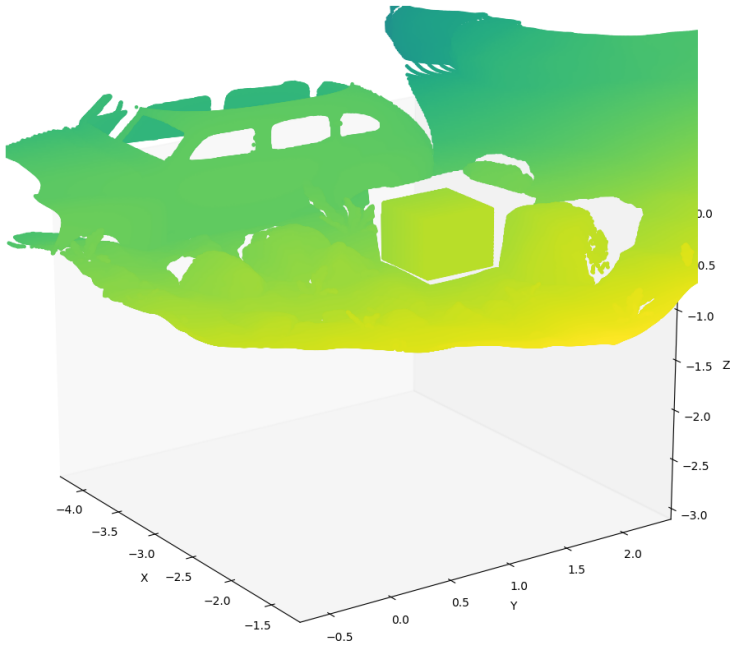
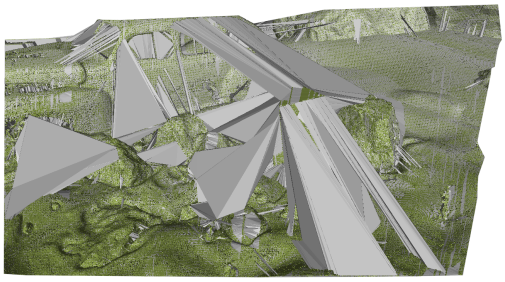


Figure 14: 3D point cloud created from one view with the sample pattern described in fig. 13b. The point cloud is so dense that the individual points cannot be distinguished.

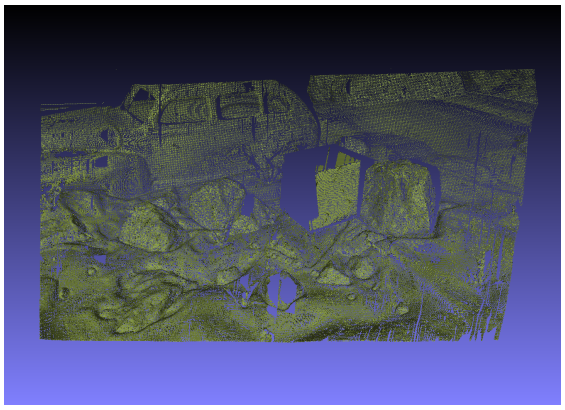
an interesting observation. This figure shows only the *points* involved in creating fig. 15a. Several of the points in the model are missing. This is especially visible for points on the cube. We know that the number of points passed to the reconstructor is correct, and we have no reason to believe that just some of the points should be moved. So the points in the reconstructed model should be the same as in fig. 15, but some points have been removed. Blaming the lack of documentation for the reconstructor, we do not know why it takes the liberty to remove these points from consideration.

In fig. 16 we have shown some snapshots of the model construction output underway from the reconstructor, to produce fig. 15a. In fig. 16a, we have shown the area of the image in fig. 15a that we focus on. An artifact has appeared in fig. 16b, on the edge of the model, indicated by the red circle. This is after 100 000 points have been added to the model, starting from the left. In fig. 16c, the artifact is removed. Now, the strangest part of this figure is that in fig. 16d, a similar artifact reappears. Furthermore, it appears in an area where *no new points have been added*. This happened several times during the reconstruction. One logical explanation is that the reconstructor falsely labels the tetrahedra containing the artifacts triangles as occupied space for maintaining the *manifold property* of the surface, meaning that the surface cannot cross itself or in other ways act non-physically.[3] This is false labeling strange, since all points are added from the same pose. In any case, the

[3]For more information about manifoldness of triangulated meshes, we refer to (Lhuillier, 2014)



(a) Surface reconstructed from one view with the sample pattern described in fig. 13b.



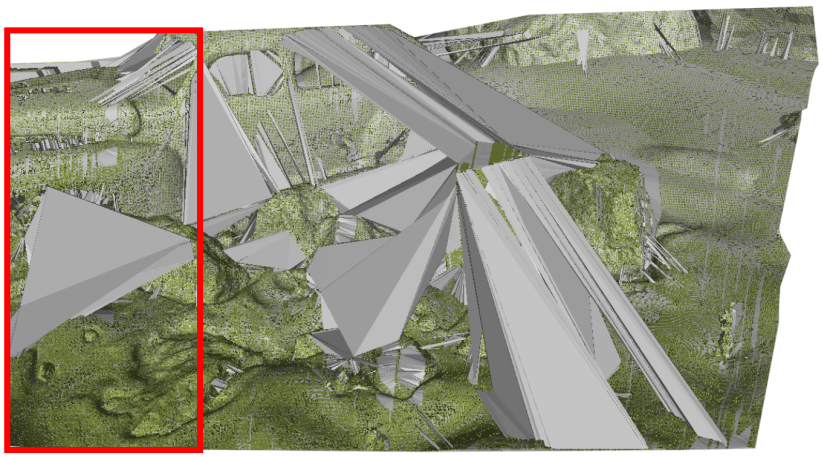
(b) Points involved in the surface reconstruction (a). Some points have been removed from the model.

Figure 15: Reconstructed surface (a) and points in the resulting model (b) when given the points shown in fig. 14 inserted one at a time. Compared to fig. 14, we see that some of those points have been inexplicably removed from the final model.

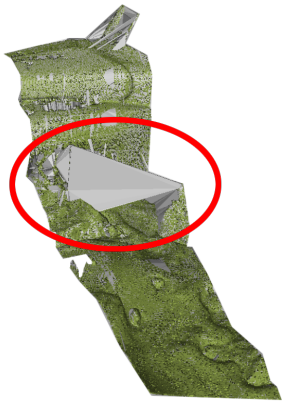
test shows that the reconstructor falsely reintroduces some artifacts after they have been correctly removed, while the points involved in the artifact have not changed.

Upon some trials, the reconstructor actually completely failed, something which had not happened during the specialization project. By "fail", it is meant that all points in the model, and accordingly the mesh, are deleted. Astoundingly, this failure depended on a single adjustable parameter, `initial_manifold_update_skip`, in a way that was very surprising to us, and at this point, a lot of confidence was lost in the correctness of the reconstructor. `initial_manifold_update_skip` determines how many frames are added before the manifold is updated with all the new points. The results were the following: If we start updating the manifold mesh early, say, from frame 2, the reconstructor failed between frames 7501 and 7750. We only save the mesh every 250 frames, so we could not say exactly at which point the reconstructor failed. Up until that point, it would work in the usual way. However, when waiting until frame (and thus also point) 7500 until updating the mesh, the reconstructor would *not* fail. We do not consider this to be anything other than erroneous behavior.

Reason 1 for abandoning reconstructor: Upon some tests, the reconstructor *failed* if updating the mesh early and succeeded if updating the mesh after many points were added. This is objectively unreliable behavior, and we lose confidence that the reconstructor actually works properly.



(a) Part of the image to study below.



(b) Reconstruction after n = 100 000 points added to model from the left

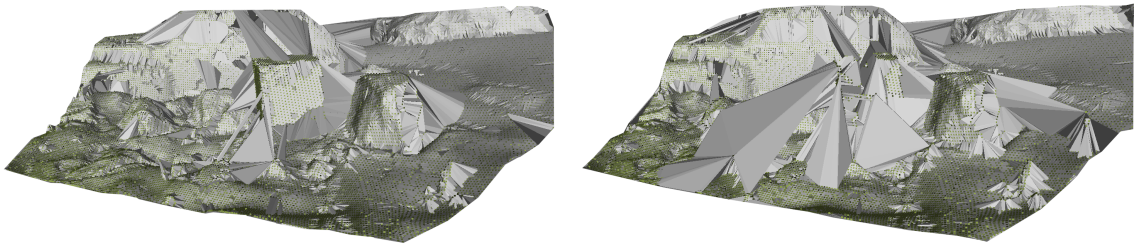
(c) Reconstruction after n = 110 000 points added to model from the left

(d) Reconstruction after n = 120 000 points added to model from the left

Figure 16: Snapshots of the model creation process, after n number of points added to model from the left. The top image shows which part of the image we focus on. The studied artifact is marked with a red eclipse. Strangely, the artifact reappears after no points have been added where it is located.

Test (B). Adjusting pose given to reconstructor. The next source of error could be the production of the JSON file. Many things could go wrong here, and we believed the two most reasonable potential sources of error to be (i) wrong units for focal length and principal point and (ii) wrongly loaded pose. Changing units in focal length immediately gave very strange results. We tried converting units to pixel values, and we also tried representing the values as millimeters, centimeters, and meters. We could not see any other units that gave proper results.

We tested four different variations for the pose to test the second possible error. First, we ran it with the regular pose as used in the specialization project. Second, we transposed only the rotation matrix. Third, we inverted the entire pose. Fourth, we negated the translation. Two results are shown in fig. 17.



(a) Regular pose (b) Inverted pose

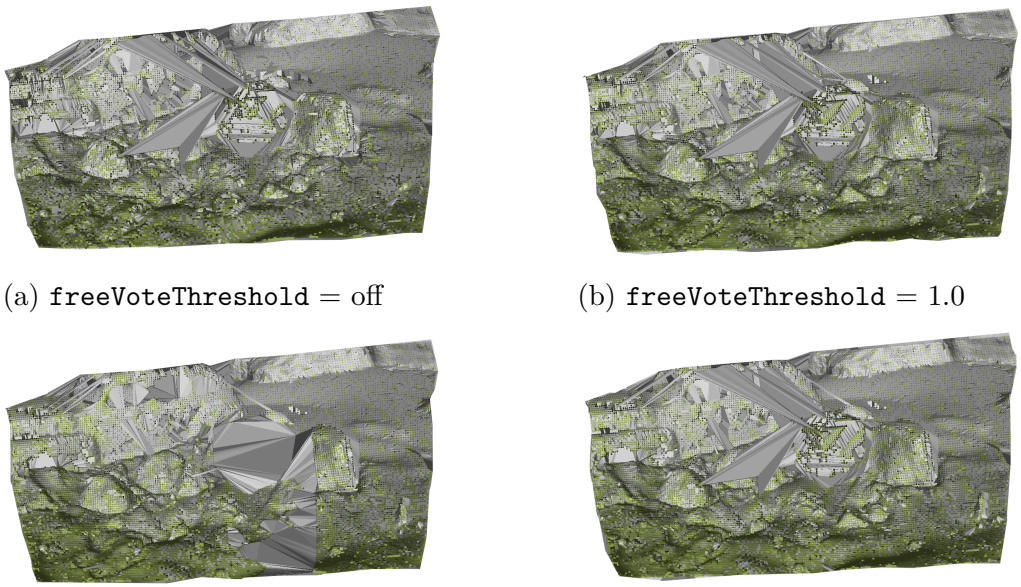
Figure 17: Result from adjusting the loading of the pose into the reconstructor. (a) shows the result as obtained from the specialization project. In (b) we tried inverting the pose before loading it to the reconstructor, in case it used the the inverse coordinate frame. Changing the rotation matrix did not affect the results.

Firstly, a simple transpose of the rotation matrix did not affect the result. This can be explained by the fact that the reconstruction is not affected by the rotation but rather the position of the camera relative to the points, as this is what conditions the rays. Any change to the rotation matrix will therefore not affect the results. The result obtained by inverting the pose before writing it to the JSON is shown in fig. 17b. We see that there are more artifacts and that more structure is lost, which is especially prominent in the cube. Lastly, negating the translation before writing the pose to the JSON produced unintelligible results. Since we still had somewhat intelligible results when inverting the pose, we cannot convincingly conclude that the pose or its units are loaded on the format required by the reconstructor. However, upon inspecting the source code of the third-party software, we could not gain real insight into the required units or pose representation.

Reason 2 for abandoning reconstructor: Changing the pose increased the number of artifacts, but did not completely spoil the reconstruction. We cannot assert with confidence that the units are loaded correctly into the reconstructor.

Test (C). Adjusting reconstruction parameters Another source of potential errors were the reconstruction-parameters. Specifically, we were not really certain of what the different parameters in the configuration file were doing.

The test was conducted by adjusting the `freeVoteThreshold` parameter. The parameter seemingly sets a threshold on what value a tetrahedron must have before it is labeled *free* space. This, in turn, relates to what weight the reconstructor assigns a tetrahedron which a ray intersects, as well as the weight of its neighbors.



(a) `freeVoteThreshold` = off (b) `freeVoteThreshold` = 1.0

(c) `freeVoteThreshold` = 1.5 (d) `freeVoteThreshold` = 2.0

Figure 18: Result from changing the `freeVoteThreshold` parameter.

These results were surprising and interesting. We see that for one value (i.e. 1.5, but this depends on the value one gives to the weights so we do not put much emphasis on the absolute value of it but rather on the relation to the other values), the cube completely disappears. This was a testament to our incomplete understanding of the reconstructor and another strong argument for not using it. It can be noted, however, that we were able to bring the cube back by increasing another parameter called `vertexRemovalThreshold` from 100 to 1000.

Reason 3 for abandoning reconstructor: We could not gain a complete understanding of how the reconstruction parameters affected the results.

As a conclusive remark, we note that the third-party reconstructor is open-source and available on GitHub (Piazza et al., 2018b). However, the repository is large, and the code is complex. We could not confidently navigate the system's source code. Choosing instead to perform some tests, we have provided three main reasons for disregarding the reconstructor in the rest of the work towards this thesis.

4 Improved SLAM Emulation Method

4.1 Point Projection

As the VAROS dataset comes with the ground truth pose and depth map, we can utilize this to project points into space or to project 3D points onto the image plane.

Furthermore, we assume an ideal camera with no skewing or distortion parameters. We can then use the pinhole camera model, illustrated in fig. 19, to obtain the pixel coordinates of a 3D point:

$$u = f_x \frac{X}{Z} + c_x \tag{4.1a}$$

$$v = f_y \frac{Y}{Z} + c_y, \tag{4.1b}$$

where c_x, c_y are constants for translating the coordinate frame from the center of the image to the corner, yielding only positive values for u and v. Writing the camera matrix as

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

we get

$$K \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} f_x X + c_x Z \\ f_y Y + c_y Z \\ Z \end{bmatrix} = Z \begin{bmatrix} f_x \frac{X}{Z} + c_x \\ f_y \frac{Y}{Z} + c_y \\ 1 \end{bmatrix} = Z \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}, \tag{4.2}$$

the last equality corresponding to eq. (4.1).

In our case, the intrinsic camera matrix was, in pixel values

$$K = \begin{bmatrix} 985.5 & 0 & 640 \\ 0 & 985.5 & 360 \\ 0 & 0 & 1 \end{bmatrix},$$

where we can navigate between pixel and meter units by multiplying by the a constant, `PIXELS_PER_METER`.

In addition, the pose of the camera must be taken into consideration when switching between what we in this thesis call *camera coordinates* and *world coordinates*. The transformation from one coordinate system to another is given through a transformation matrix,

$$T_{wc} = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix},$$

where $R \in SO(3)$ defines the rotation and $t \in \mathbb{R}^3$ defines the translation.

Points can then be transformed from world to camera coordinates through the transformation

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} T_{wc} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \tag{4.3}$$

See fig. 19 for an illustration of the pinhole camera model. This shows both the transformation, T_{wc}, and the projection of a point, either from the world and onto the camera, or the other way around. The principles are the same either way: It is based on similar triangles, as shown in the figure.

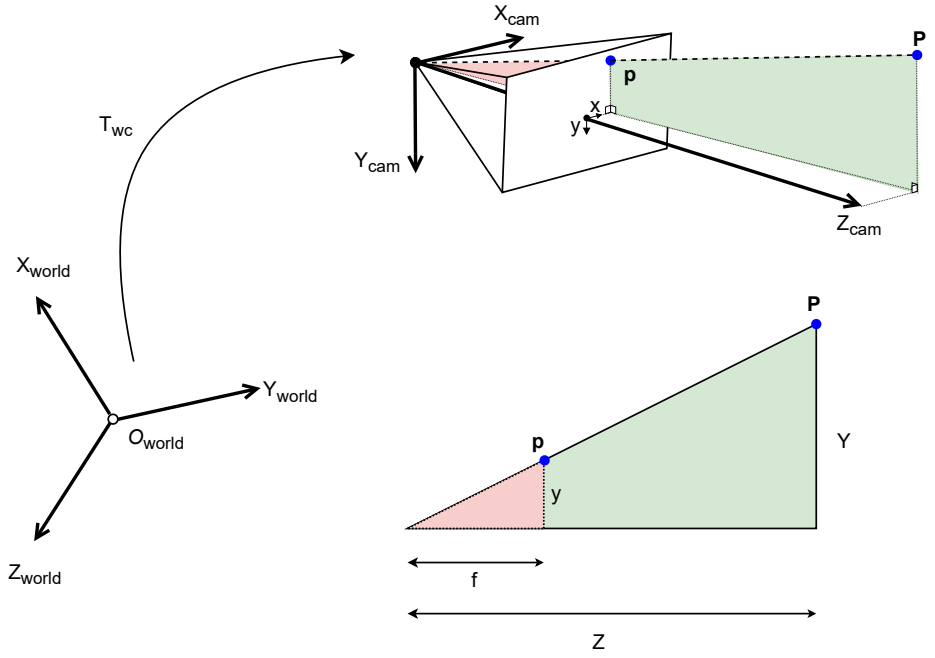


Figure 19: Pinhole camera model. The point **P** is projected onto the camera plane to obtain point **p**. Similarly, point **p** can be projected into space to obtain **P**. In both cases, we use eq. (4.3). Coordinate frames are converted using T_{wc}.

4.2 Tracking

As mentioned in section 3.1, we have in the specialization project developed a method of emulating SLAM data. This method had several drawbacks. Importantly, it did not sufficiently emulate most modern SLAM methods, because it did

not *track* points across frames. Rather, it resampled points each frame and could do feature matching on the sampled points with the points in the model in the ORB detection mode. With the random sampling mode, no data association was performed, and a new set of points were added to the model at each frame. Considering a scenario where the robot is stationary, this is clearly unwanted behavior because we would keep adding points to the model, perhaps even the same point many times.

Therefore, we decided to improve the method to include tracking of points. Assuming an ideal performance of the potential SLAM algorithm, we did not perform feature matching. Rather, data association was performed by projecting points in the model onto the image plane. If the projected point matched the ground truth depth at that pixel, we conclude that the point has been successfully tracked. However, if it does not match, it means that the point has been *obstructed*, and the point track is lost permanently.

The workflow is illustrated in fig. 20. The initialization sequence begins with the user selecting a frame to begin the system. ORB features are detected on that frame and then projected to 3D. A slight modification where we sparsify the data slightly is discussed below. Note that, for simplicity, we do not require a point to be observed across several frames for it to be added to the model. The entire algorithm is summarized in pseudo-code in algorithm 1.

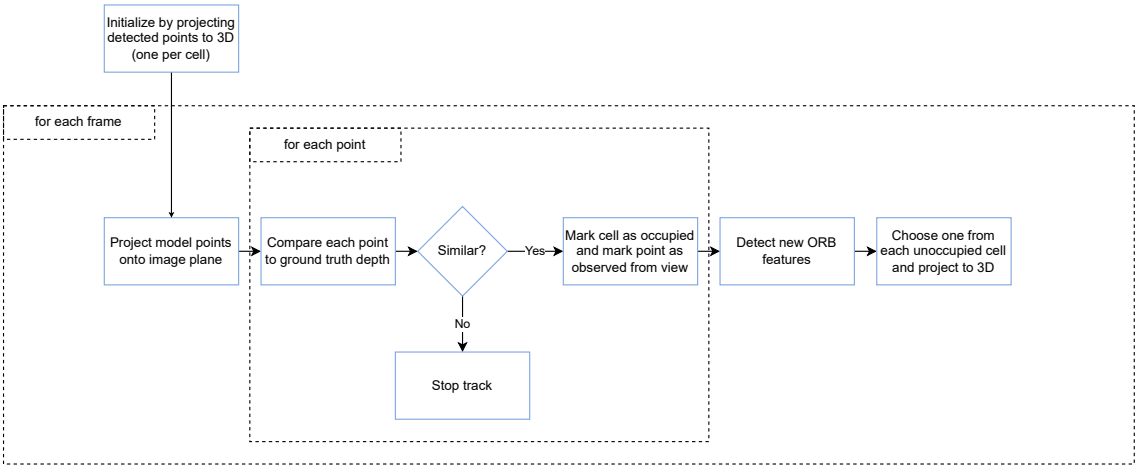


Figure 20: SLAM emulation workflow chart.

For a visual illustration of the algorithm, see fig. 21. Here, at Frame 1, we initialize the system. Three points are detected (red) and projected to 3D (blue). These 3D points are added to the model, and their track is initiated. Then, at Frame n, the three points in the model are projected onto the frame. Then the depths from the projections are compared to the ground truth depth. For points 1 and 2, the two depths are equal. In contrast, we see that point 3 has been obstructed by an

Algorithm 1 Tracking algorithm pseudo-code

```
Detect ORB features in entire image
Divide image into cells
Choose one ORB feature from each cell and project into space
k ← Number of frames
while k ≥ 0 do
    n ← Number of points in model
    while n ≥ 0 do
        Project point onto image
        Similar ← Compare to ground truth depth
        if Similar then
            Mark cell as occupied
            Mark point as tracked
        else
            Stop track of point
        end if
        n ← n − 1
    end while
    Detect ORB features in entire image
    m ← Number of detected features
    while m ≥ 0 do
        if Cell of point is occupied then
            Disregard point
        else
            Mark cell as occupied
            Project point to 3D and add to model
        end if
        m ← m − 1
    end while
    k ← k − 1
end while
```

object, in this case a grey box. Thus, the projected depth will not be equal to the ground truth depth. This is marked by a red cross in the figure. The algorithm will then stop the track of point 3 until potentially it is observed again.

When this projection and comparison procedure is completed, we resample feature points for the image at Frame n (red), and the new detected points are projected into 3D (blue). In reality, the points would have to be tracked for a couple of frames in order to be localized in 3D. For simplicity, we project the points the first time we detect them. Hence, we somewhat optimistically implicitly assume that all newly detected points could be tracked for a couple of frames for the data to be truly correct. As the focus of this report is on the surface reconstruction, we do not emphasize this point further.

Now we discuss the mentioned sparsification modification, henceforth referred

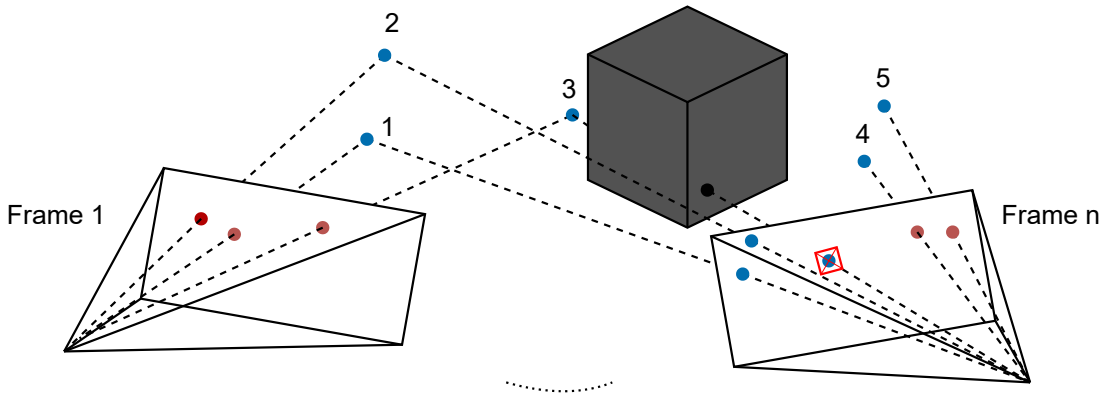


Figure 21: The tracking algorithm. Track of Point 3 is lost at view n, and track of Point 4 and 5 is initiated. The figure does not illustrate the bucketing system.

to as *bucketing*. The bucketing system is implemented in order to maintain a natural sparseness of the data. The image is divided into small *cells* of e.g. 15x15 pixels, and in this cell, we disregard all points except one. For an image of size 1920×1080, this cell-size would result in $\frac{1920 \times 1080}{15 \times 15} = 9216$ cells, whereas a typical SLAM-algorithm detects features in the magnitude of 2000 (Mur-Artal et al., 2015).

Figure 22 illustrates the bucketing system. In this example, we divided the image into 15 cells in total, marked 1 to 15. There are 4 points in the model. These points are projected onto the frame and compared to the ground truth depth. We see that in cells 6 and 14, the projected and ground truth depth are equal, marked with a blue background color. These points are *successfully tracked*. In cells 7 and 8, the projected depths are *not* equal to the ground truth depth, marked with a red cross. The track of these points is thus stopped.

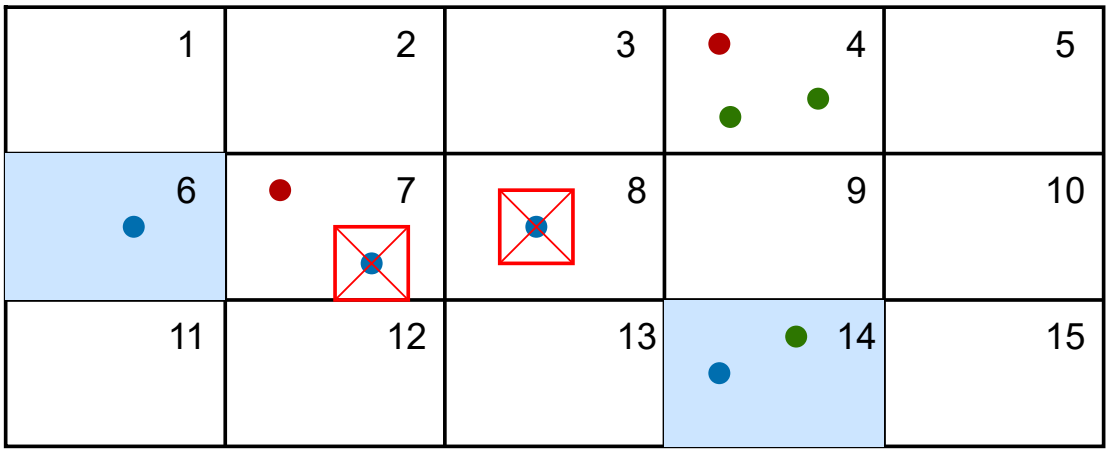
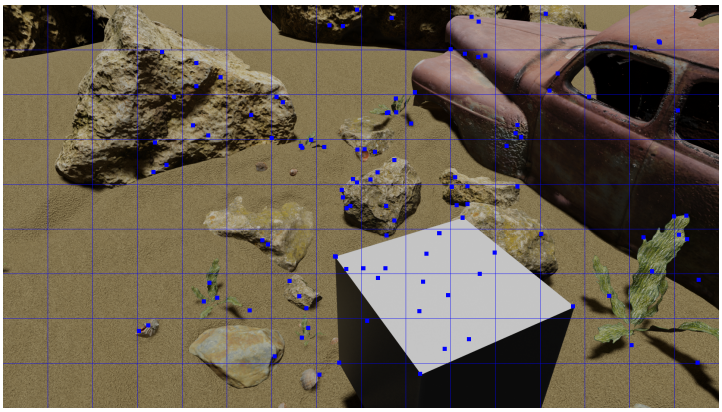


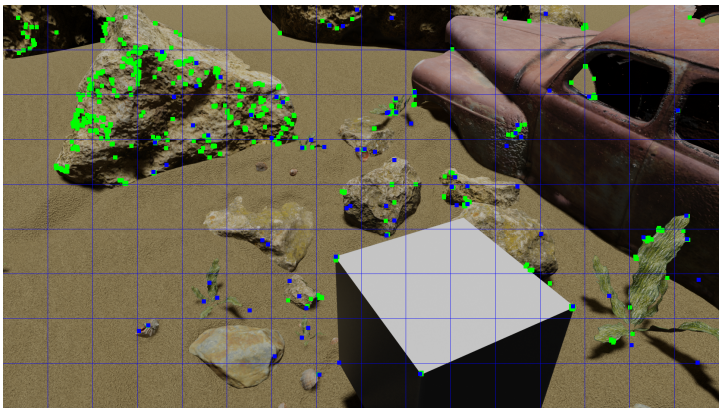
Figure 22: Bucketing scheme. Tracked model points are colored blue. Points where the tracked is stopped is marked with a red cross. New points added to model are colored red, disregarded points are colored green.

Now, the point of the bucketing appears. After we have projected and compared the model points, we resample features in the image, marked red and green. In cells with successfully tracked model points, i.e. cells 6 and 14 in fig. 22, we disregard all newly detected points. In cells where no model points are (successfully) projected onto, i.e. all cells except 6 and 14 in the figure, we choose at most one new detected point to add to the model. New points added to the model are colored red, and disregarded points are colored green.

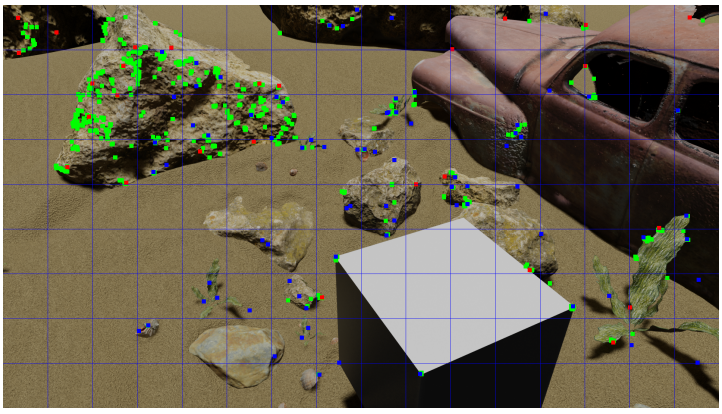
A visualization of the entire workings is shown for one frame in fig. 23. In fig. 23a, the model points (blue) are projected onto the current frame and compared to the ground truth depth. We see, for example, that a lot of points behind the cube are projected. In fig. 23b, we note that the obstructed points (e.g. behind the cube) have been removed. After this removal, we detect points in the entire image (green). Then, in fig. 23c, we select one point for each cell where no model points are correctly projected and project them to 3D (red).



(a) Project points in model (blue) onto current frame and detect where the depth do not match the ground truth depth, i.e. obstructed points.



(b) Remove obstructed points from consideration. Detect points (green) in the entire image.



(c) For cells where no model points (blue) are projected, add one new detected point (red) to model per cell.

Figure 23: Illustration of the tracking algorithm for an example frame from the Car-Cube dataset. Cell borders are marked in blue.

5 Depth Interpolation and Reconstruction Method

In this section, we first present the implemented method for interpolating depth from a sparse set of points obtained from the emulated SLAM data presented in section 4. Then, in section 5.2 we illustrate how the entire system is put together, from SLAM data, to depth interpolation, to surface reconstruction, using the third-party software Voxblox for the final reconstruction.

5.1 Per-Frame Depth Interpolation

The per-frame depth interpolation relies heavily on (Ferrera, 2019) and is most accurately described as a python implementation of the method described in chapter 6.3.1.3 of the paper, called "Depth Map Densification". The method can, with minor adjustments, be summarized with the following steps, where each step is elaborated on below:

1. Obtain the set of observed points from the SLAM algorithm (and corresponding depths).
2. Perform a 2D Delaunay triangulation of the observed points.
3. Interpolate the depth in each triangle
4. (Optional) If the point is "too far" away from the sparse measurements, do not interpolate the depth. We refer to this as *cropping* the interpolated depth image.
5. Convert 2D depth map into dense 3D point cloud.

An illustration of a workflow of the four first steps is shown in fig. 24, marked with their respective step in the mentioned procedure. Steps 2-5 are illustrated in fig. 25.

Step 1. is performed by projecting the successfully tracked 3D points onto the image plane, using the pose and camera matrix and the pinhole camera model.

The original paper also performs a keypoint densification step before triangulating. The reason is to obtain more 3D keypoints, and thus more depth information, before triangulating. They use optical flow to exclude outliers and determine camera poses and 3D positions. As we have the ground truth depth and pose, we do not have to use any special techniques to determine pose and exclude outliers. For densification, we could for example detect keypoints in the image and use all of them, with ground truths, for triangulation. We did in fact perform some tests with the more dense keypoints.

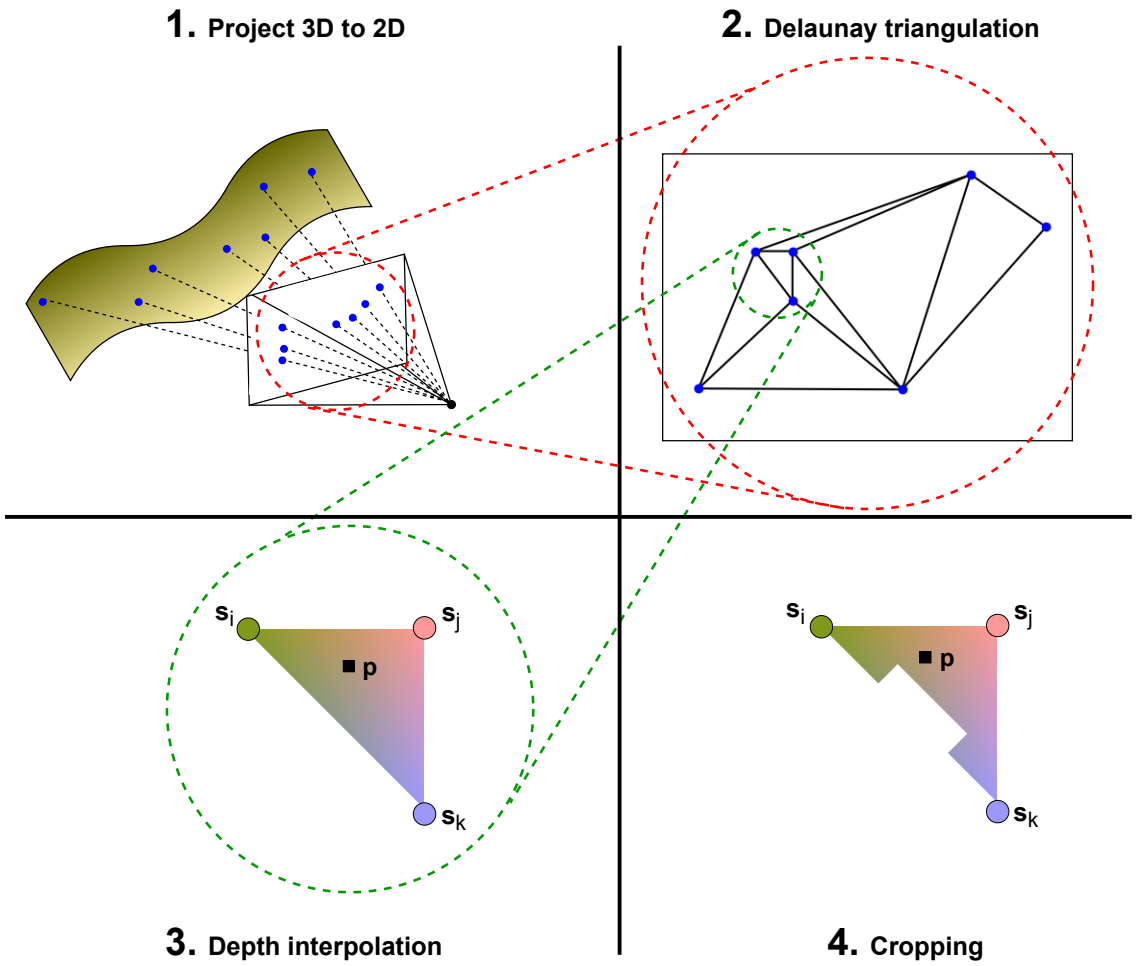


Figure 24: Depth interpolation workflow.

Ultimately, we decided to skip the keypoint densification step, because the results were qualitatively not much better, and the time complexity of the depth interpolation was severely impacted. Even with the bucketing system, we already had a decent spread of the image features and a somewhat dense sample of keypoints.

A Delaunay triangulation, as performed in step 2, is a triangulation in which no point lies inside the circumcicle of any triangle. For this step, we utilize the Delaunay library in SciPy (Virtanen et al., 2020).

For point 3, interpolation, we implement a quite simple solution: We iterate through each pixel in the image and find which triangle the pixel resides in, and in turn the vertices of this triangle. This process could be sped up by e.g. storing the index of the last triangle to check first for the next pixel. Then we interpolate the depth at the pixel based on the value of these vertices, using linear barycentric interpolation, as in (Ferrera, 2019):

If we have a point

$$\mathbf{p} = \begin{bmatrix} u_p & v_p \end{bmatrix}^\top \tag{5.1}$$

within a triangle with vertices

$$\mathbf{s}_1 = \begin{bmatrix} u_1 & v_1 \end{bmatrix}^\top \tag{5.2a}$$

$$\mathbf{s}_2 = \begin{bmatrix} u_2 & v_2 \end{bmatrix}^\top \tag{5.2b}$$

$$\mathbf{s}_3 = \begin{bmatrix} u_3 & v_3 \end{bmatrix}^\top \tag{5.2c}$$

we can calculate the position of $\mathbf{p}$ relative to the points $\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3$, by solving for the weights w_1, w_2, w_3 in the linear system of equations

$$u_p = w_1 u_1 + w_2 u_2 + w_3 u_3 \tag{5.3a}$$

$$v_p = w_1 v_1 + w_2 v_2 + w_3 v_3 \tag{5.3b}$$

$$w_1 + w_2 + w_3 = 1 \tag{5.3c}$$

After rearranging, the solution to the system can be found to be

$$w_1 = \frac{(v_2 - v_3)(u_p - u_3) + (u_3 - u_2)(v_p - v_3)}{(v_2 - v_3)(u_1 - u_3) + (u_3 - u_2)(v_1 - v_3)} \tag{5.4a}$$

$$w_2 = \frac{(v_3 - v_1)(u_p - u_3) + (u_1 - u_3)(v_p - v_3)}{(v_2 - v_3)(u_1 - u_3) + (u_3 - u_2)(v_1 - v_3)} \tag{5.4b}$$

$$w_3 = 1 - w_1 - w_2 \tag{5.4c}$$

If $\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3$ have corresponding depth values d_1, d_2, d_3, then the depth at $\mathbf{p}$, d_p, can

be interpolated as

$$d_p = w_1 d_1 + w_2 d_2 + w_3 d_3. \tag{5.5}$$

For point 4, we utilize the average distance from the pixel in question to the three vertices in the triangle to determine whether the pixel, p, is "too far" away from the points, s_i.

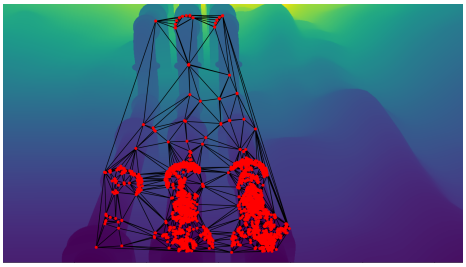
$$\texttt{distance} = \frac{1}{3} \sum_{i=1}^{3} |\mathbf{p} - \mathbf{s}_i| \tag{5.6}$$

The `threshold` could then be set to e.g. 25 pixels, and we can test whether `distance < threshold`.

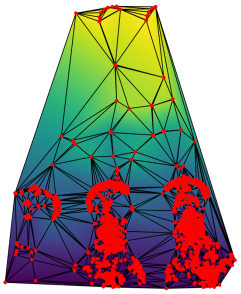
For point 5 we utilize a ROS package called `depth_image_proc` ('Ros Wiki depth_image_proc', n.d.) for converting the 3D depth map into a dense 3D point cloud. The reason for doing this is that Voxblox requires very specific data types on all its input. This means that one can not use e.g. arbitrary color representations or depth value data types. The `depth_image_proc` package makes these data type conversions easier to handle.

Figure 26 shows a sequence diagram of how the different components of the ROS environment are communicating. The green components, i.e. the content of the `depth_interpolation` node, is the self-developed component of the depth interpolation part. This node can either publish ground truth depth images, publish fully interpolated depth images, or publish cropped interpolated depth images (with a threshold set by the user). The `depth_interpolation` node can also be recognized as a green box in fig. 27. In this figure, we have not included details of the ROS communication. Here, the node takes output from the improved SLAM emulation, performs the desired depth interpolation and point cloud coloring, and sends the output to Voxblox.

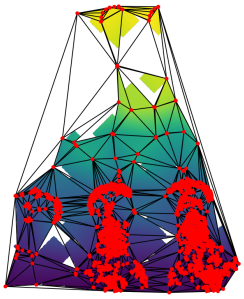
For practical purposes, we usually recorded the output in ROS and then played it later to Voxblox to test the reconstruction with different parameters. The timestamp of the transform and point cloud need to be synchronized before being published to Voxblox.



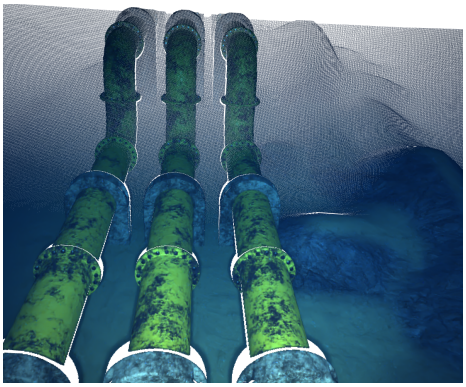
(a) Obtain set of detected SLAM points projected onto the image and their corresponding depth from the camera. Perform Delaunay triangulation on these points. Corresponds to 1 and 2 in the procedure, as illustrated in fig. 24.



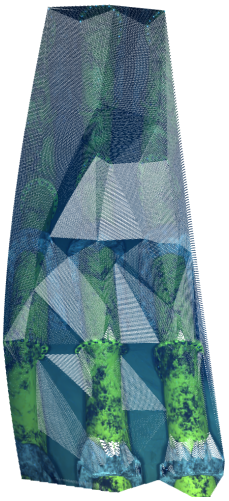
(b) Linearly interpolate the depth in the triangles obtained from (a). Corresponds to step 3 in the procedure.



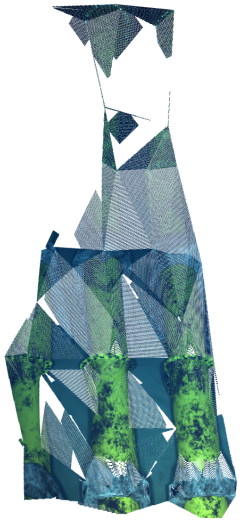
(c) Crop away interpolations that are too many pixels away from the detected points. Corresponds to step 4. in the procedure.



(d) Result from projecting depth map (a) into space, i.e. before doing any interpolations.



(e) Result from projecting depth map (b) into space.



(f) Result from projecting depth map (b) into space.

Figure 25: Ground truth depth map (a), interpolated depth maps (b), (c) , and their respective projections into space to obtain point clouds in (d), (e), and (f). The depth interpolation threshold used in (c) is set to 40 pixels in this example.

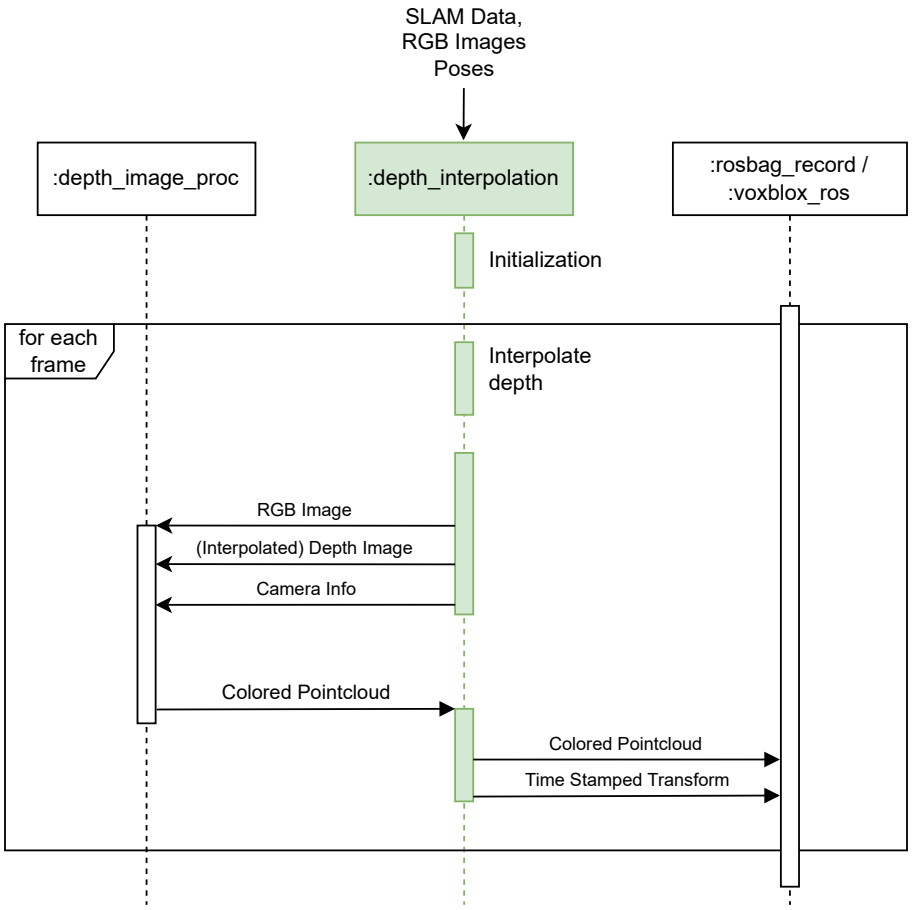


Figure 26: ROS communication chart. The depth interpolation module sends the interpolated depth map to the depth_image_proc package to obtain the corresponding colored point cloud on the correct data format. Upon receiving this point cloud, it sends it with a timestamped coordinate transform to Voxblox.

5.2 Surface Reconstruction System

The workflow of the entire system constructed in this thesis is shown in fig. 27. The green boxes are the methods presented in this thesis: The SLAM emulation method is presented in section 4, and the per-frame depth interpolation is presented in section 5.1.

The SLAM emulation keeps control of a 3D point cloud. In principle, both the pose and the position of the points could be adjusted, as if influenced by, e.g., a bundle adjustment procedure. The reconstruction system would not fail because the algorithm takes the position of the points and the pose *at a certain time* and projects the points onto the image plane for depth interpolation. This same pose is then given with the interpolated depth map to Voxblox, the third-party software presented in section 2.4. However, the previous reconstructed surface will not change with the adjusted 3D point cloud. Thus, if the adjustments are large, the reconstruction might become quite choppy and inaccurate.

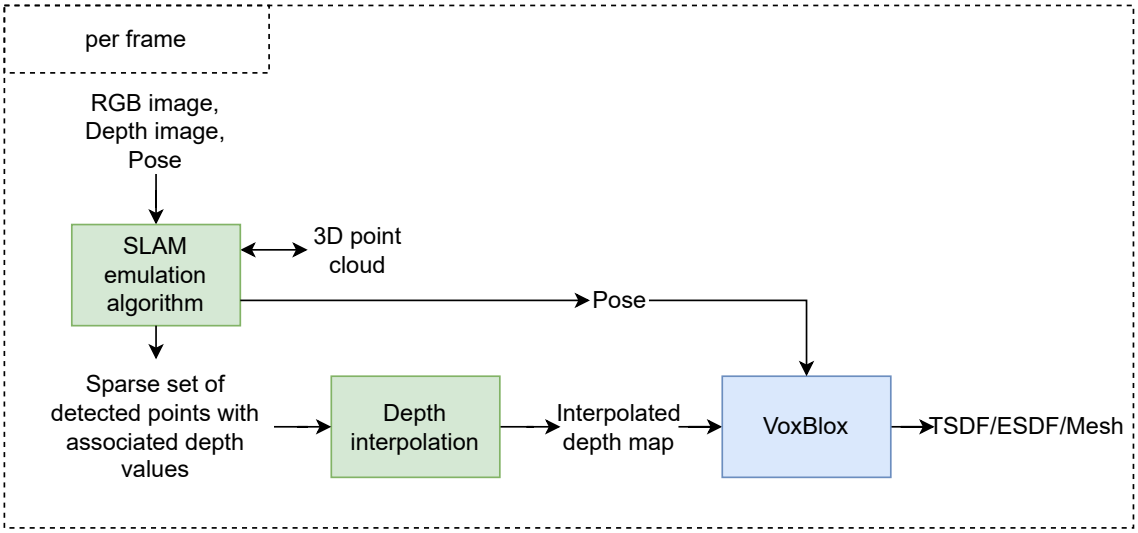


Figure 27: Workflow of the entire system. The green boxes are the components developed in this project, while the text with no boxes represent intermediary results. The depth_image_proc node shown in fig. 26 is not shown in this diagram, but can be viewed as a part of the (per-frame) depth interpolation box.

The actual surface reconstruction is performed in Voxblox, which takes per-frame depth map and poses as inputs and outputs both an ESDF (for robot planning) and a mesh (for human visualization). The ESDF results are somewhat difficult to study in a thesis because the entire ESDF is not easily visualized. Furthermore, for both the mesh and the ESDF, a quantitative study is difficult because (i) we do not actually have the ground truth surface in the simulation, and (ii) even if we had that ground truth surface, defining an error metric is quite difficult, for reasons discussed in section 6.

6 Results and Discussions

In this section of the thesis, we aim to test the components of the system both separately and in combination. First, in section 6.1, we investigate the SLAM emulation qualitatively and quantitatively. The main thing to test here is how well coverage we have of detected points in each image, which will e.g. be a big factor in how well the depth interpolation performs. In section 6.2, we investigate the depth interpolation quantitatively and give some interpretations of the results. Then, in section 6.3, we test all components together. The tracking algorithm forms the input for the depth interpolation, which in turn forms the input to Voxblox, the chosen surface reconstructor.

6.1 SLAM Emulation

Since we assume various ideal properties on the data, this section will mainly test how well the ORB detector works on the dataset, and it will give a basis for discussions in future subsections. We test for two main things: the number of points tracked in each frame and the coverage of the tracked points in the image, i.e. how large parts of the image where we detect features. Because of the bucketing system, the two will be highly correlated. This is because we typically only have one point per cell. For all tests in this subsection, we used a cell size for the bucketing system of 16×16 pixels and set the maximum number of detected ORB features in each image to 3000.

SLAM emulation results

Figure 28 shows the number of points tracked per frame, along with the percentage of cells that are occupied in the bucketing system. As we can see, the numbers vary a lot. Upon inspection, the number of tracked points is 0 for some frames. Since we project new points once we detect them, this means that 0 ORB features are detected for those frames.

Figure 29 shows the result of the tracking algorithm for some selected frames. Blue points are the 3D points in the model that are successfully tracked, i.e. they are projected onto the image and compared to the ground truth depth that we have available in the dataset, as described in section 4. The red points are the detected ORB features that are added to the model at the current frame. The green points are all detected ORB features that are *not* added to the model because another point in the model occupies its cell. The frames are not chosen at random but are rather chosen because they contain valuable information.

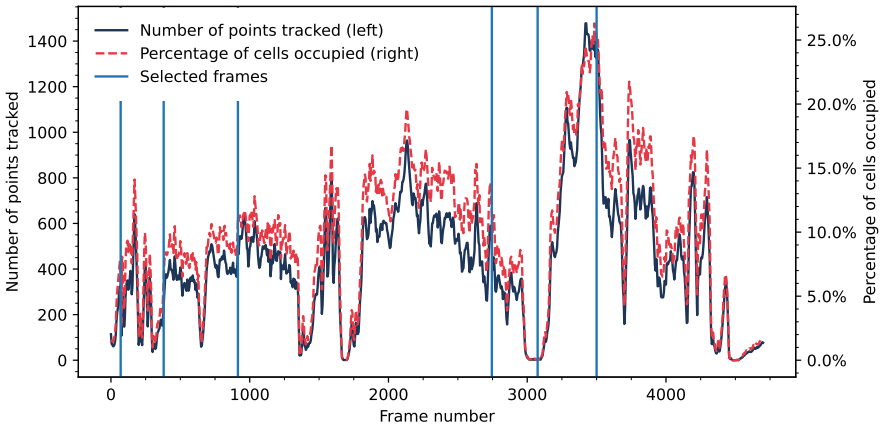
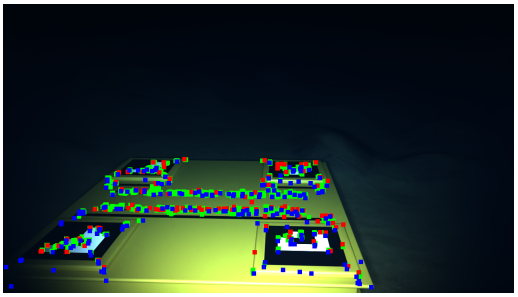
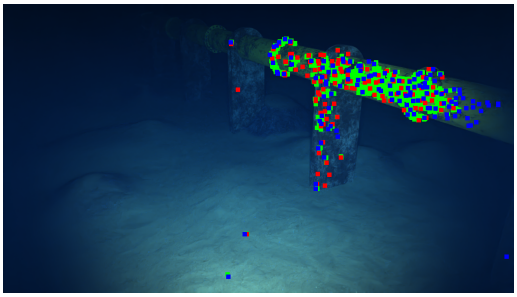


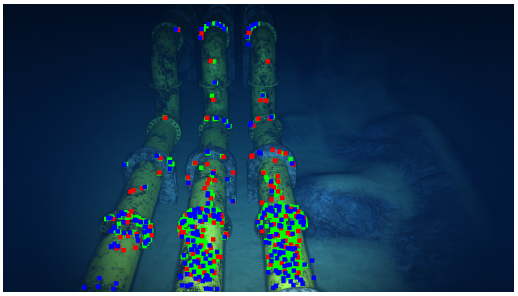
Figure 28: The dark blue line shows the number of points tracked at each frame. The red dotted line shows the number of cells in the image we have detected points, divided by the total number of cells. The vertical blue lines show the frames illustrated in fig. 29.



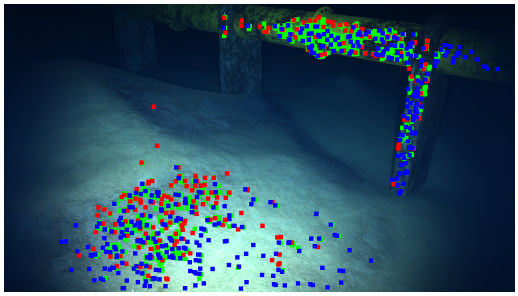
(a) Frame 70



(b) Frame 380

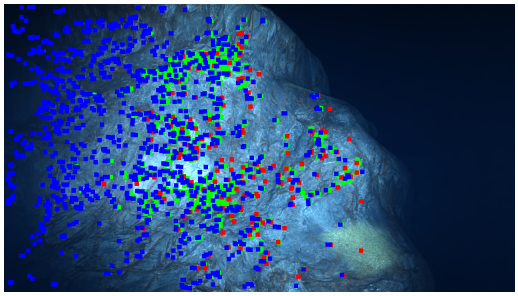


(c) Frame 915



(d) Frame 2745

(e) Frame 3075



(f) Frame 3500

Figure 29: Results of tracking algorithm for selected frames. Cell borders are not drawn. Blue points are model points projected onto the current frame. Red points are detected points added to the model at the current frame. Green points are detected points that are disregarded due to the bucketing system.

SLAM emulation discussion

In fig. 29a, there are no points detected in the smooth areas of the tagplate, but only where the color or structure changes. For fig. 29b, there are a lot of points detected along the pipe, but not many points detected outside the pipe. The same applies to fig. 29c. This is probably a result of the texture and lighting conditions. It is apparent that in darker areas, there are fewer points detected. See, for example, on the very right of fig. 29b. This area has previously been lit up by the robot but is now dark. This has resulted in points being added to the model at previous frames, indicated with blue points at the current frame, 380. This example also shows the arguably unrealistically good performance of the SLAM algorithm: All these blue points would perhaps not be properly tracked in a real setting. Because we only look for depth information when tracking points, the RGB images are actually irrelevant for the actual tracking. The gray-scale versions of the RGB images are, on the other hand, used for the ORB detection.

For fig. 29e, which depicts an empty sand ocean floor, there are almost no points detected at all. There is some light shining on the ocean floor dunes, but apparently, this is not enough for the ORB-detector. This might be an artifact of using a simulation. In a simulation, the environment is, in some way, more "perfect" than in reality, i.e. it has fewer visible artifacts and imperfections, which might make it harder to detect points. This would, of course, impair the quality of the SLAM algorithm. In a real scenario, a feature-based SLAM system based purely on an RGB camera would fail if it did not detect any features, which happened at some frames. However, since we have the ground truth available pose from the dataset, our system does not fail. In fig. 29f, where there are rocks with texture but no pipes, there are again many points detected. Here we also see quite a lot of points being tracked because the robot is quite close to the scene, and because there is nothing obstructing its view. Again, many of these points might not have been tracked in a real setting without access to the ground truth depth.

From fig. 29, it is apparent that the coverage of detected features over the entire image is, qualitatively, not satisfying. For some frames, the number of points tracked is so low that a regular feature-based VSLAM method will fail. Since the surface reconstruction is reliant on the pose from the SLAM, it will then also fail at this point. However, if the pose is available through the frames with loss of detected features, e.g. using an IMU for ego-motion estimation, the reconstruction will resume once we regain an image coverage above zero.

6.2 Depth Interpolation

For the depth interpolation, we mainly want to test three things: the interpolation *accuracy*, the *coverage* of the image, and the interpolation time. The coverage is defined as the number of pixels where we interpolated the depth, divided by the total number of pixels in the image, giving a percentage rate over how much of the image is interpolated. This number is largely influenced, naturally, by the threshold we set on the max pixel distance from a detected point we interpolate the depth, as illustrated in fig. 24 in section 5.1. For the accuracy, we test for the worst-case interpolation errors, as well as the mean and max errors for the image. The median was, upon inspection, found to be very similar to the mean, so we restrict ourselves to presenting the mean. The "error" in these tests is defined as the ground truth error minus the interpolated depth error and is calculated for each pixel in the depth image in which we interpolated the depth.

We structure the tests into four parts. Part (A) shows some of the worst interpolation errors when comparing with the ground truth depth map; Part (B) shows the mean and max interpolations errors for each frame; Part (C) tests the image coverage for each frame. Finally, part (D) tests the interpolation time.

For all tests in this section, we have chosen the parameters in table 1.

Cell size	16×16
Interpolation threshold	25 px

Table 1: Depth interpolation parameters

(A) Worst-case errors – Results

In fig. 30 we have plotted a scatter plot of the depth interpolation result from three frames. The three frames have in common that they have a worst-case depth interpolation error of at least 6 meters for one or more pixels.

Figure 31 shows the depth interpolation for the frames plotted in fig. 30, put on top of the ground truth depth map. The red dots indicate where the errors are largest.

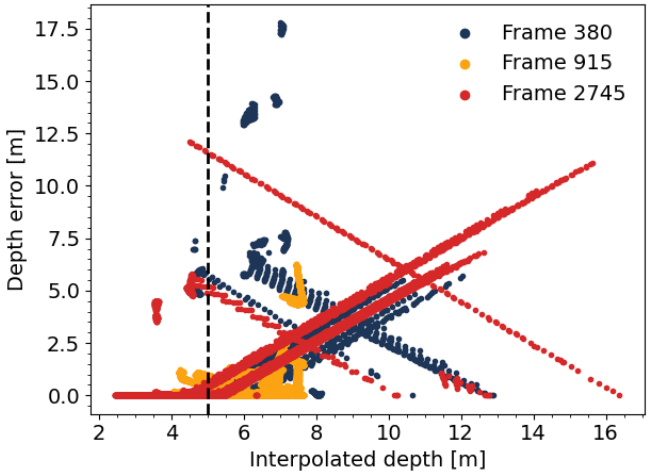


Figure 30: Depth interpolation errors, plotted against the interpolated depth at their respective pixels, for three selected frames with poor maximum depth interpolation error. The black vertical line indicates a suggested threshold on what depths to not perform interpolations in order to exclude the worst errors.

(a) Frame 380 (b) Frame 915 (c) Frame 2745

Figure 31: Interpolated depths superimposed on ground truth depths for the frames used to produce fig. 30. The red dots indicate the areas where the depth interpolation gives the worst results.

(A) Worst-case errors – Discussion

Intuitively, the worst-case interpolation errors are the most important errors to exclude. As indicated in fig. 30, many of the observed worst-case errors can be excluded by setting a maximum limit to which depths we interpolate. In fig. 30, this limit is indicated as 5 meters, and is suggested simply on an empirical basis.

Turning the attention to fig. 31, we pose the question of whether such a limit makes sense logically and not only empirically. Notably, most of the worst-case errors are located very near, or at, the edge of the pipes. This is because we triangulate points on the pipes such that parts of the resulting triangle cover the area behind the pipes. From these images, the interpolation errors do not actually appear that terrible: It is not crucial that the pipes are constructed *a little bit* wider than they are supposed to. Also, importantly, there is nothing logically hindering these

errors from appearing at close ranges, so the suggested truncation line is somewhat empirically based.

However, setting such a limit might make sense from another standpoint, which is illustrated in fig. 32: Let us say we interpolate the depth between two points in the image that are 20 pixels apart. Now, let us say that if the points are 5 meters away from the camera, that equals a distance of 10cm between the points. If the points were instead 25 meters away from the camera, this would imply a distance of 50cm between the points. Surely, interpolating the depth of two points 50cm apart is more uncertain than of two points 10cm apart. Thus a larger area will be given an inaccurate surface reconstruction, which might give rise to artifacts.

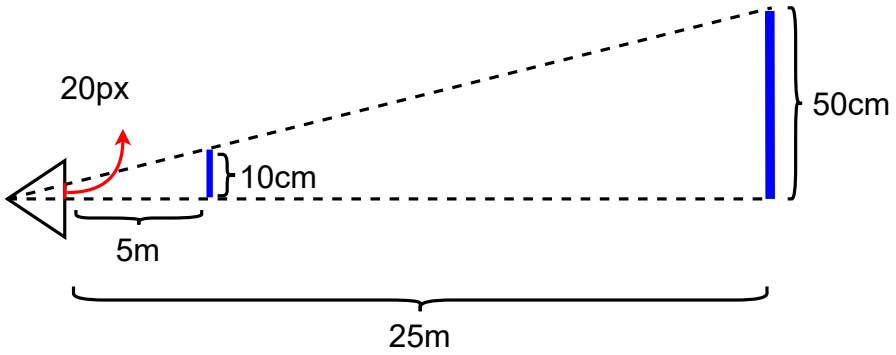


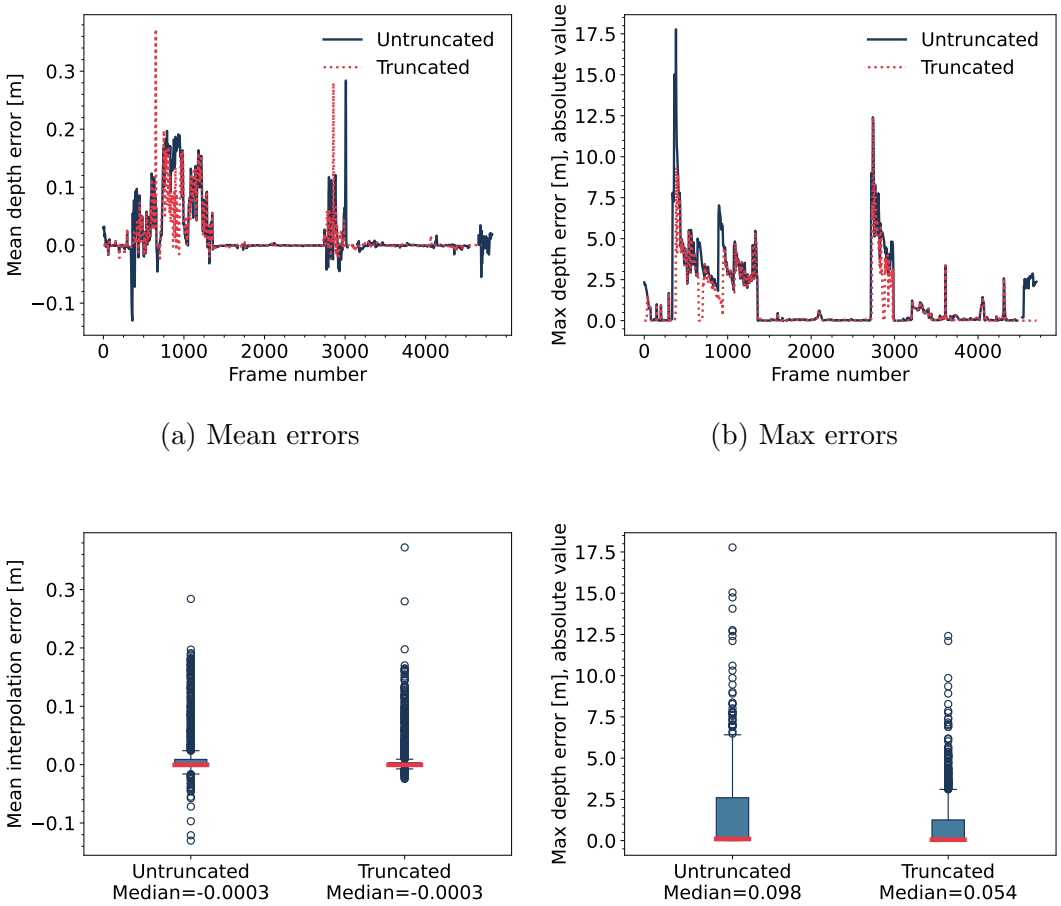
Figure 32: This figure illustrates why a longer max ray length might yield worse surface reconstruction when interpolating depth images. 20 pixels in the image is a smaller area from 5 meters distance than from 25 meters.

For this reason, we conduct tests both with and without this limit. Henceforth, this limit will be referred to *truncating* the interpolated depth.

(B) Error statistics – Results

Next, we wanted to test the reconstruction accuracy across frames, in order to get a sense of the general size of the error as well as the consistency of the error magnitude. Figure 33 shows the general magnitude of the errors. In fig. 33a, we have plotted the average depth interpolation error for each frame, whereas in fig. 33b, we plot the maximum interpolation error in each frame. Both plots include two graphs. The red, dashed line shows the error when we truncate the depth interpolation at 5 meters, as described in the previous paragraph and illustrated in fig. 30 as the vertical line. In figs. 33c and 33d we show boxplots of the mean and max interpolation error, where one datapoint in the boxplot corresponds to the mean/max error for one frame.

It is apparent that we have some spikes in fig. 33a: the leftmost red spike, which obtains a value over 0.3m, corresponds to frame 655, which we devote some further study. Figure 34a shows the interpolated depth image over the ground



(a) Mean errors

(b) Max errors

(c) Mean errors

(d) Max errors

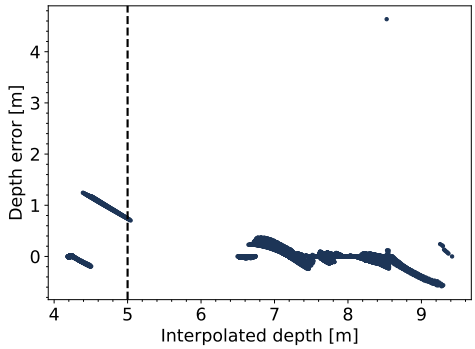
Figure 33: Depth interpolation error statistics. (a) shows the mean error at each frame, and (b) shows the max absolute error at each frame. Both figures show both the statistics for the untruncated and truncated depth interpolations, as per the line in fig. 30. (c) and (d) show boxplots of the values in (a) and (b).

truth depth image for frame 655. We have split the interpolation into two groups, illustrated by an orange and a red rectangle. Figure 34b shows a scatter plot of the interpolation errors, plotted against the interpolated depths. By inspection we see that the interpolated depths below 6 meters correspond to the red rectangle, and the rest correspond to the orange rectangle. The vertical line again indicates the truncation line.

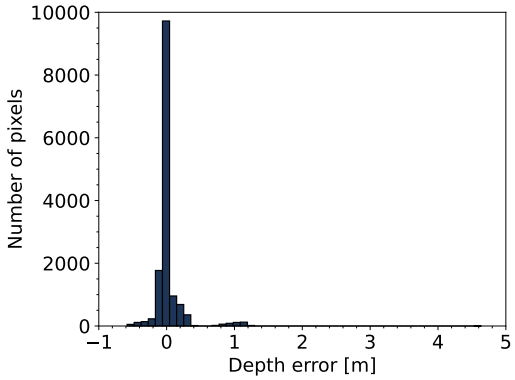
Figure 34c shows the histogram of the interpolation error for each pixel, without truncating any depths. Figure 34d shows the same but after truncating away any depths over 5 meters.

To compare the results for the truncated and untruncated interpolations in fig. 33, we first present the max and median interpolated depth for each frame, shown in fig. 35.

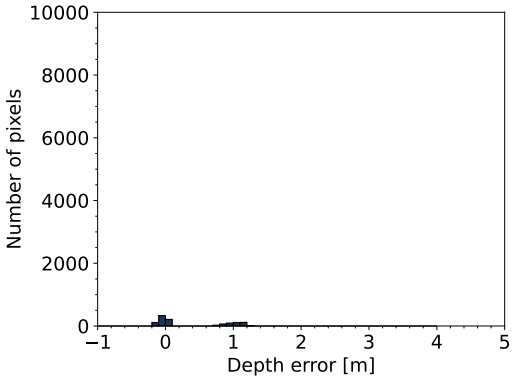
(a) Interpolated and ground truth depth maps for frame 655, divided into two groups, indicated by the rectangles.



(b) Depth interpolation errors, plotted against the interpolated depth at their respective pixels.



(c) Untruncated depth interpolation error histogram.



(d) Depth interpolation error histogram, truncated at 5 meters. The vertical axis has the same limits as in (c).

Figure 34: Interpolated depth maps and error statistics at frame 655.

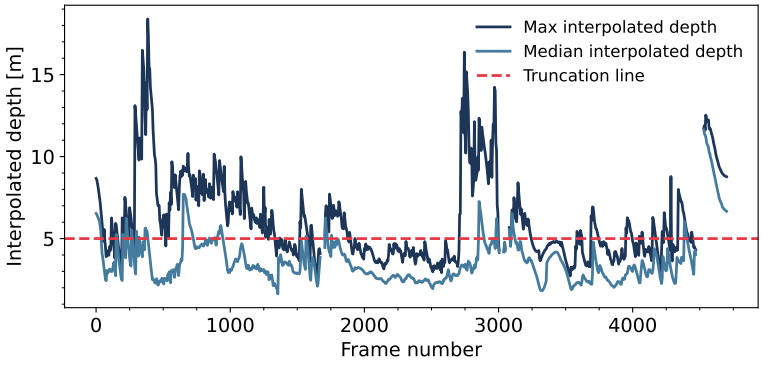


Figure 35: Max and median interpolated depth per frame, in meters.

(B) Error statistics – Discussion

We see from fig. 33a that the mean error typically stays between [-0.1, 0.2] meters, with three notable exceptions, one of which will be discussed later in relation to fig. 34.

First, we note that most of the mean errors are positive. This can be explained by the fact that we take the ground truth minus the interpolated depth at each pixel. The areas where we interpolate the depth will typically span areas where the actual depth is larger. This effect can be seen quite clearly in fig. 31b, where the interpolated areas span across the pipes, way closer than the background.

Figure 33b shows that the maximum depth interpolation error per image typically stays between [0.0, 7.5], which is confirmed in the accompanying boxplot in fig. 33d. Comparing figs. 33c and 33d, we see that it is mainly the *absolute worst* errors that are helped by the truncation.

One explanation for this, motivated by the fact that the means are quite similar for truncated and untruncated depth images, is that the interpolation accuracy is not much worse from a distance. Another explanation is that most of the depth that is interpolated is below 5 meters anyways. This is partly supported by fig. 35, in which the median interpolated depth is shown to usually stay below 5 meters. This makes sense because the (simulated) light from the robot mainly illuminates the area close to the camera.

As discussed in part (A) "Worst-case errors", the absolute worst-case errors were *not* in fact crucial to exclude, because they typically resided close to the reconstructed pipes. Actually, a truncation might exclude quite valuable and correct information. To illustrate this, we turn to the first artifact in fig. 33a, where we have a spike in the mean error *when we truncate the depth*.

The new histogram in fig. 34d shows that the majority of points have been truncated away. Most of these points had small errors, and thus valuable depth information has been lost. To explain the spike in fig. 33a, we note that the histogram from the truncated depth map error in fig. 34d contains two groups of errors: one with an error of about 0 meters, and one with an error of about 1 meter. The average of these groups gives an error of ~ 0.38 meters, which explains the spike, which we thus do not devote any further study.

As a takeaway from this study, we note that although a truncation on the depth interpolation can be supported logically and empirically, it might exclude too much valuable information to be justified.

(C) Image coverage – Results

The next item to study was the coverage of the depth interpolation over the entire image for the entire sequence, as defined in the introduction to this subchapter. Figure 36 shows the coverage as the blue line plotted against the number of points tracked as the red dashed line. On the horizontal axis is the frame number.

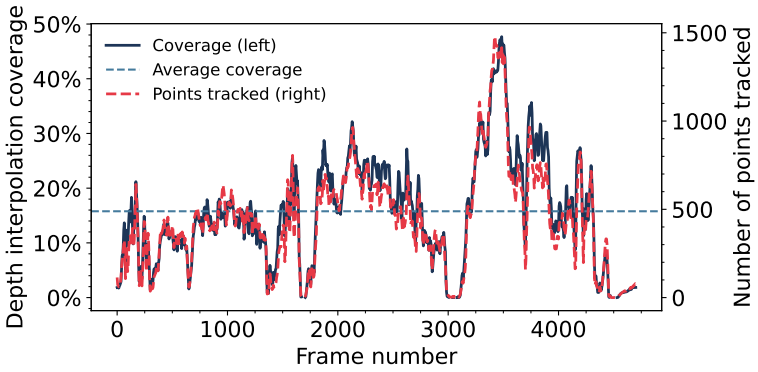


Figure 36: Depth interpolation coverage as a share of the total pixels in the image (blue line). The red line shows the average over the entire sequence shown in the figure.

The average coverage over the sequence is 15.8% and is plotted as the dashed blue line in fig. 36. The coverage takes values in the range 0% to 50%.

(C) Image coverage – Discussion

As expected, coverage and the number of points tracked are highly correlated. This is because the depth interpolation is based directly on the tracked points. Because of the bucketing system, the number of points tracked also correlates highly with the percentage of cells occupied, as previously shown in fig. 28. The number of cells occupied indicates how large portions of the image we have sparse depth samples. For some sections, we have coverage of up to 50%. This happens around frame 3500, which was shown, along with its tracking results, in fig. 29f.

For many frames, we have quite low coverage, dropping as low as zero for several frames, such as frames 1675 and 3040. In these parts of the sequence, we will naturally not reconstruct any parts of the surface. For reasons discussed in section 6.1, the reconstruction may fail at these points unless we can estimate the pose through the frames where few features are detected.

(D) Interpolation time – Result

The interpolation time per frame is shown in fig. 37 in blue. The red dotted line shows the number of points tracked per frame.

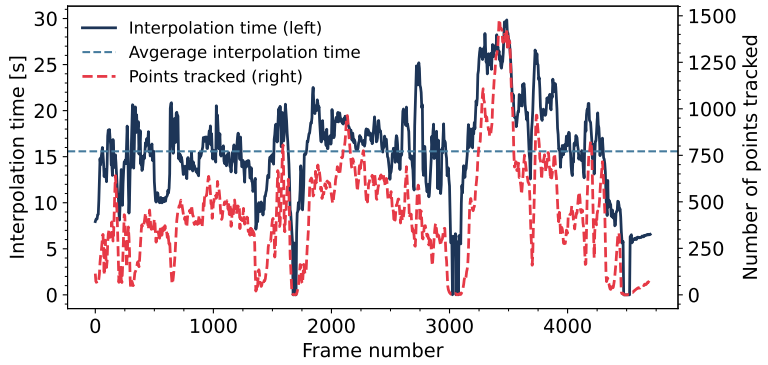


Figure 37: Interpolation time per frame over the entire sequence, plotted against the number of points tracked per frame..

(D) Interpolation time – Discussion

We see that the interpolation time is quite correlated with the number of points tracked in the image. With our implementation, this makes sense because the more points we track, the more points must be triangulated. This, in turn, is because for each pixel, we look up which triangle it resides in. A major disadvantage of this naive approach is the high computational complexity, manifesting as a high interpolation time, averaging at about 16 seconds, as shown in fig. 37. Because of this, we cannot remotely say that the implemented version of the depth interpolation works in real-time. This forms a basis for future work.

6.3 Surface Reconstruction

In this section, we aim to test the composition of the entire system, i.e. the three parts: SLAM emulation, depth interpolation, and Voxblox surface reconstruction, in composition. The two first were tested separately in in sections 6.1 and 6.2, and for a separate test of Voxblox we refer to (Oleynikova et al., 2017). For emphasis, we repeat the composition of the system: the workflow is illustrated in fig. 27, and we now test the entire system using RGB and depth images and poses as input, and obtain the outputs: TSDF, ESDF and mesh.

First, we argue why a quantitative study of the reconstruction results is difficult and why we have chosen to perform a qualitative study instead. In the consecutive subsection, we perform various tests on different subsets of the VAROS datasets. The discussions are included intermittently. The reason for this is clarity for the reader: A separate discussion section would include a lot of back-and-forth between the sections, possibly leading to unnecessary effort and confusion.

The Issue with Quantitative Reconstruction Metrics

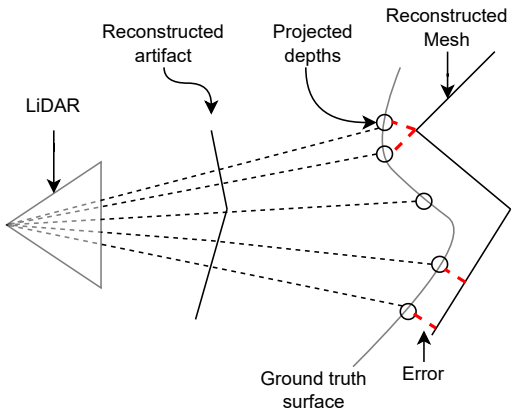
Ideally, we would like a quantitative study of the surface reconstruction, relating to some form of geometric accuracy, as well as time and space complexity. The information relating to time and space complexity of the reconstruction is not trivial to obtain without explicit information from the surface reconstructor, which is not very verbose in this case. It is difficult to estimate because we test the system with ROS and give the input to Voxblox at certain time intervals. However, we do not know how much time Voxblox actually uses for the surface reconstruction and how much time it is idle, waiting for more data. So, for information regarding the reconstruction time, we refer to the results obtained in (Oleynikova et al., 2017).

Furthermore, a quantitative study of the reconstruction *accuracy* is difficult as it is challenging to consistently measure the reconstruction error. The first point is that the VAROS dataset does not include the ground truth surfaces of the simulated scene. This may be overcome by defining error metrics similar to (Romanoni & Matteucci, 2015) or (Romanoni & Matteucci, 2018). These two error metrics are both based on LiDAR measurements, which are comparable to our ground truth depth maps. The two error metrics are, in principle, quite similar but will in some instances yield significantly differing results.

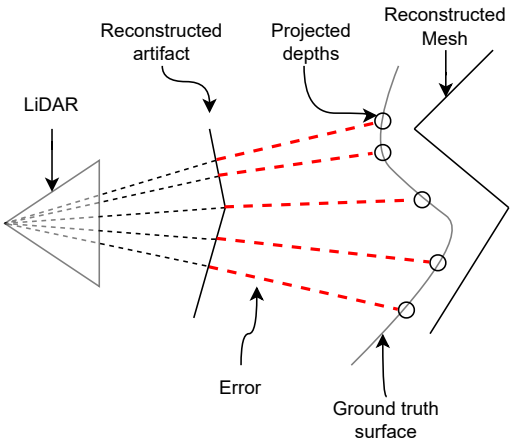
(Romanoni & Matteucci, 2015) defines the reconstruction error as "the average of the distances between each Velodyne point and the nearest mesh triangle". Velodyne is a type of LiDAR. The way we interpret this is that, for each point in a LiDAR scan, the point is projected into space using the pose of the scanner. Then, after the point is projected, one takes the Euclidean distance from the point to the

nearest reconstructed surface. This is illustrated in fig. 38a. Here the LiDAR points are projected through an artifact in the reconstruction (i.e. a mesh that is reconstructed erroneously and should not be there), and then measured to the closest reconstructed mesh. In this case, the error will be quite small.

(Romanoni & Matteucci, 2018) defines the reconstruction error by "comparing the depth images of the reconstruction rendered in each frame, against the distance of the Velodyne points projected on the same image". Our interpretation of this is illustrated in fig. 38b. The reconstructed surface model is projected onto the sensor plane. In the example in the figure, it is the *artifact* that is projected onto the sensor plane and compared to the depth at the corresponding pixels from the LiDAR scans. An alternative way of looking at this error metric is that each image point is traced to the first surface it hits, and then the error is calculated as how far this is from the actual depth measured by the LiDAR. In this case, the error will be quite large.



(a) Error metric in (Romanoni & Matteucci, 2015).



(b) Error metric in (Romanoni & Matteucci, 2018).

Figure 38: Illustration of two selected error metrics. In (a) the LiDAR points are projected onto space and then measured against the reconstructed surface. In (b), the reconstructed surface is projected onto the sensor frame and compared against the measure depth.

The previous example illustrates the issue of defining an exact and consistent error metric for the reconstruction accuracy. In addition, it is not trivial to implement even the described error metrics, as we would have to study the representation and data format of the mesh, and how to project it into the simulated camera plane. For these reasons, we chose to perform qualitative studies of the accuracy.

About the Experiments

The qualitative reconstruction studies are mainly performed to examine how different parameters in the reconstruction affect the results. The tests are performed on the sequence released with the VAROS publication (Zwilgmeyer et al., 2021) as of 15 April 2022. We have split the dataset into three sets. The subsets of the data were chosen to show some of the complex and varied results. Testset 1 is recorded as the robot's simulated trajectory moves over three pipes running along the ocean floor, testset 2 is from a trajectory over a tagplate, and testset 3 is the entire published sequence. Each testset is treated separately in its own subchapter.

For testset 1, we only show the reconstructed mesh output from Voxblox, as this is most easily visualizable. For testsets 2 and 3, we also show the resulting ESDFs.

We test the reconstructions by adjusting the different reconstruction parameters involved in the reconstruction. There are six parameters that we adjust to test how they affect the results:

1. **Depth map method**. This parameter determines what kind of depth information we utilize in the reconstruction and can take four values: (i) Ground Truth (GT); (ii) Interpolated, Uncropped (Int); (iii) Interpolated, Cropped (Int-C); and (iv) Tracked Points Only (Points). When using ground truth for reconstruction, we give the entire ground truth depth map to Voxblox for reconstruction. This data is readily available in the VAROS dataset. Since we also utilize the ground truth pose (also given from the dataset) in the reconstruction, this constitutes the best possible reconstruction we can obtain from Voxblox on the dataset. The second possible parameter value, "Int", means the we interpolate the depth in the triangles from the triangulation of the poitns obtained from SLAM, and we do *not* crop pixels far from the detected SLAM points. Conceptually, this will give similar results as fig. 25b. The parameter "Int-C", is the same as "Int", except that we *do* crop pixels far away from detected points. This is conceptually the same as fig. 25c. Lastly, the "Points" parameter means that we only utilize the depth of the SLAM points in the reconstruction and not any ground truth or interpolated depth. This is the same as only using the red points in fig. 25a.

2. **Keyframe interval**. This parameter relates to how many frames in the simulated video we skip before including a frame in the reconstruction. A keyframe interval of k utilizes frames $[\ldots, n-k, n, n+k, \ldots]$ in the reconstruction, where n is an arbitrary keyframe number.

3. **Voxel size**. This parameter relates to the sampling size of the SDF, as il-

lustrated in fig. 3. The larger this parameter is, the more rays traced will be merged together into a single measurement, and the fewer point samples of the SDF we get. In other words, we get a more smoothed-out mesh. As the voxel size approaches zero in the limit, the resulting SDF and mesh will approach the point cloud obtained by projecting the pixel-wise depth into 3D at each frame. For intuition, we also mention that as the voxel size approaches infinity in the limit, the resulting TSDF will only have one sampled value, and the entire field will be enclosed in this single-valued voxel.

4. **Depth interpolation pixel threshold**. This parameter relates to how far away from the detected SLAM points we interpolate the depth, i.e. what we call cropping. This distance is calculated as in eq. (5.6). We do not interpolate the depth at the pixels where `distance > threshold`. Looking at fig. 25c, we can see an example where the depth interpolation is cropped when the distance to the SLAM points (red) is too big.

5. **Cell size**. The cell size refers to how big the cells are in the bucketing system of the SLAM emulation method, as outlined in section 4.2, and illustrated e.g. as the blue lines in fig. 23. The cell size influences how dense the SLAM point cloud is, and thus how much information we enter into the depth interpolation: a larger cell size implies fewer points and thus less information. It should therefore be expected a poorer result with a larger cell size.

6. **Max ray length**. The max ray length refers to which depth values should be projected and not: the parameter sets a threshold on the depth values of the points we want to project into space. If the interpolated depth in the image is larger than the max ray length, it is not projected into space. This test is inspired by the results in section 6.2, where a max ray length of 5 eliminated some of the worst depth interpolation results, as illustrated in fig. 30.

For the first testset, we test variations in all these six parameters. For testsets 2 and 3, we only test a subset of the parameters.

6.3.1 Testset 1: Pipes

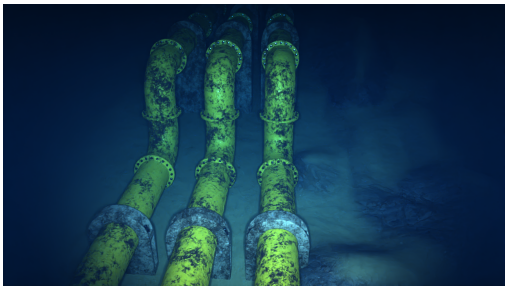
The first test used frames 930 through 1020 of the VAROS dataset, which we call "Testset 1: Pipes". The video is simulated as the robot moves along three yellow pipes. Four sample frames from this sequence is shown in fig. 39.

The tests of the different parameters are summarized in table 2. The leftmost column indicates the ID and name of each test. The table shows the values of the parameters tested in the different tests. First, we define a *base case*, which is a specific set of parameters and the resulting model. In all other test on this testset 1, we only change one parameter from the base case per test.

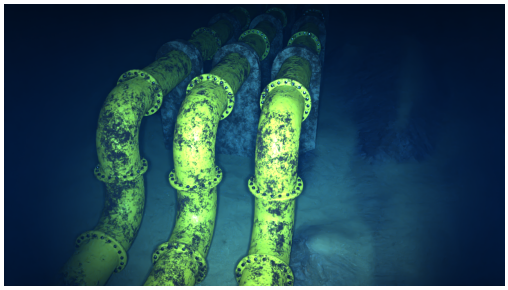
Test	Depth Map Method	KF Interval	Voxel Size [m]	Depth Threshold [px]	Cell Size [px]	Max ray length [m]
Base case	Int-C	5	0.05	25	16×16	25
Test 1. Depth map method	**GT, Int, Int-C, Points**	5	0.05	25	16×16	25
Test 2. Keyframe interval	Int-C	**1, 5, 15, 30**	0.05	25	16×16	25
Test 3. Voxel size	Int-C	5	**0.02, 0.05, 0.10, 0.20**	25	16×16	25
Test 4. Depth interpolation threshold	Int-C	5	0.05	**15, 20, 25, 30**	16×16	25
Test 5. Cell size	Int-C	5	0.05	25	**8×8, 16×16, 40×40, 80×80**	25
Test 6. Max ray length	Int-C	5	0.05	25	16×16	**5, 10, 15, 25**

Table 2: Testset 1 parameters. The leftmost column defines which parameter we change in that test. For the depth map method we define GT: Ground Truth depth map; Int: Interpolated depth map with no threshold; Int-C: Interpolated depth map with threshold, i.e. cropped; and Points: only (sparse) points with known depth. The voxel size refers to the side lengths of the voxel cube.

The *base case result*, i.e. the resulting mesh from using the base case parameters in table 2, is shown in fig. 39e. The discussion for this base case reconstruction is reserved for when it is compared to the other parameters.

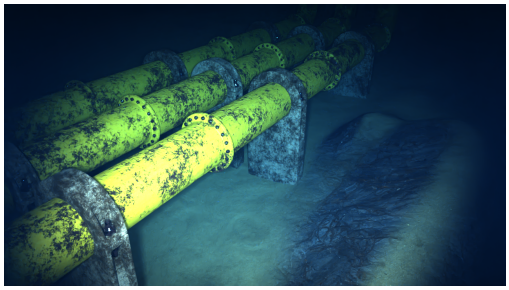


(a) Frame 930

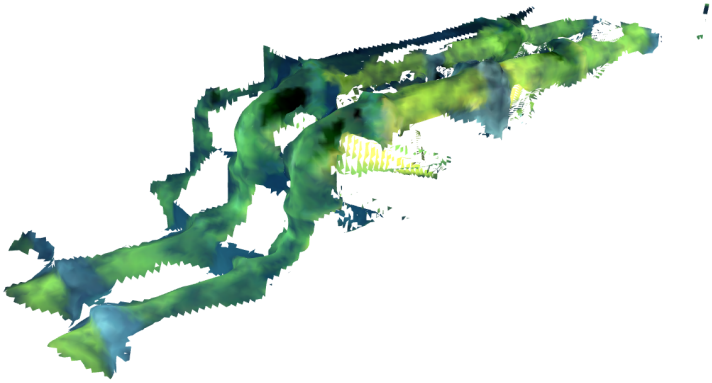


(b) Frame 960

(c) Frame 990



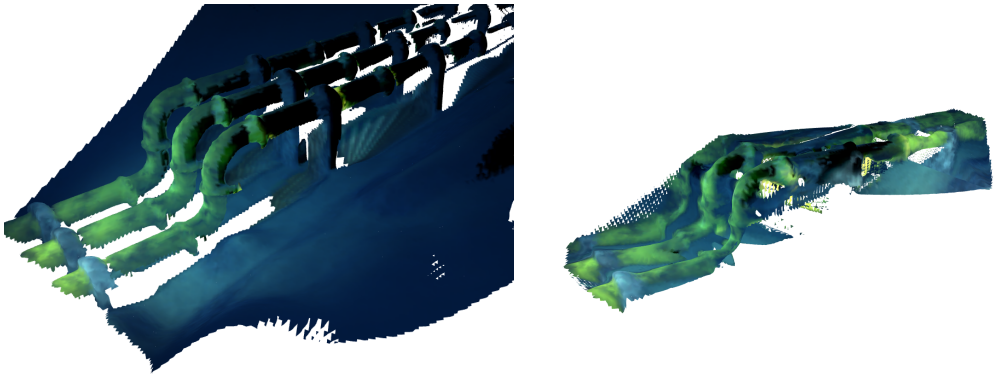
(d) Frame 1020



(e) Resulting mesh using parameters in the "base case" row of table 2.

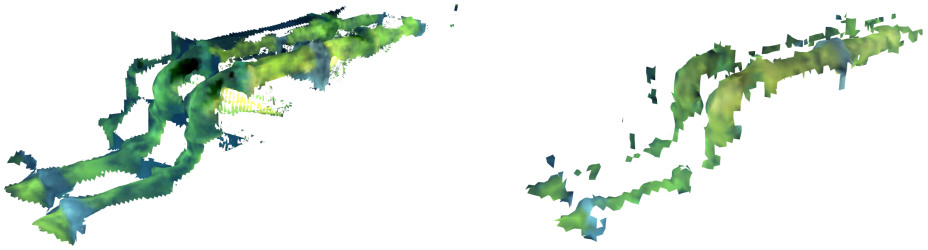
Figure 39: Test 1 images (a)-(d), and "base case" mesh result (e).

Test 1. Depth map method – Results. First, we wanted to test the effect of the choice of the depth map method on the results. We tested four different depth maps: the reconstruction in fig. 40a used the ground truth pixel-wise depth map over the entire image, in fig. 40b the entire interpolated per-frame depth map, in fig. 40c the interpolated per-frame depth map with a threshold as to not interpolate depth far away from SLAM points, as presented in section 5.1, and in fig. 40d only using the points from the SLAM emulation, giving a very sparse depth map. The results are shown in fig. 40. The parameters used in the reconstructions are given in table 2 in row 1: "Depth map method", with the only parameter changing across the reconstructions being the depth map method.



(a) Ground truth per-frame depth

(b) Estimated per-frame depth without threshold



(c) Estimated per-frame depth with threshold (base case)

(d) Only points from SLAM, increased voxel size

Figure 40: Results from changing the "depth map method" parameter. The four images are snapshots of the resulting 3D mesh model. These four snapshots are taken from the same pose. The description of each subfigure refers to what depth map was used to build the model.

Test depth map method – Discussion. From fig. 40, there are a couple of things we can say about the depth map's influence on the reconstruction. In fig. 40a, we see the accuracy of the reconstruction method when using the ground truth depth map for each frame. This result may be comparable to having a very precise LiDAR scanner, which can give pixel-wise depth measurements, as provided by the ground truth depth. This accuracy is starting to become available underwater (3D at Depth, 2022), but for cost or space constraints, it may not be feasible to install on an AUV yet.

Figure 40b shows the result from the depth interpolation *without* cropping away interpolated depth too far away from the detected SLAM points. The problem with this approach is that some of the interpolated depths become very inaccurate.

In the dataset with the pipes, the depth interpolation struggles especially in the space between the pipes, because few points are detected there. From fig. 40b, it appears that the gap is just filled in on the level of the pipes, which is of course wrong: the ground is well below the pipes. The color is taken from the RGB-image, and that is why the color is still blue. Cropping away parts of the depth map partially solves this problem, as can be seen in fig. 40c, in that not as much erroneous surface is reconstructed between the pipes. From that figure, however, it appears like Voxblox erroneously tries to connect the pipes to the ground. The ground is not reconstructed, so the artifact looks like a deformation of the pipes. For clarity, the artifact is marked with a red rectangle in fig. 41. We name it the"pipe-to-ground" artifact, because it has likely arisen because the depth interpolation has triangulated points on the pipes to points on the ground and interpolated depths accordingly.

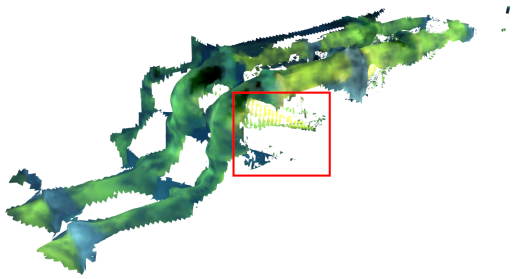


Figure 41: The "pipe-to-ground" artifact in the base case reconstruction.

Interpolating the depth as we do, i.e. triangulating projected SLAM points in 2D, will be very imprecise when dealing with e.g. sharp edges, such as in this subset of the data. Figure 42 illustrates the problem. The sharp edges will not be

constructed properly; rather, the depths are interpolated linearly between detected points.

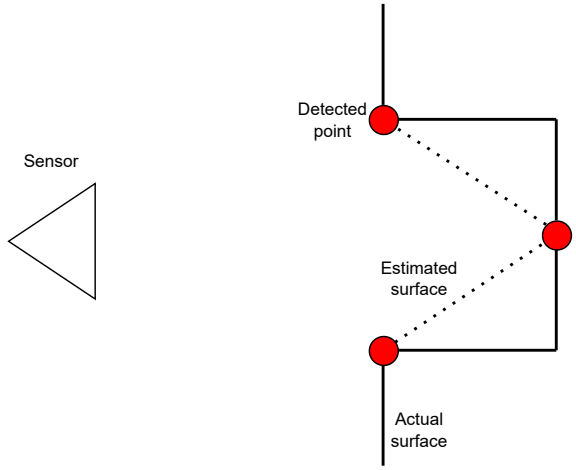
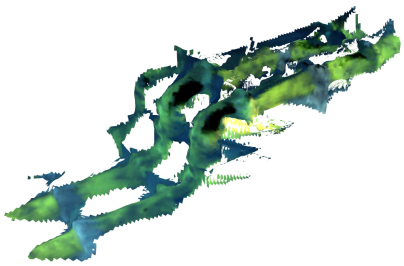


Figure 42: Possible depth interpolation error around corners. Red points are detected points with known depth, the solid line is the actual surface and the dashed line is the interpolated depth and surface.

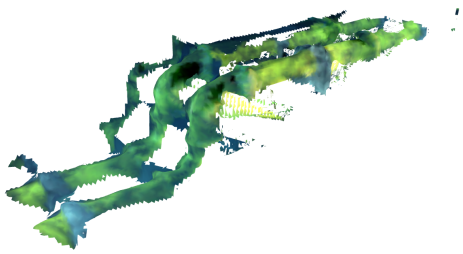
Figure 40d shows the results of not having any depth map at all, i.e. we only use the points from slam to reconstruct the surface. This is done similarly to the other depth map, except that the depth map is very sparse in this case. The voxel size is increased in order for the mesh to cover more area. One can argue that this method gives less, but more accurate, surface patches, depending on the voxel size: If we increase the voxel size more and more in order to get a larger mesh, the surface approximation will quickly deteriorate. In that case, it is in principle the marching cubes algorithm used to extract the mesh that does the approximation rather than the depth interpolation.

Given that the result from the using the ground truth depth map looks so accurate, one takeaway from this test is that one may want to exchange the depth interpolation for a more accurate estimation technique. This is made possible by the modular nature of the system. For example, (Ochs et al., 2017) uses Principal Component Analysis to estimate a depth map from a discrete set of samples, as obtained from the SLAM system. Another possibility is the approach presented by (Knutsson & Westin, 1993) and modified by (Pham et al., 2006), which uses normalized convolution to interpolate data from incomplete samples, or one can use newer methods based on neural networks, such as (Eldesokey et al., 2020).

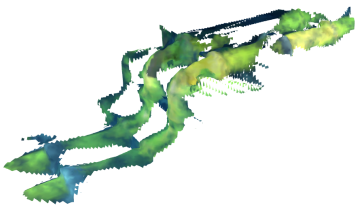
Test 2. Keyframe interval – Results. Second, we wanted to test the effect of the keyframe interval on the reconstruction quality. The *keyframes* are the frames that are involved in the reconstruction, i.e. all other frames than the keyframes are completely disregarded from the SLAM and the reconstruction. The base case has five frames between each keyframe and is shown in fig. 43b. The simulated camera frames per second is 10, so a keyframe interval of five would yield two keyframes every second. We also tried a keyframe interval of 1, meaning that we include all frames from the camera in the reconstruction process. That result is shown in fig. 43a. Furthermore, we tried a keyframe interval of 30 frames, reconstructing the scene with only frames [930, 960, 990, 1020], and is shown in fig. 43c. Lastly, we reconstructed using only frames [930, 975, 1020], a keyframe interval of 45, which is shown in fig. 43d.



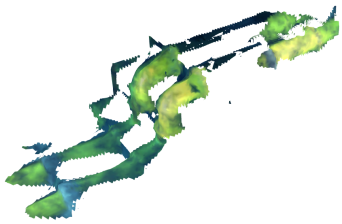
(a) Keyframe interval = 1 frame



(b) Keyframe interval = 5 frames (base case)



(c) Keyframe interval = 30 frames



(d) Keyframe interval = 45 frames

Figure 43: Results from changing the "keyframe interval" parameter.

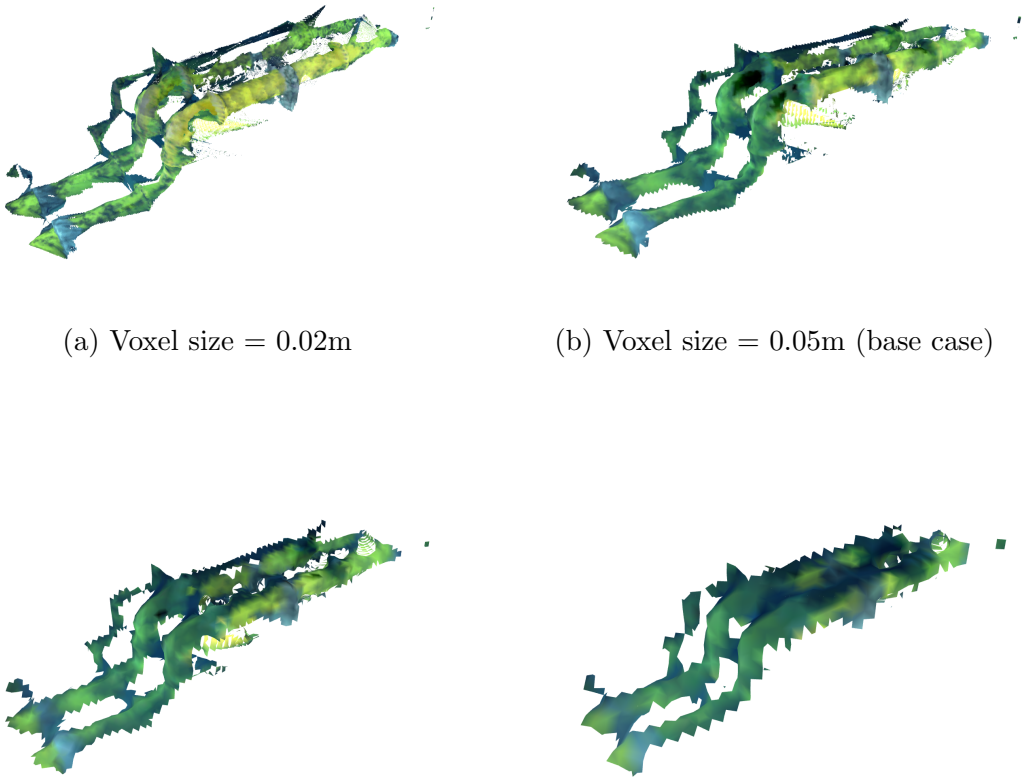
Test keyframe interval – Discussion. Changing the keyframe interval also yielded some interesting results: From fig. 43, it appears that a shorter keyframe interval yields a more "noisy" mesh. This is probably because, given more keyframes, more points are detected, and more (uncertain) depth is interpolated. For example, look at the difference between fig. 43a and fig. 43d. In the former, with a keyframe interval of 1, there are more artifacts. Ideally, with a lot of information, the isosurface from the TSDF should converge to the actual surface, but even with a keyframe interval of 1, we clearly do not get enough information. Ideal information would, in this case, require one detected feature per pixel, which is by construction impossible because of the bucketing system. Also, we should ideally view the scene from all angles, which is certainly not the case in the recorded sequence.

The keyframe interval of 45 frames actually seems to yield the least noisy results. This is somewhat surprising as one may expect there to be quite a lot of error in the per-frame depth estimate. An explanation is that we detect fewer points in total, so that we, by chance, avoid the most erroneous interpolations. Combined with the depth map cropping, we simply end up with less information. Indeed, there are quite large parts of the pipes that remain unreconstructed.

The keyframe interval of 5, depicted in fig. 43b and which forms the test base case, yields a fairly good reconstruction of the pipes. There are some artifacts, e.g. the mesh that stretches from the pipes towards the ground, but one can argue that this is acceptable since the robot only traversed over the pipes. If the robot were to try to go under the pipes, the system would likely remove these artifacts because we would have more information. The keyframe interval of 30, depicted in fig. 43c, did not have the pipe-to-ground artifact issue, but gave some noise and holes in the pipes.

One takeaway from this test is that a shorter keyframe interval does not necessarily yield better results: a keyframe interval of a little over five frames looks appropriate.

Test 3. Voxel size – Results. The voxel size refers to the size of the voxels used to discretely sample the TSDF/ESDF, as described in section 2.3. The base case used a voxel size of 0.05m. We tried both smaller and larger voxel sizes. Reconstructing with a voxel size of 0.02m is shown in fig. 44a. Voxel sizes of 0.10m and 0.20m is shown in figs. 44c and 44d, respectively.



(a) Voxel size = 0.02m

(b) Voxel size = 0.05m (base case)

(c) Voxel size = 0.10m

(d) Voxel size = 0.20m

Figure 44: Results from changing the "voxel size" parameter.

Test voxel size – Discussion. If we only reconstructed with one frame, we would perhaps like the voxel size to be about the same as the expected maximum depth interpolation error. This is because we could then assume that all rays actually end up at worst one voxel away from the correct voxel, so the mesh might actually be extracted without many anomalies.

However, when we are reconstructing with more frames, we can allow the voxel size to be smaller than this, because the distance value of each voxel will be updated as we process more frames, and many frames of imprecise depth interpolation may yield a more and more precise mesh, because we get new points detected and alternative triangulation patterns of the 2D sparse depth measurements.

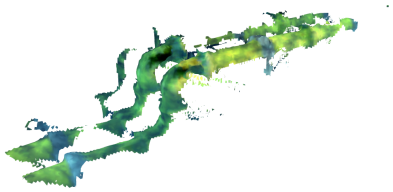
The size of the voxel also depends on the use case of the mesh. If the point of the mesh is to give only the general structure of the scene, a large voxel size may work fine. Naturally, the voxel size is a determining factor in the reconstruction time and space complexity. If we want fine details on the surface, we need a smaller voxel size.

Intuitively, a larger voxel size will smooth the mesh out more, and more of the rays traced from the camera to the voxel will be fused and averaged. Thinking in terms of limits, as the voxel size approaches zero, the reconstructed surface will approach the point cloud projected from the depth interpolation.

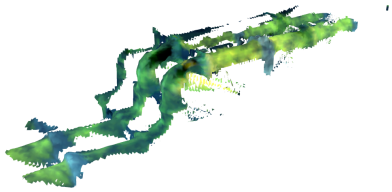
A voxel size of 0.02m, shown in fig. 44a, gives a very fine-grained reconstruction. Details are reconstructed, but errors in the depth interpolation are also greatly influencing the final mesh. A voxel size of 0.2m does not reveal as much error in depth interpolation, as can be seen from fig. 44b. However, we lose a lot of the details, and even the gap between the pipes is closed because of the increased voxel size. More details are prevalent in figs. 44b and 44c, showing the results of voxel sizes 0.05m and 0.10m, respectively. The pipe-to-ground artifact is less prevalent with a voxel size of 0.10m, but the choice of voxel size depends on the application.

A takeaway from this test is that a too small voxel size might be unnecessary given the inaccuracies in the depth interpolation. Depending on the nature of the application, a voxel size of about 10cm looks appropriate for general scene reconstruction.

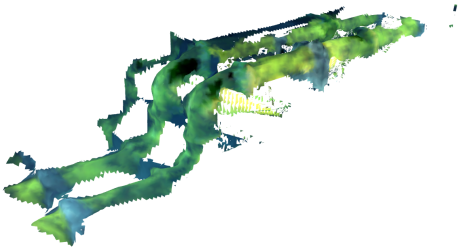
Test 4. Depth interpolation threshold – Results. The depth interpolation threshold refers to the threshold over where in the image we should not interpolate the depth, as illustrated in fig. 24, and explained in section 5. To summarize briefly, it means that: when the average distance from a point to the three vertices of the triangle in which the point is in exceeds this threshold, we do not interpolate the depth at this point, and no information about this depth is then given to Voxblox. The base case (fig. 45c) set the threshold to 25 pixels. From fig. 40b from the depth map method test, we see the results of not having any threshold. The results of having thresholds of 15, 20 and 30 pixels are shown in figs. 45a, 45b and 45d, respectively.



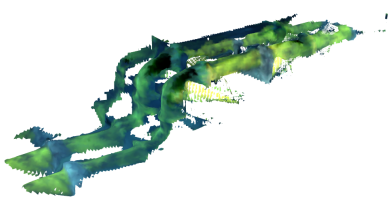
(a) Threshold = 15px



(b) Threshold = 20px



(c) Threshold = 25px (base case)



(d) Threshold = 30px

Figure 45: Results from changing the "depth interpolation threshold" parameter.

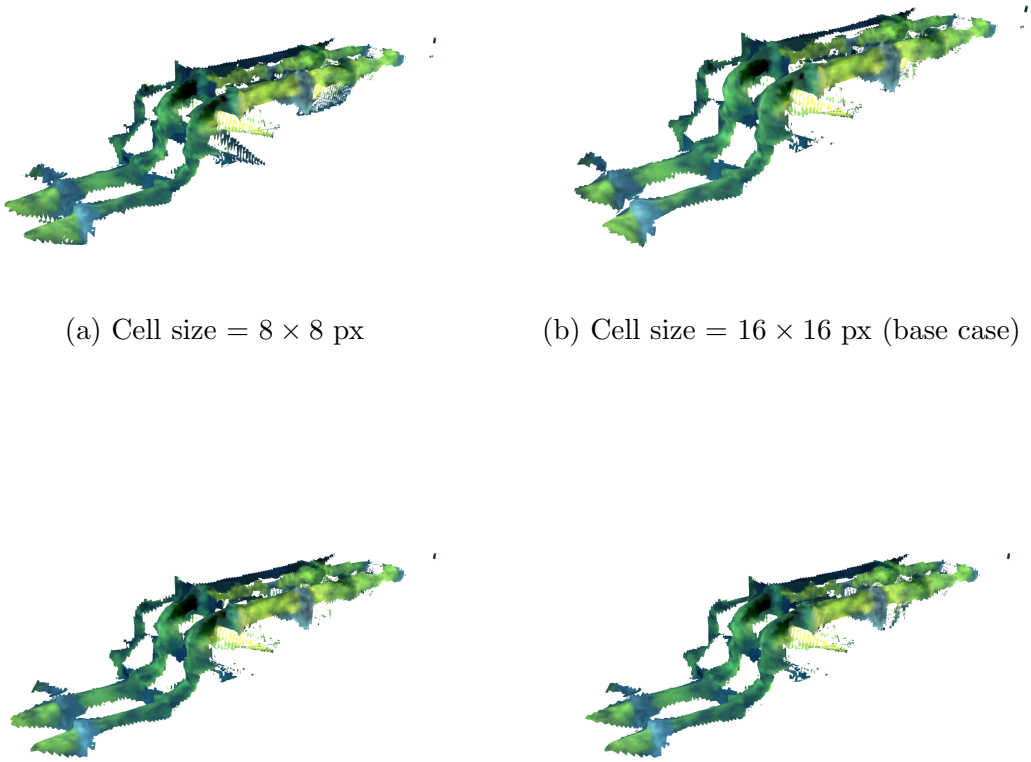
Test depth interpolation threshold – Discussion. A smaller depth interpolation threshold would give less, but more accurate, information. For example, a threshold of 1 pixel means that we only interpolate the depth if we have three keypoints at an average distance of less than 1 pixel from the pixel we want to interpolate the depth.

The result depicted in fig. 40b, from the depth map method test, we see the result of not having a threshold at all. Then, we get quite an erroneous reconstruction. The space between the pipes, for example, is filled with mesh, as it should not be. Also, there appear to be quite a lot of strange artifacts on top of the pipes.

Moving on to the threshold test, we see that many of the strange artifacts are actually removed with a threshold of 30 pixels, shown in fig. 45d. For example, most of the space between the pipes remains unreconstructed rather than being erroneously filled in. The results for this subset of the data are not much different from using a threshold of 25 pixels, i.e. the base case, shown for comparison in fig. 45c. With a threshold of 20 pixels, it appears that e.g. the pipe-to-ground artifact still mostly remains, but we have marginally less reconstructed surface. Setting the threshold also depends on the purpose, and should probably be set through trial and error. (Ferrera, 2019) found through testing that a satisfying value was 20 pixels. Lowering the threshold to 15 pixels, we start seeing, from fig. 45a, that larger patches of the surface are not reconstructed, and we can still see traces of the pipe-to-ground artifact. However, as discussed, the parts that *are* reconstructed will be more accurate.

A takeaway from this test is that there is a balance between having *enough* information and having *correct* information, and based on this test, a threshold of about 20 pixels looks appropriate for general applications.

Test 5. Cell size – Results. The cell size refers to the size of the cells in the bucketing system presented in section 4. A larger cell size would imply more sparse points in the SLAM data, and vice versa. The base case uses a 16×16 pixels size for the cells, giving a total of $\frac{1280}{16} \times \frac{720}{16} = 80 \times 45 = 3600$ cells. With a similar calculation, a cell size of 8×8 px yields 14400 cells; 40×40 px yields 576 cells; and 80×80 px yields 144 cells. All of these numbers apply per frame, so after one frame with e.g. 80×80 px cells, we can have a maximum of 144 points in the model. After several frames, more points can be involved in the reconstruction, both because some points are lost track of and replaced with new points, and because several points in the model can be projected onto the same cell. The results from adjusting the cell size parameter are shown in fig. 46.



(a) Cell size = 8×8 px

(b) Cell size = 16×16 px (base case)

(c) Cell size = 40×40 px

(d) Cell size = 80×80 px

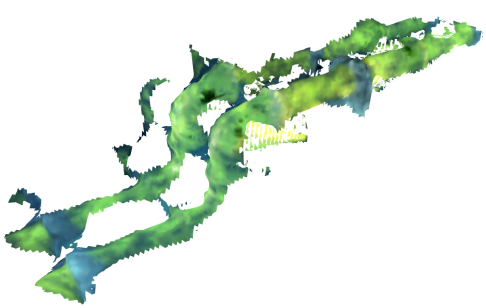
Figure 46: Results from changing the "cell size" parameter.

Test cell size – Discussion. Adjusting the cell size in the bucketing system tests how well the reconstruction performs with more or less sparse SLAM data. It is quite apparent from fig. 46 that the four test results look very similar. Only small differences reveal that it is not the same reconstruction.

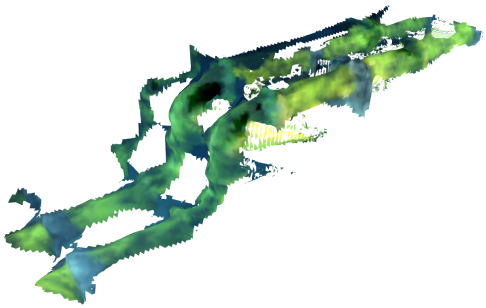
Due to the similarity of the four results, we have the following takeaway from this test: Possibly depending on the implementation, the speed of the depth interpolation is probably largely reduced with a larger cell size, so one may want to increase the cell size, i.e. the sparseness of the SLAM data, in order to obtain a lighter and faster performance, without losing much reconstructed area and accuracy.

Test 6. max ray length – Results. The *max ray length* refers the max length that Voxblox projects a ray. The reason for testing the max ray length is twofold. The first reason relates to the reconstruction *accuracy*, and the motivation behind this was presented in section 6.2. The second reason for testing the max ray length relates to the color of the mesh, which is probably of most importance to a potential human involved in the system. The points detected at a distance will probably be in dark areas of the image because we are underwater. For this reason, we hypothesize that Voxblox will assign dark colors to the mesh that arises from points and depth interpolated from a distance.

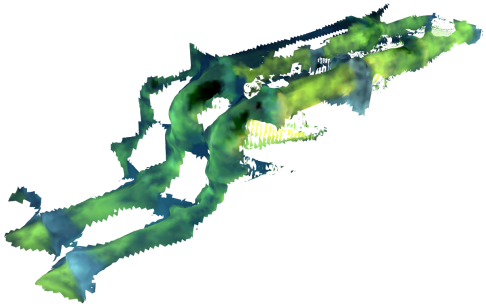
The results are presented in fig. 47.



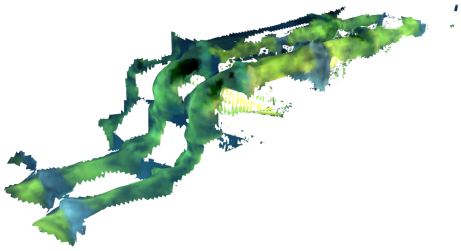
(a) Max ray length = 5m



(b) Max ray length = 10m



(c) Max ray length = 20m



(d) Max ray length = 25m (base case)

Figure 47: Results from changing the "max ray length" parameter.

Test max ray length – Discussion. When it concerns the reconstruction accuracy, we do not see very much difference across the results. Especially max ray lengths of 10m, 20m and 25m, depicted in figs. 47b to 47d, respectively, do not have very notable differences in reconstruction quality. This may be because we do not actually detect many points in the dark areas of the image, so the depth interpolation is mostly performed in the brighter areas, which are typically closer than around 5 meters from the camera anyways. Some parts of the pipes, especially the parts furthest away in the images, are not reconstructed with a ray length of 5m, seen in fig. 47a. This is probably because the robot in the simulated trajectory moves to the right of the pipes, i.e. closer to the nearest pipe from the angle the pictures are taken.

When comparing fig. 47a to fig. 47d, we see that the colors in the former mesh are somewhat brighter than in the latter. This is in line width our hypothesis. However, the effect is not as notable as we expected, probably again for the reason that most detected points are in the bright areas.

A preliminary takeaway from this test is that a limit on the max ray length might deem unnecessary, as most points are detected in bright points quite close to the camera.

6.3.2 Testset 2: Tagplate

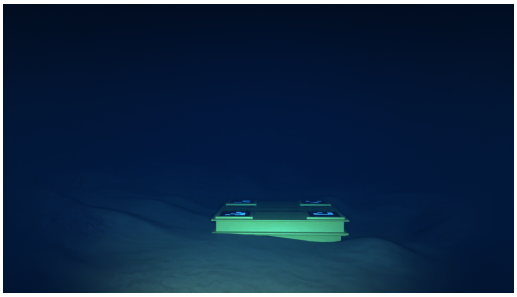
The second test used frames 5 through 95 of the VAROS dataset. The video is simulated as the robot moves over a pad with some tags, which we call the "tagplate". Four sample frames from this sequence is shown in figs. 48a to 48d. The reconstruction parameters for the reconstructions called "base case" and reconstructions 1-3 is show in table 3. Reconstructions 1 and 2 both use the ground truth depth map, i.e. no depth interpolation is involved. The only differences between reconstructions 1 and 2 are the keyframe interval and the max ray length. Reconstruction 3 and the base case both use depth interpolation with an interpolation threshold of 25 pixels. The only difference between these reconstructions is the max ray length. This test is performed for three main reasons. First, to see how well the reconstruction performs on other parts of the data. Second, to test how well the reconstruction would perform on this data with a more accurate depth estimation. Third, we test the influence of limits on the max ray length on this part of the data, to test our preliminary takeaway on this parameter from testset 1.

	Depth Map Method	KF Interval	Voxel Size [m]	Interpolation Threshold [px]	Cell Size [px]	Max ray length [m]
Base case	Int-C	5	0.05	25	16×16	25
Reconstruction 1	GT	1	0.05	N/A	N/A	5
Reconstruction 2	GT	5	0.05	N/A	N/A	25
Reconstruction 3	Int-C	5	0.05	25	16×16	5

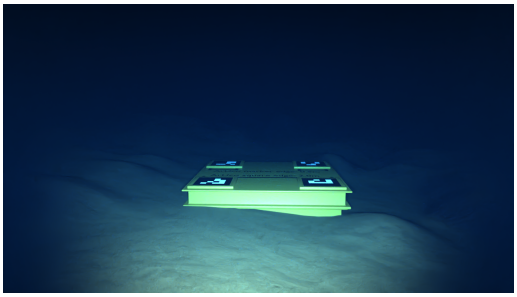
Table 3: Testset 2 parameters. Int-C and GT is as introduced previously, i.e. GT: Ground Truth depth map, and Int-C: Interpolated depth map with threshold, i.e. cropped.

The testset is selected because it is quite flat, in contrast to testset 1. In addition, it contains an object which can be detected and reconstructed, and it is thus possible to compare the reconstruction with the object.

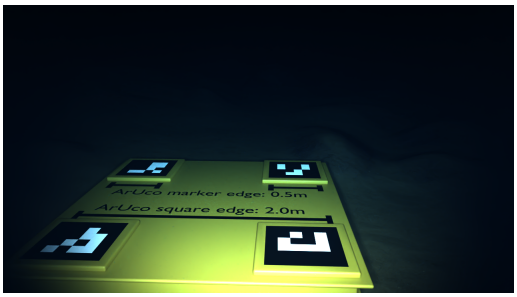
In fig. 48, we show some sample images from the sequence, as well as the resulting mesh and ESDF from using the "base case" parameters in table 3. In fig. 49, we compare the resulting meshes from using the parameters in table 3.



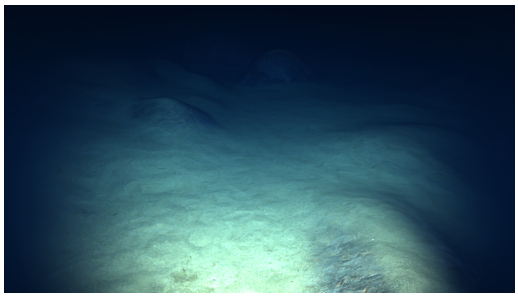
(a) Frame 5



(b) Frame 35

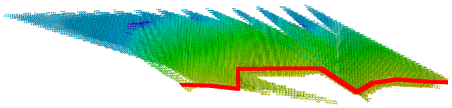


(c) Frame 65



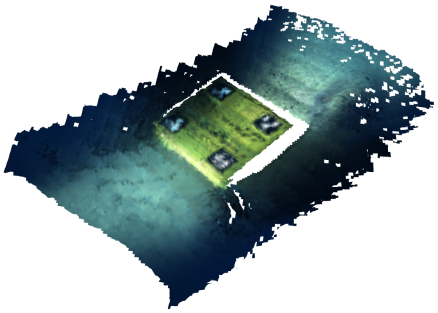
(d) Frame 95

(e) Resulting mesh from using the "base case" parameter in table 3.

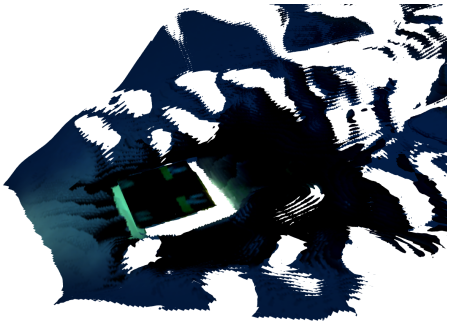


(f) Resulting ESDF from using the "base case" parameter in table 3, drawn with the approximate surface intersection.

Figure 48: Testset 2 images (a)-(d) and resulting mesh (e) and ESDF (f), shown from the same pose. For the ESDF, blue values indicate larger distances than green, and red colors indicate negative values, i.e. inside the surface model.

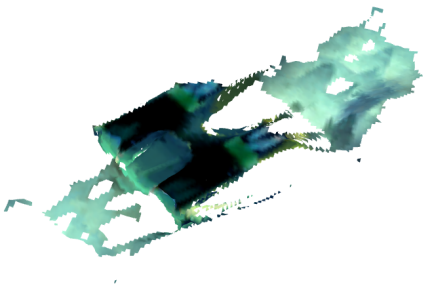


(a) Resulting mesh from using the parameters in the row “Reconstruction 1” in table 3.



(b) Resulting mesh from using the parameters in the row “Reconstruction 2” in table 3.

(c) Resulting mesh from using the parameters in the row “Reconstruction 3” in table 3.



(d) Resulting mesh from using the parameters in the row “Base case” in table 3.

Figure 49: Meshes resulting from reconstructing with parameters in table 3

Testset 2 – Discussion In what we have again called the "base case" scenario, we have used the same reconstruction parameters as in the base case for testset 1, and are repeated in table 3. We note that the ESDF is not easily interpreted accurately. It is presented as a snapshot of the 3D ESDF sampled at each voxel. The color represents the estimated Euclidean distance from the mesh. The intersection between the mesh and the ESDF is attempted drawn in with the red line in fig. 48f. Despite the visualization difficulties, the ESDF looks quite as expected. This is expected because the blue colors are further away than the green colors, and the imagined "level curves" of the colors seem to approximately follow the red line, i.e. the surface mesh. However, since all measurements are taken from approximately the same angle, the distance is not entirely accurate, and there appear to be small patches of erroneous values in the ESDF. This is because the ESDF is built directly from the TSDF, which measures distances along the sensor ray. If the robot were to traverse over the plate again in the other direction, Voxblox would probably update all voxels to more accurate values.

We see from figs. 49a and 49b that reconstruction 1 looks very accurate, and reconstruction 2 reconstructs a large area. These two reconstructions are, however, based on the ground truth depth maps, and are unrealistically good when only using a camera. The only differences between the two reconstructions are the keyframe interval and the max ray length. Increasing the max ray length naturally lets us reconstruct a larger area of the scene. However, this figure really shows the effect on the color of the mesh: The tagplate in fig. 49b is almost not distinguishable from the ground because it is so dark. One solution to this color issue is to update the color once the camera is closer to the scene, or weigh the color more heavily when merging if the camera is close to the scene.

The only difference between reconstruction 3 and the base case is the max ray length. Again, we see that in the base case, with a max ray length of 25 meters, the color on the plate is darker than in reconstruction 3, shown in fig. 49c. Also, more parts of the scene are reconstructed, as expected. We are only removing information when shortening the max ray length.

From fig. 29a from section 6.1, we see that no points are detected on the smooth areas of the tag plate. This may explain why some areas are reconstructed in the base case but not in reconstruction 3: When we are close to the scene, distances in the scene give larger distances in the image than when we are far from the scene; the same principle as in fig. 32. Hence, when we ignore points further away than 5 meters, we have to get close to reconstruct. But then, large parts of the image are cropped away, so we are left with less mesh. Once again, we conclude that a limit on the max ray length seems unnecessary.

6.3.3 Testset 3: Full Sequence

The third testset is the full released sequence from the VAROS dataset, with a keyframe interval of 5. In this test, the goal is to obtain results from the reconstructor for a lot of data in order to get a better understanding of the performance of the reconstructor and to detect possible anomalies. In addition, we want to test where the ESDF is constructed and by inspection check if the results look as expected.

For the reconstruction in fig. 50, we used the parameters for Reconstruction 1 in table 4. For the reconstruction in fig. 51, we used the parameters for Reconstruction 2 in table 4.

We use a large voxel size, as we do not wish to test for details in this reconstruction but rather for larger structures. All subfigures are snapshots of the same reconstruction. Both images in each row are taken from the same pose. The left column shows the resulting mesh, and the right column shows the corresponding ESDF.

	Depth Map Method	KF Interval	Voxel Size [m]	Interpolation Threshold [px]	Cell Size [px]	Max Ray Length [m]
Reconstruction 1	GT	5	0.2	25	16×16	25
Reconstruction 2	Int-C	5	0.2	25	16×16	25

Table 4: Testset 3 parameters. GT: Ground Truth; Int-C: Interpolated depth map with threshold, i.e. cropped far away from the points with known depth.

Figures 51a and 51b depict an overview of the scene, while figs. 51c to 51f depict the pipes mainly. Figures 51g and 51h mainly depict the tagplate.

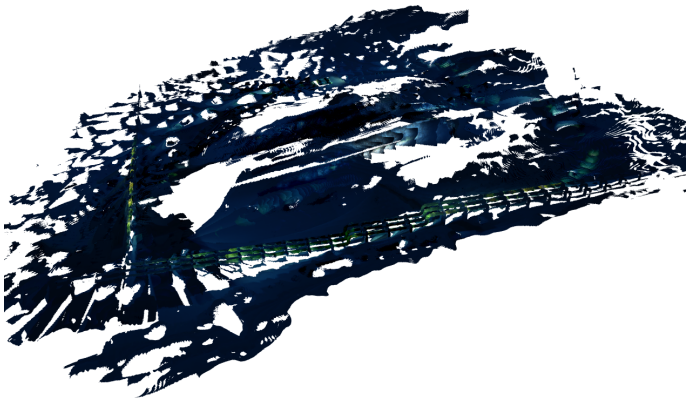
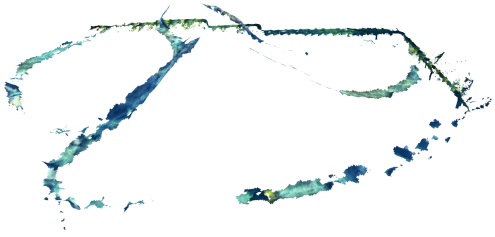
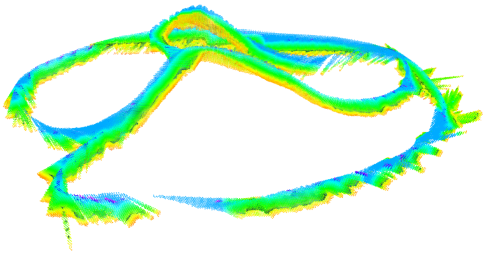


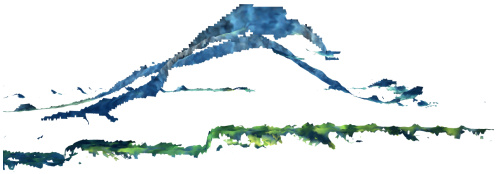
Figure 50: Resulting mesh from using the parameters of Reconstruction 1 in table 4.



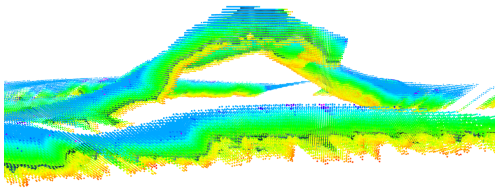
(a) Overview of the scene, mesh



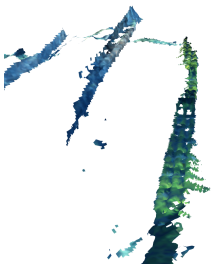
(b) Overview of the scene, ESDF



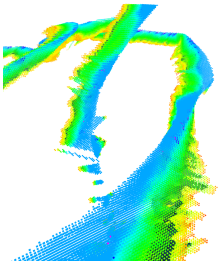
(c) Pipes from the side, mesh



(d) Pipes from the side, ESDF

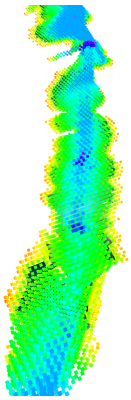


(e) Pipes from above, mesh



(f) Pipes from above, ESDF

(g) Tagplate from above, mesh



(h) Tagplate from above, ESDF

Figure 51: Full reconstruction 2. Results from reconstructing the entire sequence. All four images are taken from the same reconstruction, but from different poses. Blue ESDF colors indicate larger distances than green, and orange colors indicate negative values, i.e. behind the scene.

Testset 3 – Discussion One notable aspect of the reconstruction in fig. 50 is the dark color. Again, this is because much of the color is added to the mesh when the robot is far away and does not really illuminate the scene. Compare this to fig. 51, and we see much brighter colors. The main reason for this is probably that the detected points, and thus the interpolated areas, are located in brighter parts of the image, as in the previous test. This also results in a lot less surface area being reconstructed. This is a challenge stemming from the limited sight underwater: we can mainly reconstruct the environment closer than about 5 meters to the robot.

From fig. 50, we can see that there are quite a lot of holes in the scene. Many of these holes would be smaller or disappear if the robot had also traversed the scene in the other direction; they are mainly the result of the camera being pointed in a direction and not "seeing" what is behind the bumps. Otherwise, we see that because we use the ground truth pose, the scene is stitched nicely together.

We give some attention to the constructed ESDFs in fig. 51. From fig. 51d, the results are quite as expected: the green color over the pipes seems to mostly follow the pipes. An exception is to the left of the image, over the first elevation in the pipes, the area shown in fig. 52.

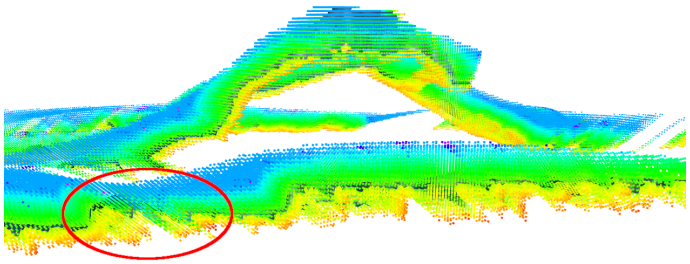


Figure 52: Area in fig. 51f with artifact.

A reason for this artifact may be that the algorithm detects points on the ocean floor, so that the system then interpolates depths from the pipe to the ground, an artifact we called "pipe-to-ground" in testset 1. Hence, the circled area is not necessarily less correct than the rest if the observed colors here are outside the pipes. It is, however, not consistent with the rest of the ESDF around the pipes.

In both fig. 51f and fig. 51h, we see that, seen from above, we have blue colors along the trajectory of the robot, green outside of that, and yellow and orange at the edges of the reconstructed areas. If the ground is flat, one may expect the entire area over the scene to be the same color, as seen from directly above. This should be the case because all points at a certain height above the ground form a "level curve" in the ESDF with the same color. Seen from above, we should only see the top layer of this ESDF.

The explanation is probably that Voxblox only allocates voxels where rays have been cast through, i.e. inside the field of view of the camera. Thus we only get an

ESDF inside this volume. This is illustrated in fig. 53, where the transparent voxels are outside the field of view (FOV) of the camera and are thus not allocated and part of the final ESDF. As a consequence, the ESDF will form a triangle-like shape above the ground, with the top having blue colors and the base having more green and orange colors.

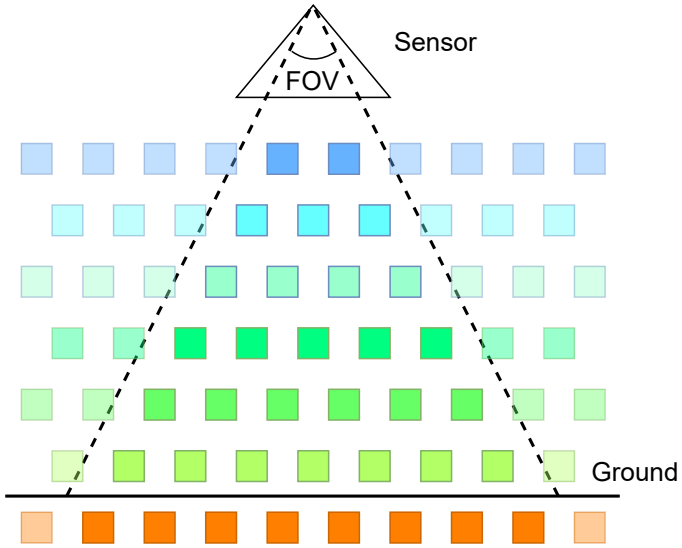


Figure 53: ESDF created from sensor with limited Field of View (FOV)

Viewed from the top, this phenomenon will look as in fig. 51h. We can vaguely identify the triangle shape of the ESDF in the right part of fig. 51f. It is a little twisted because the robot did not move straight along the pipes but slightly from left to right.

Conclusively, although the ESDF in fig. 51h may look a bit strange from above, we argue that it is in fact as expected.

7 Conclusion and Future Work

In this final section, we summarize our contributions and main findings and provide some suggestions for areas of future work.

7.1 Conclusion

In this thesis, we have completed a study of the visual artifacts obtained in the results of the specialization project and motivated our decision to change to another third-party surface reconstruction software. Furthermore, we have improved the SLAM emulation algorithm. The improvement is twofold: First, we now perform data association on points across the frames using a developed tracking method utilizing the ground truth depth map. Second, we have implemented a bucketing system, implicitly offering the user larger freedom in adjusting the point cloud density.

In section 5 we implemented a method for interpolating the depth in an image using only the depth of the observed SLAM points, which will be available in a real system (though not as precise as in a simulated dataset). The method can crop the depth interpolation at a user-specified distance from the detected points in order to exclude the worst interpolations. In the same section, we showed how the entire system can be combined with a third-party surface reconstructor, called Voxblox, to perform surface reconstruction from SLAM data.

The modular system allows for flexibility: the simulated SLAM data can be exchanged with real SLAM data, and the system can be used in real applications. Reliant on a working SLAM algorithm, the system will fail wherever the SLAM algorithm fails. Furthermore, the depth interpolation method was found to be too slow to obtain real-time performance. Conveniently, given the modular structure, the SLAM method can be chosen freely, and the depth interpolation can be exchanged for a more sophisticated method. One can also exchange the third-party surface reconstructor or custom-make one.

Testing the surface reconstruction parameters, we found that suitable values were: a keyframe interval of a little over 5, voxel size of about 10cm, depth interpolation threshold of 20 pixels, a large cell size, possibly 40×40 pixels or bigger depending on the application, and a max ray length of about 25 meters.

7.2 Future Work

Some areas of future work may be performed to improve the suggested reconstruction scheme. There are mainly four improvements we suggest.

(i) The first area of future work relates to incorporating uncertainty information about the depth estimation into Voxblox. Voxblox already has a weighting scheme

based on the depth of the estimation. This weighting scheme is tailored to a typical LiDAR scanner, with larger relative uncertainty at greater distances. However, we know that the depth interpolation is more inaccurate than the SLAM points, and it is natural to infer that the uncertainty of the interpolation estimate increases the further in the image we are from the tracked SLAM points. This info can be incorporated into the Voxblox weighting scheme. We suggest a couple of different weights, which can be tested and compared. The first is to have a weight function that is 1 at the SLAM points and decreases linearly to 0 where the average distance to the tracked SLAM points is at the defined threshold according to eq. (5.6). Another weighting scheme can be to have a weight that is inversely proportionate to this average distance. One more suggestion that can be added in addition is based on the assumption that the uncertainty in an interpolated triangle is large if the three SLAM point vertices have very different depth values. The weight at a pixel can then e.g. be inversely proportional to the difference in the depth to a co-located pixel.

(ii) Another area of future work can be to test the system with a more sophisticated depth estimation method. Some examples were mentioned in section 6.3.1, in the discussion relating to the choice of the depth map estimation method. To repeat some examples, (Ochs et al., 2017) uses Principal Component Analysis, and (Pham et al., 2006), uses normalized convolution to interpolate data from incomplete samples.

(iii) The third area of future work we suggest relates to the mesh coloring in Voxblox. In the results, we found that when the depths are interpolated and projected from a distance, Voxblox collects colors from quite dark areas in the image. Voxblox appears to fuse color measurements with equal weight regardless of where the robot is relative to the surface. One solution can be to update the color of the mesh at voxel once the robot is closer than it was when it previously applied color to the mesh, instead of fusing the colors together. Alternatively, one can implement a weighting scheme that puts more weight on the colors applied to the mesh when the robot is closer to the scene. Intuitively, this would brighten and diversify the colors applied to the mesh and give a more intelligible mesh.

(iv) The final improvement we suggest relates to the time complexity of the depth interpolation. We found that our implementation did not satisfactorily reach the time constraints set by real-time applications. Another line of future work could thus be to improve this implementation. For example, one can store the index of the previous triangle to check if the next pixel resides in the same triangle. This will often be the case if we loop through each pixel in the image.

Bibliography

3D at Depth. (2022). *SL4 - a Step to higher Speed remote operations*. Retrieved 21st May 2022, from https://3datdepth.com/product/sl4

Bærentzen, J. A., Gravesen, J., Anton, F. & Aanæs, H. (2012). Volumetric Methods for Surface Reconstruction and Manipulation. *Guide to computational geometry processing* (pp. 287–308). Springer London. https://doi.org/10.1007/978-1-4471-4075-7_17

Bernardini, F., Mittleman, J., Rushmeier, H., Silva, C. & Taubin, G. (1999). The ball-pivoting algorithm for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, *5*(4), 349–359. https://doi.org/10.1109/2945.817351

Campos, C., Elvira, R., Gómez, J. J., Montiel, J. M. M. & Tardós, J. D. (2020). ORB-SLAM3: An accurate open-source library for visual, visual-inertial and multi-map SLAM. *arXiv preprint arXiv:2007.11898*.

Chan, T. & Zhu, W. (2005). Level set based shape prior segmentation. *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, *2*, 1164–1170 vol. 2. https://doi.org/10.1109/CVPR.2005.212

Curless, B. & Levoy, M. (1996). A volumetric method for building complex models from range images. *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, 303–312.

Eldesokey, A., Felsberg, M., Holmquist, K. & Persson, M. (2020). Uncertainty-aware cnns for depth completion: Uncertainty from beginning to end. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 12014–12023.

Ferrera, M. (2019). *Monocular visual-inertial-pressure fusion for underwater localization and 3d mapping.* (Doctoral dissertation). Université Montpellier.

Hornung, A., Wurm, K. M., Bennewitz, M., Stachniss, C. & Burgard, W. (2013). OctoMap: An efficient probabilistic 3D mapping framework based on octrees. *Autonomous Robots*, *34*(3), 189–206. https://doi.org/10.1007/s10514-012-9321-0

Kazhdan, M., Bolitho, M. & Hoppe, H. (2006). Poisson surface reconstruction. *Proceedings of the fourth Eurographics symposium on Geometry processing*, *7*.

Khatamian, A. & Arabnia, H. R. (2016). Survey on 3D Surface Reconstruction. *Journal of Information Processing Systems*, *12*(3), 338–357. https://doi.org/10.3745/JIPS.01.0010

Knutsson, H. & Westin, C.-F. (1993). Normalized and differential convolution. *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 515–523.

Lau, B., Sprunk, C. & Burgard, W. (2010). Improved updating of euclidean distance maps and voronoi diagrams. *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 281–286.

Lhuillier, M. (2014). 2-manifold tests for 3d delaunay triangulation-based surface reconstruction. *Journal of Mathematical Imaging and Vision*, *51*, 98–105.

Lorensen, W. E. & Cline, H. E. (1987). Marching cubes: A high resolution 3d surface construction algorithm. *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, 163–169. https://doi.org/10.1145/37401.37422

Mur-Artal, R., Montiel, J. M. M. & Tardos, J. D. (2015). Orb-slam: A versatile and accurate monocular slam system. *IEEE transactions on robotics*, *31*(5), 1147–1163.

Newcombe, R. A., Izadi, S., Hilliges, O., Molyneaux, D., Kim, D., Davison, A. J., Kohli, P., Shotton, J., Hodges, S. & Fitzgibbon, A. (2011). KinectFusion: Real-time dense surface mapping and tracking. *2011 10th IEEE International Symposium on Mixed and Augmented Reality, ISMAR 2011*, 127–136. https://doi.org/10.1109/ISMAR.2011.6092378

Nguyen, C. V., Izadi, S. & Lovell, D. (2012). Modeling kinect sensor noise for improved 3d reconstruction and tracking. *2012 Second International Conference on 3D Imaging, Modeling, Processing, Visualization Transmission*, 524–530. https://doi.org/10.1109/3DIMPVT.2012.84

Nießner, M., Zollhöfer, M., Izadi, S. & Stamminger, M. (2013). Real-time 3d reconstruction at scale using voxel hashing. *ACM Transactions on Graphics (ToG)*, *32*(6), 1–11.

Ochs, M., Bradler, H. & Master, R. (2017). Learning rank reduced interpolation with principal component analysis. *2017 IEEE Intelligent Vehicles Symposium (IV)*, 1126–1133.

Oleynikova, H., Millane, A., Taylor, Z., Galceran, E., Nieto, J. & Siegwart, R. (2016). Signed distance fields: A natural representation for both mapping and planning. *RSS 2016 Workshop: Geometry and Beyond-Representations, Physics, and Scene Understanding for Robotics*.

Oleynikova, H., Taylor, Z., Fehr, M., Siegwart, R. & Nieto, J. (2017). Voxblox: Incremental 3d euclidean signed distance fields for on-board mav planning. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

Pham, T. Q., Van Vliet, L. J. & Schutte, K. (2006). Robust fusion of irregularly sampled data using adaptive normalized convolution. *EURASIP Journal on Advances in Signal Processing*, *2006*, 1–12.

Piazza, E., Romanoni, A. & Matteucci, M. (2018a). Real-time cpu-based large-scale three-dimensional mesh reconstruction. *IEEE Robotics and Automation Letters*, *3*(3), 1584–1591.

Piazza, E., Romanoni, A. & Matteucci, M. (2018b). *Realtime-manifold-mesh-reconstructor*. Retrieved 31st May 2022, from https://github.com/Enri2077/realtime-manifold-mesh-reconstructor

Romanoni, A. & Matteucci, M. (2015). Incremental reconstruction of urban environments by Edge-Points Delaunay triangulation. *IEEE International Conference on Intelligent Robots and Systems*, *2015-Decem*, 4473–4479. https://doi.org/10.1109/IROS.2015.7354012

Romanoni, A. & Matteucci, M. (2018). Real-Time CPU-Based Large-Scale Three-Dimensional Mesh Reconstruction. *3*(3), 1584–1591.

Ros wiki depth_image_proc [Accessed: 2022-04-20]. (n.d.).

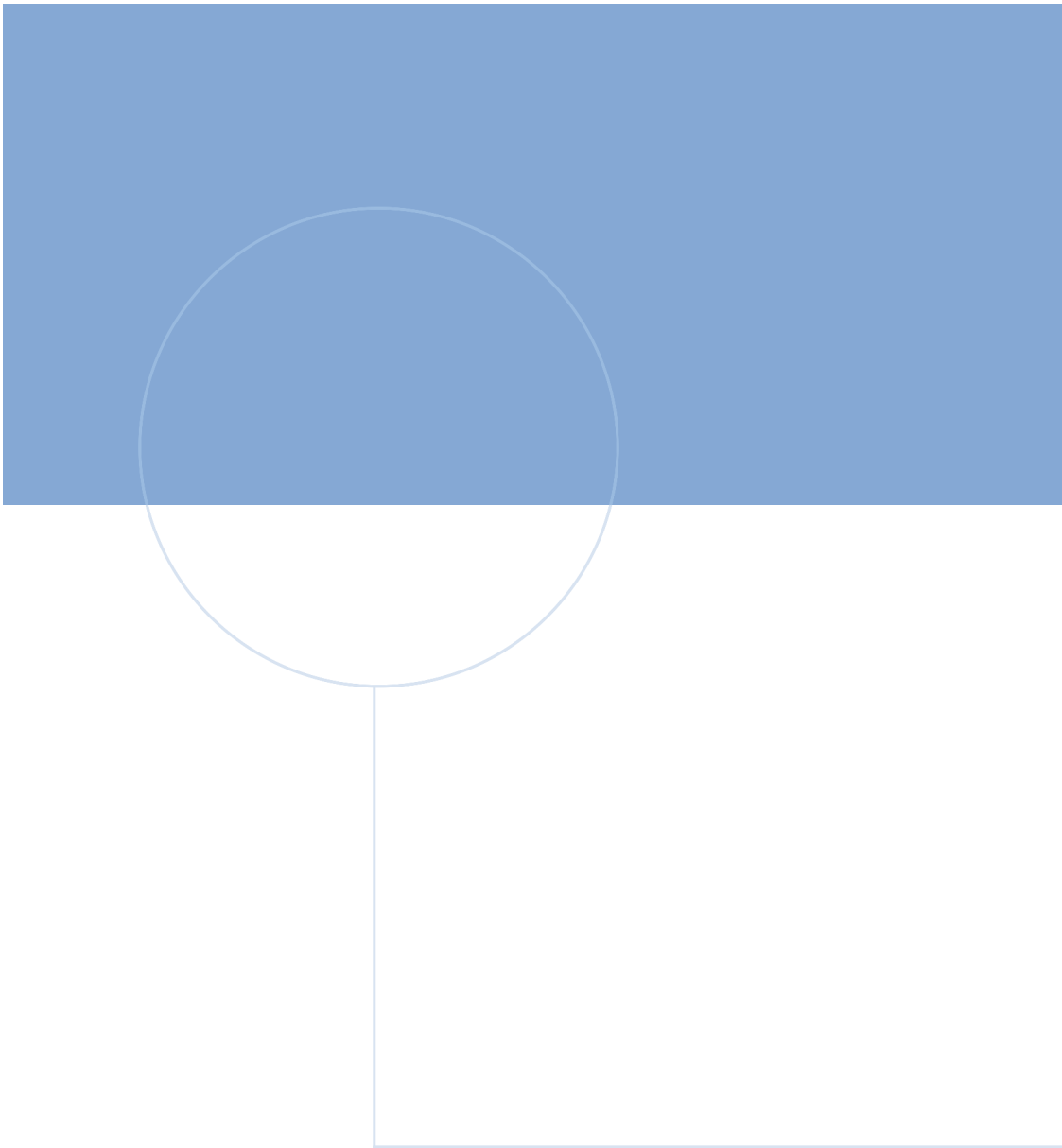
Rublee, E., Rabaud, V., Konolige, K. & Bradski, G. (2011). Orb: An efficient alternative to sift or surf. *2011 International Conference on Computer Vision*, 2564–2571. https://doi.org/10.1109/ICCV.2011.6126544

Schmidt, M. S. (2021). Underwater real-time incremental surface reconstruction from dynamic 3d point clouds [Internal Report, Department of Engineering Cybernetics, NTNU].

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., ... SciPy 1.0 Contributors. (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, *17*, 261–272. https://doi.org/10.1038/s41592-019-0686-2

Zucker, M., Ratliff, N., Dragan, A. D., Pivtoraiko, M., Klingensmith, M., Dellin, C. M., Bagnell, J. A. & Srinivasa, S. S. (2013). Chomp: Covariant hamiltonian optimization for motion planning. *The International Journal of Robotics Research*, *32*(9-10), 1164–1193.

Zwilgmeyer, P. G. O., Yip, M., Teigen, A. L., Mester, R. & Stahl, A. (2021). The varos synthetic underwater data set: Towards realistic multi-sensor underwater data with ground truth. *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 3722–3730.



NTNU

Norwegian University of
Science and Technology