NTNU
Norwegian University of
Science and Technology

Department of Engineering
Cybernetics

TTK4550 - Engineering Cybernetics,
Specialization Project

Underwater Real-Time 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

20th December 2021
Trondheim, Norway

Abstract

For an autonomous robot, having a map of its surroundings is crucial to perform certain tasks such as path planning and interaction with the environment. In this project report, we implement a method for emulating VSLAM data on a synthetic dataset of an underwater scene. In addition to the raw images, the dataset contains ground truth pose of the camera and depth at each pixel, which is utilized for projecting points into space. Both random sampling of points in each frame and the use of ORB feature detection and matching is implemented and compared. We then test a state-of-the-art real-time incremental surface reconstruction method – identified through a literature review – on the emulated VSLAM data, and compare the results obtained from different parameter settings for the reconstructor and VSLAM data. The reconstruction accuracy and speed is evaluated qualitatively and quantitatively, respectively.

Preface

This report constitutes the Specialization Project 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. The goal of the AROS project is to improve existing marine robotics technology in order to achieve full autonomy in subsea operations, including underwater sensing, situational awareness, and motion planning, in addition to energy autonomy. This allows for the ability to perform demanding observation and intervention tasks, in order to achieve greener, safer and more cost-efficient operations. This report is a part of work package two of project AROS: *Next-best-view and 3D reconstruction for AIAUVs*.

[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.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

The last decades have seen an increasing amount of autonomy in robots. Increasingly, these robots operate in rather unconventional environments. For example, one may want an underwater vehicle, such as the Eelume vehicle (Eelume AS, 2021), to operate on the ocean floor. The conditions here are challenging, both from a control perspective and from a sensing perspective. Several of the sensors used above water have reduced functionality, or are simply unusable underwater. Alternatively, they are usable but expensive, heavy, and thus not suitable for an autonomous underwater robot. In recent years, Visual Simultaneous Location and Mapping (VSLAM) methods based purely on RGB-cameras have gained much attention, and have become increasingly accurate. An example of a state-of-the-art VSLAM system is ORB-SLAM3 (Campos et al., 2020), which uses the ORB feature detector (Rublee et al., 2011).

From a VSLAM system, the result will often be an incrementally adjusted point cloud in $\mathbb{R}^3$. However, this point cloud is usually insufficient for an agent to infer enough information about its environment to carry out its expected tasks. For an underwater robot, such tasks could be path planning, inspection, cleaning, and operating valves. Therefore, one may want to *reconstruct* a surface from the point cloud. This reconstructed surface should – in some sense – minimize the difference between the actual surface of the scene and the reconstructed surface. At the same time, the reconstruction should be fast and lightweight, in order to gain real-time performance. This is especially crucial for path planning and collision avoidance.

Several different reconstruction methods exist, and there is usually a trade-off between different performance measures, notably time-complexity, space-complexity, and some measure of geometric and topological accuracy. Furthermore, different methods assume different properties on the input data, and will seek to optimize certain properties of the output data. There is also an element of mathematical complexity of the methods, and this report will therefore also provide some mathematical preliminaries.

1.2 Objective and Contribution

The objective of this project report is twofold. The first objective is to identify a suitable real-time surface reconstruction method that only uses information about: i) the images and detected points in each image, ii) the pose of the camera at each image, and iii) the location of the points in the point cloud. We will identify such a reconstruction method through a literature review. When a method has been

identified, we want to test the method on a synthetic dataset of an underwater scene, which is a precursor to the VAROS dataset (Zwilgmeyer et al., 2021). VSLAM data for this dataset is not readily available, but can be obtained either by running an existing VSLAM method on the dataset or by using the accompanying ground truth depth images and poses to *emulate* VSLAM data from the dataset. The latter has the advantage of being more easily customized and more robust.

The second objective is thus to emulate VSLAM data on the synthetic dataset. Then, we will evaluate the selected surface reconstruction method on the emulated VSLAM data. The reconstruction accuracy and speed will be tested in various ways and presented qualitatively and quantitatively, respectively.

The main contribution of this report is to provide a method for emulating VSLAM data on the synthetic dataset, which can be directly applied to the VAROS dataset. The emulated VSLAM data can readily be used for a variety of purposes, such as surface reconstruction, testing loop closure methods, and comparing different VSLAM methods in underwater environments. We will use the emulated VSLAM data to test the performance of an incremental surface reconstruction method.

1.3 Structure of Report

We will first explore some relevant literature in section 2, including some preliminary mathematics, some surveys on 3D surface reconstruction with further references, and different methods of reconstructing surfaces from static and dynamic 3D point clouds. In section 3, we will present our method for emulating VSLAM data on the synthetic dataset. In section 4, we will test a selected surface reconstruction method on the emulated VSLAM data, and the results will be discussed in section 5. In section 6, we will present some potential areas of further work. Lastly, we add some conclusive remarks.

To avoid confusion, we will usually refer to surface reconstruction methods as "surface reconstructors", while "our method" refers to the method of emulating VSLAM data.

2 Selected Literature

In this section, we will present selected literature relating to surface reconstruction from point clouds. For all definitions, we will see how they relate to the subject of surface reconstruction. First, in the preliminaries, we will touch upon the topics of manifoldness and its relation to the Delaunay triangulation. Second, there will be presented a summary of different techniques in surface reconstruction from point clouds, through selected surveys. Lastly, we will build our way to a state-of-the-art incremental surface reconstructor through a review of the historical progress of these methods. There are many different branches of this active research area, and we will focus on the history of methods leading up to the surface reconstructor to be discussed in this report.

2.1 Preliminaries

Topology and manifolds. Informally, topology is the study of spatial structure and properties of geometric objects that are preserved under continuous deformations: It studies how objects are "put together" rather than the exact shape of objects. An example of a topological space is an n-dimensional manifold, which is a space that locally is homeomorphic to Euclidean space. For example, a two-dimensional manifold, or 2-manifold, is called a surface, which locally resembles the Euclidean plane. More formally, let $S \subseteq \mathbb{R}^3$ be the surface. If all points in S have a neighborhood in S that is homeomorphic to a disk, then S is a 2-manifold in $\mathbb{R}^3$ (Lhuillier, 2014). Manifolds in some sense extend the notion of Euclidean spaces to include curved spaces, where regular notions of angles and distances generally do not hold. The *genus* of a surface is, informally, the number of holes in the surface.

Delaunay triangulations and Voronoi diagrams. A Delaunay triangulation is often defined through its dual, the *Voronoi diagram*. Given a set P of N vertices in a 3D Euclidean space, $P = \{P_1 \cdots P_N\}$, the Voronoi diagram is a sequence $V_1 \cdots V_N$ of convex polyhedra where all points in V_i have P_i as its nearest vertex. Formally,

$$V_i = \{x \in \mathbb{R}^3 : \forall j, 1 \leq j \leq N, |x - M_i| \leq |x - M_j|\}, \tag{1}$$

as described in (Boissonnat, 1984). Now, if we link together the vertices whose Voronoi cells are adjacent, we obtain a *Delaunay triangulation* of the original vertices, resulting in a set of tetrahedra. This set of tetrahedra satisfies what we will call the *Delaunay property*. A triangulation of a set satisfies the Delaunay property if none of the vertices in the set lies in the circumsphere of any tetrahedron in the triangulation. A circumsphere of a tetrahedron is a sphere that passes through all

vertices in the tetrahedron.

A significant advantage of the Delaunay triangulation for surface reconstruction is that it can be computed efficiently. This makes it suitable for real-time applications, where processing speeds are crucial.

Another advantage is that if we view the vertices as samples of a given surface, then the boundary of a Delaunay triangulation for a set of vertices can be viewed as its convex hull, which is an approximation of the surface of the object. In fact, (Boissonnat, 1984) states that it is the best polyhedral approximation of the surface according to criteria such as distance between the approximation and the object.

Further, one may impose several properties on the boundary of a Delaunay triangulation of an object sampled at the surface. One such property is that it is a 2-manifold. This property is useful because it allows for descriptions of the surface in terms of surface normal and curvature which is used by many mesh processing algorithms, e.g. smoothing (Litvinov & Lhuillier, 2013). Intuitively and informally, it means that the surface behaves "nicely" in that it does not allow for self-crossings and each direction is well defined. Also, the Delaunay triangulation has "theoretical warranties for surface reconstruction in both geometric and topological senses" (Litvinov & Lhuillier, 2014).

There are many ways to construct a Delaunay triangulation. Some examples provided in (Su & Drysdale, 1997) are "Dwyer's divide and conquer algorithm, Fortune's sweepline algorithm, [...] and Barber's convex hull based algorithm". Of these, they showed Dwyer's divide and conquer algorithm to be the fastest. (Biniaz & Dastghaibyfard, 2012) proposes an even faster, circle-sweep Delaunay triangulation algorithm.

Manifold checks for triangulated surfaces. We define three tests to check whether a given surface reconstruction is manifold: The general test; the subtraction test; and the addition test.

First, we need some definitions. We use a notation inspired by (Lhuillier, 2014) and (Litvinov & Lhuillier, 2013).

Given vertices $\mathbf{a}, \mathbf{b}, \mathbf{c}$ in space, let $\mathbf{ab}, \mathbf{ac}, \mathbf{bc}$ be the undirected edges connecting the vertices, and let $\mathbf{abc}$ denote the triangle formed by the vertices. The $\mathbf{a}$-*opposite* edge is then $\mathbf{bc}$. Given a list L of tetrahedra formed by a triangulation of some space, let δL denote the list of triangles included in exactly one tetrahedron in L, i.e. the triangles on the boundary of L. Let $|\delta L|$ denote the union of triangles in δL, forming the boundary of L.

Now, let T be a triangulation of a space. Let $O \subseteq T$ be a set of tetrahedra such that every triangle in δO is bounded. The task at hand is to show that $|\delta O|$ is manifold.

The general test is passed if and only if for every vertex $\mathbf{v}$ in every triangle of δO, the following property holds: All triangles in δO that have $\mathbf{v}$ as a vertex can be ordered as $t_0 \cdots t_k$ such that $t_i \cap t_{(i+1) \mod (k+1)}$ is an edge, and this edge is contained in exactly two triangles (Lhuillier, 2014). An alternative formulation is that the $\mathbf{v}$-opposite edges must form a cyclic graph (Litvinov & Lhuillier, 2013).

Now let $\Delta \in O$ denote a tetrahedron. Let $|\delta(O \setminus \Delta)|$ denote the boundary obtained by removing Δ from O, in our case the new surface. If $|\delta O|$ is manifold, then $|\delta(O \setminus \Delta)|$ is manifold if the combination of number of (vertices,edges,triangles) is in $\{(0,0,0),(3,3,1),(4,5,2),(4,6,3),(4,6,4)\}$. We call this the subtraction test on $\{\Delta, O\}$.

The addition test on $\{\Delta, O\}$ is similar to the subtraction test, except that $\Delta \in T \setminus O$ and we check for manifoldness on $|\delta(O \cup \Delta)|$ (Litvinov & Lhuillier, 2013).

2.2 Surface Reconstruction from Point Clouds

2.2.1 Surveys

Survey on 3D Surface Reconstruction (Khatamian & Arabnia, 2016) discusses some issues encountered in surface reconstruction, most of which are discussed in further detail in (Berger et al., 2016), and will be presented in the paragraphs related to that survey. For now, we restrict ourselves to a summary of a typical constraint faced in surface reconstruction: The dataset size. Some methods are not able to handle very large datasets, due to a lack of available hardware resources. Furthermore, there is a trade-off between accuracy and performance in surface reconstruction algorithms. Accuracy can be measured in many different senses, for example topological consistency and geometric accuracy, and so can performance. One can for example either look at spatial complexity or time complexity. Filtering and simplification are two measures that can be taken for increasing the performance. Filtering reduces unwanted or unnecessary input points while simplification samples the input to decrease the input size. Both of these techniques increase the performance by reducing the size of the input data, but might decrease the accuracy for the same reason.

In (Khatamian & Arabnia, 2016), they also present different ways of classifying surface reconstructors. There are three main ways of doing so:

i) Explicit vs. Implicit. An *explicit* method yields a surface that is explicitly defined, in that a function precisely defines the location of the surface. There are two main distinctions within the explicit surfaces: parametric and triangulated. To quote the paper, "A parametric surface is the deformation of a primitive model that covers an arbitrary portion of the points". The other type of explicit sur-

faces is created from connecting points with their neighbors to obtain triangles or tetrahedra, and is thus called triangulated. It is a very intuitive representation of a surface, but may not be resilient to different artifacts of the input data. One such method is the Ball-Pivoting Algorithm (Bernardini et al., 1999), which will be elaborated on in section 2.2.2.

An *implicit* surface is usually described by a function at a constant value, i.e. the points

$$\mathbf{x} \in \mathbb{R}^3 \text{ s.t. } f(\mathbf{x}) = \gamma, \tag{2}$$

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

There are two types of implicit surfaces: variational and typical. Typical implicit surfaces use radially symmetric basis functions. One important typical implicit surface reconstruction is the Poisson surface reconstruction (Kazhdan et al., 2006), which will also be presented in section 2.2.2. Variational implicit surfaces use a variety of basis functions to represent more complex shapes.

Several kinds of functions are used to represent implicit surfaces. Regular choices are distance functions and convolutions of symmetric functions. The first is related to cost minimization techniques and the latter to fitting techniques.

The process of finding an implicit surface consists of two parts. First, one needs to find a function that fits the input data, and then it has to be post-processed in order to be visualized, for example by marching cubes (Lorensen & Cline, 1987), because the implicit function is usually quite complex.

ii) Interpolated vs. Approximated. Another distinction found in literature is whether the method is *interpolated* or *approximated*. The distinction is not too different from the explicit versus implicit distinction, as interpolated methods are usually explicit and approximated methods are usually implicit. For a surface to be interpolated, at least a subset of the input points must lay on the reconstructed surface. This is not a requirement for approximated surfaces, as these surfaces usually involve some kind of minimization problem over parts of – or the entire – surface.

iii) Anisotropic vs. Isotropic. Informally, a manifold is *isotropic* if measurements coming from all directions are the same. In this sense, it relates to how smooth the surface is, and that it does not have sharp edges, corners, instantaneous jumps, and other features where measurements depend on the direction. If the surface has such features, it is *anisotropic*.

A Survey of Surface Reconstruction from Point Clouds (Berger et al., 2016) mostly discusses priors that can be utilized in the reconstruction, but also touches upon issues and alternative data in the point cloud. We will split the discussion in three parts. Part A discusses common issues with the input point cloud that should be dealt with. Different methods clearly have different degrees of resilience against these issues. Part B is a short discussion on what types of data accompany the input point cloud, and how these can be utilized. Part C examines how different a priori knowledge, or priors, in short, can be utilized to create the best surface reconstruction for the purpose.

A. Common issues with input point cloud

- **Non-uniform sampling density**
  Typically, 3D point clouds originating from real scanners have *spatially varying sampling densities*. This causes trouble for defining a local *neighborhood* around a point. A neighborhood is a set of points "close" to a given point. One method is to select the k nearest neighbors (knn) of the point. Another is to define a ball of radius ϵ and choose the points in this ball centered at the given point. More sophisticated methods exist, but will not be discussed here.
- **Noise**
  3D point clouds originating from images or scanners are not perfect and are usually subject to some level of *noise*. Noise is considered to be a small random displacement of a point on the surface, with the distribution depending on different aspects of the scanner. In the presence of noise, one should be careful interpolating every point exactly, as the points generally will not be exactly on the surface.
- **Outliers**
  An *outlier* is a misdetected point that is not near the surface. They can be considered to be distributed in space according to some distribution. These points should ideally be disregarded when constructing the surface, usually through robust methods like RANSAC.
- **Misalignment**
  Misalignment is a common source of error in VSLAM based point cloud construction. This is because one tends to have drift in these systems, resulting in an increasingly erroneous estimated pose. Thus, during loop closure, one can have two misaligned scans of the same object, which has to be dealt with, for example by adjusting the estimated pose of the scanner.
- **Missing data**
  Missing data refers to regions of the surface where the sampling density equals

zero. They are a common artifact of vision-based surface reconstruction due to occlusions and textureless regions. Several assumptions can be applied to the surface to deal with this problem, which will be discussed in the next segments.

B. Point cloud input that can be utilized

- **Surface normals**

  The *normals* of the points in the point cloud can be considered as a sampled normal of the surface, and is very useful in certain reconstructors, such as the Poisson surface reconstruction. They may be constructed from the point cloud alone, and can be either *oriented* or *unoriented*. They are usually computed using some notion of the neighborhood of the point, e.g. an ϵ-ball or knn, for estimating the local geometry of the shape. One way to obtain an unoriented normal is to use principal component analysis: Let o_i be the centroid of the neighborhood of x_i, $Nbhd(x_i)$, and let $\otimes$ define the outer product of two vectors. Defining the matrix

$$N = \sum_{y \in Nbhd(x_i)} (y - o_i) \otimes (y - o_i) \tag{3}$$

  we let n be the smallest eigenvector of N, or its negative (i.e. we do not know its orientation). A way to construct an oriented normal is to take the output of the previous procedure, connect the points in a graph, and let the weights in the graph's edges correspond to the similarity of the edge's vertices normals. Then, starting from a seed point, one can propagate the normal orientation over the resulting graph. Both of these methods are described in more detail in (Hoppe et al., 1992).

- **Scanner/Camera information**

  Scanner information can give useful information about the reliability of the point cloud, in regards to outliers, noise, and shape. For example, if we know that a LIDAR has been used to scan a surface that is very tilted with respect to the scanner, then we know that the confidence of these points is low. Further, RGB-cameras can be used to complement other scanners, finding features where typical depth sensors may not. RGB-cameras can also be used alone, as is typical in VSLAM methods.

C. Priors that can be utilized

- **Surface smoothness priors**

  We first introduce some notation. Let P be a point cloud sampling of a shape

S. Points in P are denoted as $\mathbf{p}_i \in P$. There may also be a normal field N such that for each $\mathbf{p}_i$ there is an accompanying normal $\mathbf{n}_i \in N$, i.e. the surface normal at each point is part of the input data. Different kinds of surface smoothness surface priors can be imposed on the surface. They can be divided into two main useful priors: local and global smoothness.

Local smoothness assumes that the surface is smooth only close to the data. (Hoppe et al., 1992) used the method of signed distance fields to recreate a surface, by finding an approximation of the distance of each point in space to the surface, i.e. by finding a function $\Phi : \mathbb{R}^3 \to \mathbb{R}$, that maps every point $\mathbf{x}$ in space to its distance to the surface, i.e.

$$\Phi(\mathbf{x}) = ||(\mathbf{x} - \mathbf{p})|| \tag{4}$$

where $\mathbf{p}$ is the closest point on the surface to the arbitrary point $\mathbf{x}$. Since we only have sample points of the surface we could instead take the distance from $\mathbf{x}$ to the tangent plane of the point $\mathbf{p}_i$, using a first-order approximation of the surface around $\mathbf{p}_i$. The resulting distance function is then

$$\Phi(\mathbf{x}) = (\mathbf{x} - \mathbf{p}_i) \cdot \mathbf{n}_i, \tag{5}$$

where $\mathbf{p}_i$ is the closest point to $\mathbf{x}$ in the point cloud, and $\mathbf{n}_i$ is the corresponding normal. This method is quite sensitive to noisy data and especially inverted normals. Several methods have been developed to deal with this issue, typically by adjusting the signed distance function to utilize moving least squares, hierarchical, or locally optimal projection methods.

Global smoothness priors are utilized with radial basis functions, indicator functions and volumetric segmentation.

Radial basis functions uses a low-degree polynomial $g : \mathbb{R}^3 \to \mathbb{R}$ and a basis function $\phi : \mathbb{R}^+ \to \mathbb{R}$ to construct the implicit function

$$\Phi(\mathbf{x}) = g(\mathbf{x}) + \sum_j \lambda_j \phi(||\mathbf{x} - \mathbf{q}_j||) \tag{6}$$

where λ_i are found by enforcing $\Phi(\mathbf{p}_i) = 0$. The radial basis functions are typically selected as $\phi(r) = r^2 \log(r), \phi(r) = \exp(-cr^2)$, or $\phi(r) = \sqrt{r^2 + c^2}$ (Carr et al., 2001).

Indicator functions work by labeling which points are in the interior and exterior of the object. These methods work by viewing the input points and normals as samples of the surface normals, and the methods conceptually boil

down to minimize a quadratic equation of the form

$$\arg\min_{\chi} \int ||\nabla\chi(\mathbf{x}) - \mathcal{N}(\mathbf{x})||_2^2 \tag{7}$$

Since the equation is quadratic, the solution is found by applying the gradient to form the Poisson equation

$$\Delta\chi = \nabla \cdot \mathcal{N} \tag{8}$$

Methods of this kind are sensitive to wrongly oriented normals, but methods exist to account for this (Alliez et al., 2007).

Volumetric segmentation works by discretizing the volume, for example through a Delaunay triangulation of the points, and explicitly labeling which cells are inside and outside of the surface. The smoothness can then be utilized to perform this labeling. This method resembles the method in (Litvinov & Lhuillier, 2013), which will be described in detail in section 2.3. The difference is that they utilize a scanner visibility prior to perform the labeling, as described in the next point.

- **Visibility priors**
  There are three types of visibility priors described in (Berger et al., 2016): Scanner visibility, exterior visibility, and parity.

  Scanner visibility is often used to obtain the line of sight of each point in the point cloud. Through the use of space carving, the goal is to mark the regions observed by the scanner as empty, and the regions not observed as not empty. The regions that are not empty can then often be inferred as the interior of the model, while the empty regions are defined as the exterior. The empty regions can e.g. be inferred through a triangulation of the points followed by labeling which points are observed from which poses, defining a line of sight, or ray, through the tetrahedra which can be marked as empty.

  Exterior visibility refers to the visibility information that stems purely from the position of the observing camera, i.e. without other information from the scanner such as which points are observed at which pixels. Methods exploiting this prior seek to find which points are occluded when viewing the point cloud from a given position.

  The *parity prior* is used to determine whether a given point is inside or outside a model with a closed surface. Given a ray from the camera to the point, if the ray intersects the surface an odd number of times, the point is inside the model. Conversely, the point is outside the model if it intersects the surface an even number of times.

- **Other priors**

  The *volume smoothness* prior refers to how fast the volume of a shape varies spatially. The prior is often used to compensate for lacking data. By assuming, for example, that the volume varies smoothly in any direction, one can say something about how the shape looks in the presence of missing data. These methods often use the radius of spheres traveling along the medial axes to measure local thickness, and the local thickness should then be smoothly varying. Often, the medial axes are approximated by curves called skeletons, where the skeleton curve and its distance to the point cloud can for example be parameterized cylindrically (Tagliasacchi et al., 2009).

  One assumption one can make about the object to be reconstructed is that it can be reconstructed with a combination of simple, primitive, geometric shapes. This gives rise to the prior known as *geometric primitives*. It can be used to remove noise from the model, but similarly one may lose fine details in the model as it may be treated in the same way as noise.

  The *global regularity prior* makes use of the fact that many objects, especially man-made objects, have certain recurring elements to the structure. For example, a building may have many similar windows in a certain pattern, and walls in between. This can be used as a prior to fill in more details when data is missing, especially details which are impossible to infer from the data alone. It can also be used to correct for drift and other forms of imperfect scans.

  If large amounts of data on 3D objects are available, one can utilize this in reconstructions where scans are highly incomplete. The reconstructor can for example store a database of objects, and look up the objects when a new imperfect scan is arriving to check for best matches. This is called a *data-driven prior*.

  In the *user-driven* prior the user is involved in the reconstruction adjusting certain aspects of the reconstructed shape, such as smoothness, regularity, and geometric primitives.

2.2.2 Selected Methods

We here briefly present two surface reconstructors that we upon further study decided not to use. The Poisson surface reconstruction was avoided due to reconstruction time complexity. It is usually used as an offline batch surface reconstruction technique that processes all points at the same time. The Ball-Pivoting Algorithm was skipped due to the reduced quality of the mesh, and the advantage Delaunay triangulations have that they can easily utilize visibility priors. As the Delaunay

methods are the basis of our approach, we study them further in section 2.3.

Poisson Surface Reconstruction (Kazhdan et al., 2006) is an example of an implicit function method. In this method, a model, M, is represented as an indicator function, χ, taking values of 1 inside the model and 0 outside the model.

The key idea of the Poisson reconstruction method is that "the oriented point samples can be viewed as samples of the gradient of the model's indicator function". The gradient of the indicator function, $\nabla\chi$, will be zero everywhere, except near the border of the model, ∂M. At the border, it will be unbounded, representing an instantaneous jump in the indicator function from 0 to 1. Therefore, the indicator function has to be convolved with a smoothing filter, $\tilde{F}$, before one can consider its gradient. Denoting the set of sampled points S, they arrive at the following approximation for the gradient of the indicator function, partially utilizing Stokes' theorem:

$$\nabla\chi_F(q) = \nabla(\chi_M * \tilde{F})(q) \approx \sum_{s\in S} |\mathscr{P}_s|\tilde{F}_{s.p}(q)s.\vec{N} \equiv \vec{V}(q) \tag{9}$$

$$\Delta\chi_F \equiv \nabla\cdot\nabla\chi_F = \nabla\cdot\vec{V} \tag{10}$$

In the paper, they arise at the Poisson equation because "$\vec{V}$ is generally not integrable (i.e. it is not curl-free)". Another viewpoint, presented in (Kazhdan & Hoppe, 2013), can be the following:

Given a vector field $\vec{V}$ on the domain M, we want to find the scalar function χ that, when taking the gradient, is closest to $\vec{V}$, i.e.

$$\min_{\chi}\int_M |\nabla\chi - \vec{V}|^2 \implies \Delta\chi = \nabla\cdot\vec{V} \tag{11}$$

where the implication comes from the fact that a minimum should have a derivative of 0. The key idea here is anyways that the Poisson equation can be used to "integrate" a vector field. The resulting surface can be extracted from the estimated indicator function. This is done by selecting all the points in space that have the same value as the average of the indicator function at the sampled points. If γ denotes this value, the surface can be extracted according to eq. (2).

The Ball-Pivoting Algorithm for Surface Reconstruction (Bernardini et al., 1999) is mathematically simpler than Poisson reconstruction. It is a triangulated explicit method, and interpolates the points in the point cloud. The key idea is to define a sphere with radius ρ, and from a seed triangle let the ball pivot over one of the triangle's edges until it hits another point. When it hits another point, the

algorithm connects the point to the mesh by creating a triangle from the edge to the point. This is done for each new edge in each triangle until a mesh unfolds on the sampled surface. If there are points that are not covered by this first mesh, the algorithm selects a new seed and creates another mesh from that seed. See algorithm 1 for pseudocode of the high-level workflow.

This algorithm is computationally cheap, but it is not very accurate. The mesh tends to have a lot of holes, and it might connect the wrong points to go in the mesh. It is not resilient to noise, since it interpolates all, or most of, the points.

Algorithm 1 The Ball-Pivoting Algorithm

```
1: procedure BPA(S, ρ)                ▷ S: point cloud, ρ: radius of pivoting ball
2:     while new seed triangle found do
3:         create seed triangle
4:         mark edges as active
5:         while active edges remaining do
6:             get active edge
7:             ball-pivot around edge
8:             if hit new point then
9:                 create triangle from edge to point
10:                mark new edges as active
11:            else if hit point in mesh and adding point keeps mesh manifold then
12:                create triangle from edge to point
13:                mark new edges as active
14:            else
15:                mark edge as inactive
16:            end if
17:        end while
18:    end while
19:    return Mesh
20: end procedure
```

2.3 Incremental Surface Reconstruction from Point Clouds

2.3.1 Incremental Solid Modeling from Sparse and Omnidirectional Structure-from-Motion Data (Litvinov & Lhuillier, 2013)

(Litvinov & Lhuillier, 2013) introduces a method for reconstructing a manifold surface from point clouds which are incrementally updated with new points and camera poses. The method builds on the more general sculpting methods in 3D Delaunay Triangulation, in which one labels tetrahedra inside and outside the model, and the surface is inferred as the border between these. An important topic through the paper is to preserve the manifold property of the surface, i.e. it cannot cross itself or otherwise act non-physically. The three different manifold tests defined in

section 2.1 are imposed to check for manifoldness.

Information is also used about the camera pose and its relation to the observed 3D points: through the use of ray tracing one obtains information about which tetrahedra are inside and outside of the model. This is a visibility prior, as discussed in section 2.2.1. For example, if a point is visible 'through' a tetrahedron previously thought to be inside the model, one should perhaps now label it as outside.

Sculpting methods in a 3D triangulation. The so-called *sculpting methods* are the foundations for the real-time reconstructors discussed in this report. The name of this group of methods gives an intuitive illustration of the workings of these methods. Given a 3D triangulation of a set of vertices in space, these methods seek to carve away the tetrahedra which are outside the model or surface, and leave behind the matter inside the surface. Informally, in a computer vision context, one can view the vertices as points stemming from a VSLAM system, and rays going from the camera to the surface as beams that evaporate tetrahedra between the camera and detected point. This makes sense, because the space between the camera and the point detected on the surface must naturally be free space. The exception is when one is dealing with transparent objects, but then the camera is better assisted with other sensors for scene reconstruction anyways. When tetrahedra have been marked as "matter" and "free", or similarly, "inside" and "outside" the model, the surface is extracted has the border between these two categories of tetrahedra, i.e. the triangles with one side pointing inside the model and the other outside, given that this triangulated surface mesh satisfies the manifold property.

These sculpting methods can be augmented to work incrementally. The paper suggests six steps to handle each increment in the reconstruction process: *Enclosing, shrinking, point insertion, ray tracing, growing, and post-processing*. Each of these steps will be explained in the following paragraphs, after an introduction to the notation.

We use a notation inspired by (Litvinov & Lhuillier, 2013), but omit most of the time information. The method operates on:

- a 3D Delaunay triangulation T. This is a triangulation of all points added in the model so far.
- a list F of freespace tetrahedra such that $F \subseteq T$. The list is given from the ray-tracing procedure, and the tetrahedra in F may or may not be included in the tetrahedra outside the model, O, defined in the next point. Thus, even though a tetrahedra is actually free space, it may in the reconstruction be a part of the interior of the model, i.e. marked as matter. This is due to the manifold restriction we set on $|\delta O|$.

- a list O such that $O \subseteq F$. δO is the list of triangles that are included in exactly one tetrahedra of O, $|\delta O|$ is the union of triangles in δO, and $|\delta O|$ must be manifold. The tetrahedra in O are the tetrahedra outside the model, and the border of O, $|\delta O|$ is the surface of the reconstructed model. All tetrahedra in O will thus not be visible in the reconstruction, i.e. they are marked as free.
- a list $P_{t'}$ of new Structure-from-Motion points obtained at step t', each new point accompanied by observations from a list of camera locations $\mathbf{c}_t$, $t \leq t'$.

At each step t', the 3D triangulation T is updated with the added points $P_{t'}$; the list F is updated with new visibility information and points, and so is O and δO, while preserving the manifold property on $|\delta O|$.

The methods rely on three assumptions for complexity reasons. The first assumption is that a reconstructed point does not rely on future images. This assumption is naturally valid for VSLAM systems, because future images are not known at the time the point is constructed. Also, if a point needs to be updated, it can simply be deleted and inserted another place. The second assumption is that all rays from camera to point $\mathbf{p}_i\mathbf{c}_t$ are bounded. Detection of far-away objects such as stars would practically violate this condition, and in a VSLAM system, one may have to be careful not to violate this condition. Underwater, however, we do not consider this an issue, as the sight is limited. The third assumption is that the diameters of the tetrahedra are bounded, and this condition is met through the insertion of Steiner points. The Steiner points are a set of points distributed evenly in space such that they form a 3D grid. A simple example is shown in fig. 1. The step length, $l_{Steiner}$, of the Steiner grid, or Steiner points, is defined as the shortest distance between two Steiner points, which should be equal for all Steiner points.

We now proceed to describe the workflow of the six steps as outlined earlier, which is the general workflow of all methods building on this method.

1) Enclosing. The enclosing step ensures that all tetrahedra that are affected by the point insertion will be removed from O, because they will now invalidate the Delaunay property and will have to be recreated. If D is the set of all tetrahedra that are destroyed by the point insertion for this reason, then define an enclosing set E, such that $D \subseteq E \subseteq T$, and stop the shrinking process when $O \cap E = \emptyset$. (Litvinov & Lhuillier, 2013) states that the reason we use such an enclosing set is that it works better in practice. In this method, E is chosen as the set of all tetrahedra contained in a ball of radius $r + l$ centered in the camera location. Here, l is the maximum diameter of the tetrahedra, i.e. $\sqrt{3}l_{Steiner}$, and r is the length of the ray $\mathbf{p}_i\mathbf{c}_t$. This enclosing set may seem needlessly large, and it is in fact one of the aspects improved in (Romanoni & Matteucci, 2018).

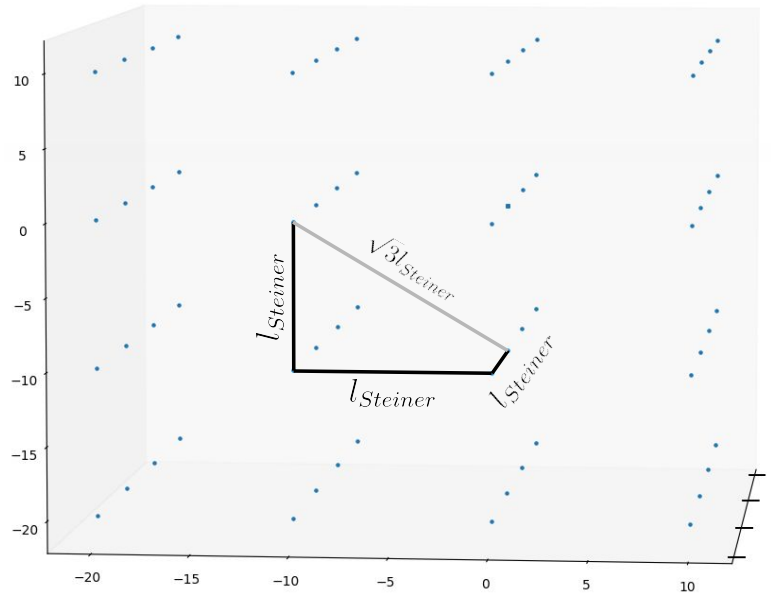


Figure 1: 3D Steiner grid example. The blue dots are the Steiner points. Here, the step length, $l_{Steiner}$, is set to 10m. The longest distance between two Steiner points, which is the bounding diameter of the tetrahedra, is $\sqrt{3}l_{Steiner}$.

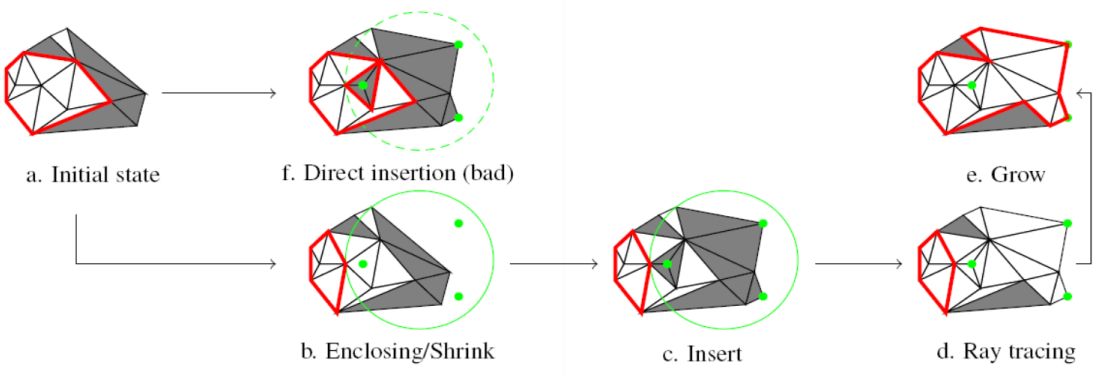


Figure 2: Workflow of the incremental point insertion in 2D, illustrating the important steps in the method (b-e), and why direct insertion is not ideal (f). All triangles are in T, white triangles are in F, and the red line is $|\delta O|$. Copied from (Litvinov & Lhuillier, 2013).

2) Shrinking. The shrinking step aims to iteratively remove tetrahedra from O while $|\delta O|$ remains manifold, until none of the destroyed triangles D are in O, i.e. we stop when $O \cap E = \emptyset$, which implies $O \cap D = \emptyset$, because $D \subseteq E$. The order in which we remove tetrahedra from – or add tetrahedra to – O is not inconsequential, and we would like to minimize the chance that a tetrahedron which is very likely free is removed from O, as O represents the outside of the model. By "very likely free", we mean that it has a large intersection counter as outlined in step 4) ray tracing. To reduce the chance that we remove free-space tetrahedra from O, we greedily remove tetrahedra from O that have a low intersection counter. The tetrahedra may be removed one at a time, each time testing for the manifold property using the subtraction test, as described in section 2.1. They also may be removed several at a time, each time testing for the general test on the entire boundary. The way the shrinking is performed does not actually guarantee $O \cap D = \emptyset$. In this case, the point insertion must be treated specially in order to maintain the Delaunay and manifold property, as described in the next paragraph.

3) Point addition. When we have shrunk $|\delta O|$ while maintaining the manifold property, the next step is to insert the points in the model. Each point added will destroy a set D of tetrahedra that invalidates the Delaunay property. Some elements of D may also be elements of O, as mentioned in the last step. The way the method handles this is to simply not add the point to the model, and remove the point entirely from consideration. The paper states that this is expected to be rare. If none of the elements of D are in O, then we simply insert the point and create new tetrahedra in place of D, which are initially labeled matter. See fig. 2 (c) for a graphical illustration.

4) Ray tracing. The ray tracing of a single point roughly consists of the following two steps: 1. Create a ray from camera location $\mathbf{c}_t$ to point $\mathbf{p}_i$; 2. For each tetrahedron the ray intersects, increment its intersection counter. A naïve way of doing this in an application is to do ray tracing for all points and all views for the current 3D triangulation. This would be very expensive and not suitable for a real-time application. We thus only do ray tracing for two sets of rays. The first is the set of rays belonging to the new points arriving at the current time-step. We trace the rays both from current and previous views to these new points. The second set is the set of rays accumulated to the previous time step that *can* intersect the new tetrahedra. For details on an efficient way to find this set, we refer to (Litvinov & Lhuillier, 2013).

5) Growing. The growing step is done by adding tetrahedra from $F \setminus O$ that has a triangle in δO into O while maintaining the manifold property of $|\delta O|$, and the goal is to make O as large as possible. See fig. 2 (e) for an illustration. The growing step can be seen as an inverse of this shrinking step. We greedily select the tetrahedron with the highest intersection counter to add in O. Furthermore, several tetrahedra can be added at once to allow for genus changes. Also here we refer to the paper for details.

6) Post-processing. The nice effect of enforcing the manifold property on the surface is that it easily allows for mesh refinements, such as smoothing. The method applies a genus refinement procedure and an incremental surface smoothing as explained in (Yu & Lhuillier, 2012a) and (Yu & Lhuillier, 2012b), respectively.

Visual artifacts. We devote some space to the discussion of visual artifacts, i.e. elements of the reconstructed surface that do not resemble the real surface. Visual artifacts were a problem in (Litvinov & Lhuillier, 2013), and the authors discussed and suggested a solution to this problem in (Litvinov & Lhuillier, 2014), which involves a post-processing step. Before delving into solutions to the problem, we study the definition and cause of these artifacts.

(Litvinov & Lhuillier, 2014) defines a visual artifact in the following way: A visual artifact A is a set of tetrahedra $A \subseteq F \setminus O$ and Γ_A is connected. Γ_A is the adjacency graph of the tetrahedra A, where the nodes are the tetrahedra and the edges are the triangles connecting the tetrahedra. This means that A is reconstructed as matter, i.e. $A \cap O = \emptyset$, but it is actually free space, i.e. $A \subseteq F$.

An example of a visual artifact is shown in fig. 3. The white triangles are free space F, the grey triangles are matter, $T \setminus F$. Furthermore, the red line defines the border $|\delta O|$, so the white triangles to the left of the border are the outside triangles, O. The triangle labeled 1, which we denote A_1, is a visual artifact, because it is white, i.e. $A_1 \subseteq F$, but it is not in O, i.e. $A_1 \cap O = \emptyset$. In the reconstruction, this will therefore appear as matter, or occluded space, whereas it in reality is free space. In a typical reconstruction, there might be a lot of visual artifacts, and most of them are not critical. (Litvinov & Lhuillier, 2013) therefore defines the terms *visually critical artifact* and *visually critical edge*: A *visually critical edge* is a line $\mathbf{ab}$ such that $\mathbf{a} \in P, \mathbf{b} \in P$ and $\exists \mathbf{c} \in C$ such that $\widehat{\mathbf{acb}} > \alpha$. A *visually critical artifact* is then a visual artifact with at least one tetrahedron containing a visually critical edge. The parameter α is user-defined, and the paper suggests setting it to $5°$. The idea behind why these particular artifacts are critical, is that the critical edge is so long that it is is deemed unrealistic and severely reduces the reconstruction quality.

The cause of these visual artifacts lies in the nature of the reconstruction al-

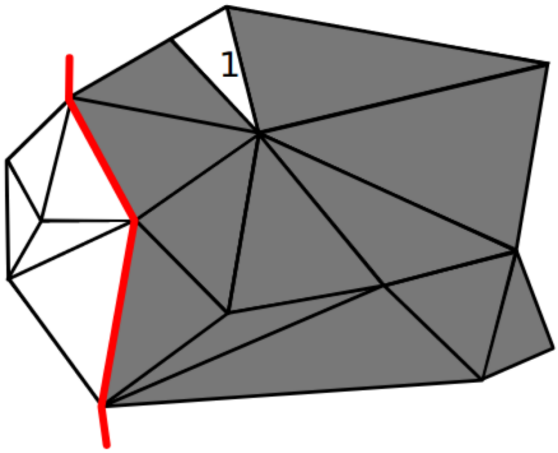


Figure 3: The triangle labeled 1, A_1, illustrates a visual artifact in 2D. All triangles are in T, white triangles are in F, and the red line is $|\delta O|$. In this case, $A_1 \in F$, but $A_1 \notin O$. Figure copied from (Romanoni & Matteucci, 2018).

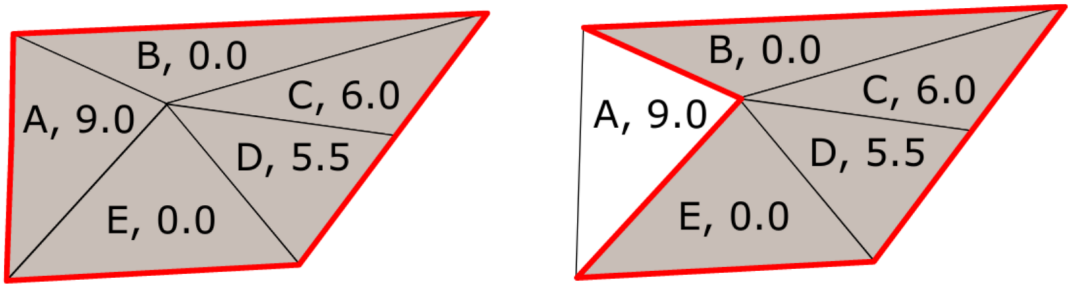


Figure 4: 2D illustration of how a greedy algorithm might not select the "optimal" surface. The greedy algorithm chooses triangle A, while choosing C and D would have been better. All triangles are in T, the white triangle is in O, and the red line is $|\delta O|$, assuming that the object is surrounded by triangles in O. Figure copied from (Romanoni & Matteucci, 2015).

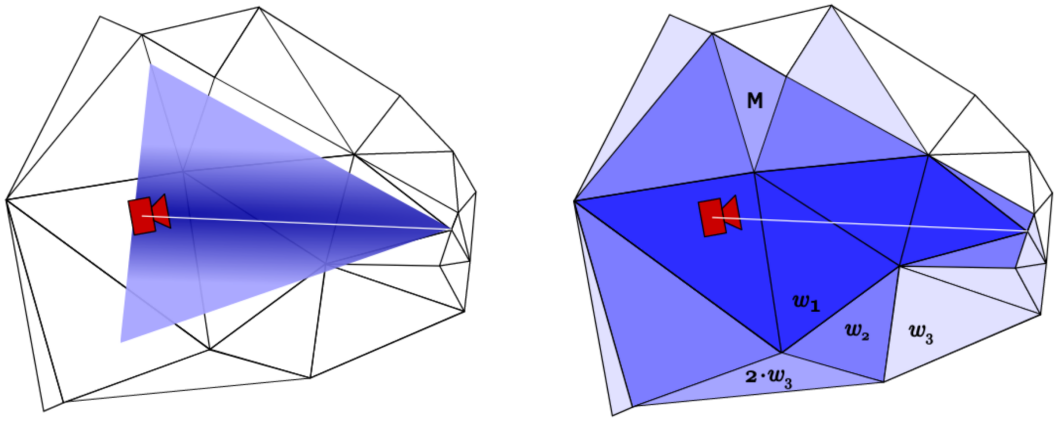
gorithm. By imposing a manifold constraint on the surface, there will be situations where free space tetrahedra cannot be added to O, as it would break the manifold property of $|\delta O|$. The greedy method presented previously does not guarantee an "optimal configuration" of O. An optimal configuration of O could for example be to maximize the total intersection count of the tetrahedra in O. We may get stuck in local extrema with the greedy method. Consider fig. 4, which shows an example of how the method might get stuck in a local extremum. The grey triangles are $T \setminus O$ and the white is O. The greedy algorithm will select triangle A to put in O next because it has the highest intersection counter: 9.0. However, this prevents triangles C and D from being added to O, which has a total incremental counter of 11.5, because it would violate the manifold property of $|\delta O|$. (Litvinov & Lhuillier, 2014) thus suggests removing some tetrahedra from O and adding some other, to "kick the algorithm out of its local extrema", as they describe it. This is a post-processing step, and it improves other post-processing steps that e.g. break the Delaunay property, which is unfortunate for previously discussed reasons. The drawback of (Litvinov & Lhuillier, 2014) is still the time complexity of the operation, preventing it from running in real-time.

2.3.2 Incremental Reconstruction of Urban Environments by Edge-Points Delaunay Triangulation (Romanoni & Matteucci, 2015)

(Romanoni & Matteucci, 2015) improves (Litvinov & Lhuillier, 2013) in two important ways. The first contribution is a new selection of points to triangulate, i.e. a new feature extractor that they call "Edge-points". The second contribution, and the one we will direct most attention to, is a new heuristic for removing visual artifacts in the reconstruction. We begin with a brief discussion of the edge-points before presenting the new heuristic.

Edge-points. The extraction of edge points is inspired by the geometry of a typical urban environment, and the inspiration is made clear already in the name of the paper. The main idea behind this feature is that the edges of the 3D triangulation should lay close to real-world 3D edges, and the claim is that the usual Harris corner detector does not sufficiently give rise to this property. We will not devote much attention to this feature extractor as the scope of our report is to construct a surface from any given VSLAM data.

Inverse cone heuristic. As discussed, visual artifacts in the reconstruction were a problem in (Litvinov & Lhuillier, 2013). (Romanoni & Matteucci, 2015) suggests another solution, which can be used in addition to the method proposed in (Litvinov & Lhuillier, 2014). This solution is based on *preemptively* removing the visual



(a) (b)

Figure 5: The inverse cone heuristic applied to a triangulation in the ideal (continuous) setting (a), and the discrete setting (b). The intensity of the color represents the value of the weights. Figure copied from (Romanoni & Matteucci, 2015).

artifacts, by augmenting the intersection counter in the ray tracing step so that the greedy algorithm chooses a different ordering of the tetrahedra.

As inferred from (Litvinov & Lhuillier, 2014), the long edges are likely to be visually critical edges. We thus want to make sure to correctly label large tetrahedra. Now, another observation made in (Romanoni & Matteucci, 2015) is that the large tetrahedra are usually around the camera trajectory. This is because the camera is often looking at a scene from some distance, and is usually not very close to the observed features, where the tetrahedra are quite small. Thus, in a continuous setting we would want to assign weights to a large space around the front of the camera, and a small space closer to the point. This gives rise to the inverse cone heuristic, which can be viewed as a cone with the tip at the point.

One way of accomplishing this in the discrete setting is by utilizing the mentioned observations. If we add weights to neighboring tetrahedra to the tetrahedra intersected by the ray, this will usually naturally form some kind of inverse cone by the observation that larger tetrahedra are close to the camera and vice versa. This is illustrated in fig. 5. Hence, they have called this intersection weight scheme the *inverse cone heuristic*.

With this heuristic, they were able to reduce the runtime of the artifact removal from 0.43 seconds in (Litvinov & Lhuillier, 2014) to 0.001-0.010 seconds. The results are not exactly comparable, as they were run on different datasets and different hardware, but the paper states that both the datasets and hardware are “similar”.

Nevertheless, the runtime of the entire surface reconstruction algorithm did not reach real-time performance. A real-time solution is presented in section 2.3.3.

2.3.3 Real-Time CPU-Based Large-Scale Three-Dimensional Mesh Reconstruction (Romanoni & Matteucci, 2018)

The method presented in (Romanoni & Matteucci, 2018) further improves (Romanoni & Matteucci, 2015) by optimizing it in order to run in real-time on a single CPU. It accomplishes this through five main improvements:

1. Incremental Steiner Points insertion for incremental map expansion.
2. Shrink only what you need for less shrinkage, and resultingly less growing.
3. Boundary spatial hashing for faster lookup of boundary elements.
4. Next tetrahedra caching to check if the ray we are tracing goes through the same tetrahedra as some previous ray, so that we start looking with these.
5. Ray tracing scheduler to handle moving points in the point cloud, and efficiently perform the ray tracing step when points are moved.

We go through each of these steps in the following paragraphs.

The first improvement is the *incremental Steiner points insertion*. The idea is that the Steiner points initially span a volume around the camera, and when new points are added outside this volume, new Steiner points have to be added in this direction to always have all observed points inside the volume spanned by the Steiner points. This is different from the previous method, where the Steiner points were fixed.

The second improvement is what they call *shrink what you need*. As mentioned in section 2.3.1, the enclosing set was needlessly large, and thus a new way of retrieving the enclosing set is defined. Instead of looking at the maximum distance between the camera and added point, we are now only concerned in the position of the added points relative to the Steiner points. The maximum distance between two Steiner points is, as explained previously, $\sqrt{3}l_{Steiner}$, where $l_{Steiner}$ is the step length of the Steiner points. The ideal enclosing set is then found through the convex hull of the added points expanded by a radius of $\sqrt{3}l_{Steiner}$. If (i, j, k) defines the position of a Steiner grid cell in which a new point, p, appears, we want to enclose all tetrahedra with vertices in cells with location (a, b, c) where

$$
\begin{aligned}
i - 2 \leq a \leq i + 2 \\
j - 2 \leq b \leq j + 2 \\
k - 2 \leq c \leq k + 2,
\end{aligned}
$$

i.e. the cube centered at the grid cell of the new point, with length, width, and height equal to $5l_{Steiner}$.

The third improvement is the use of *boundary spatial hashing*. The motivation behind this improvement is that when new tetrahedra should be added to O, the previous methods had to consider the entire surface of the boundary, which does not scale as the boundary grows. The boundary spatial hashing, hashes the position of a grid cell, (i, j, k) into a vector containing pointers to the boundary tetrahedra intersecting the cell, so that relevant tetrahedra can be quickly retrieved.

The fourth improvement is the *next tetrahedron caching*. This is based on the assumption that many rays generally tend to go in the same direction. Thus, we can utilize information from previous ray tracings when tracing the path of a new ray. The way ray tracing is performed in previous methods is to start at the detected point, loop through the four facets of its incident tetrahedra until reaching the facet the ray intersects, move to the neighboring tetrahedron containing that triangle, and repeat the process for each tetrahedron traversed. In this new method, each tetrahedron traversed saves the index of the next tetrahedron the ray tracing moves moves to. This way, when another ray passes through this tetrahedron, the first triangle we check for a ray passing through, is the facet corresponding to the saved next tetrahedron, possibly saving the time it takes to check the other three facets.

The fifth improvement is the *ray tracing scheduler*. The ray tracing procedure in this method does, in contrast to (Litvinov & Lhuillier, 2013) and (Romanoni & Matteucci, 2015), handle moving points. This is done in three steps: Ray tracing; ray retracing and ray untracing. Ray tracing is done in the same way as before, using the inverse cone heuristic. Ray retracing only traces rays passing through new tetrahedra created through a point insertion or removal. Ray untracing is conceptually the opposite of ray tracing, meaning that weights are decremented and the ray is deleted.

When a point is deleted, the ray retracing step is performed on the affected tetrahedra. Then, ray untracing is performed for all rays from all cameras ending in the removed point. Conversely, when we add a point, ray tracing is performed on all rays ending in the point, and then perform ray retracing on all rays intersecting the conflicting tetrahedra. This way, when a point is moved, we can simply remove the point, triggering the associated ray tracing/untracing, and then add the point at the new location. For efficacy, the operations are scheduled in a particular manner, avoiding duplicate operations. For detailed information, we refer to (Romanoni & Matteucci, 2018).

As a final remark, the method has the ability to update the manifold surface every n-th time step of the VSLAM system, instead of at every single time step. All the same points and information will still be included in the reconstruction. As a default setting in the distributed code, $n = 4$.

3 Method

In this section, we present our method for emulating VSLAM data on the synthetic dataset. The motivation for emulating VSLAM data is to test a suitable surface reconstructor on this data. We considered the reconstructor presented in *Real-Time CPU-Based Large-Scale Three-Dimensional Mesh Reconstruction* (Romanoni & Matteucci, 2018) to be suitable for our purpose of incrementally reconstructing a surface from a dynamic point cloud in real-time, and we started by considering the installation and input data format required by the reconstructor.

There were several issues in the process of figuring out how to correctly run the reconstructor. The installation procedure was not trivial: The project is not maintained anymore, and so the project turned out not to be compatible with newer versions of some of the required packages. Thankfully, we were able to install everything successfully through a thorough inspection of the issues on the repository combined with several increments of trial and error in package installing. Successful installation could be verified through a provided SFM reconstruction example.

The authors had also provided some code on how to run the VSLAM reconstruction, but unfortunately had not provided any data. Nor was the format of the necessary VSLAM data provided. Thus, if we wanted to test the method, we had to inspect the source code to reverse engineer the format. After some discussion, it was decided to spend the necessary amount of time to manually figure out the input data format, by reading the source code of the method. We chose this course of action because we believed the method was suitable for our purpose and could be a good starting point for further work. In addition, it was beneficial that the entire source code of the method was available under the GNU General Public License v3.0.

The resulting data format that was interpolated from the source code is listed in listing 1, and requires some explanation. The data is formatted as a JSON. All numerical values presented are for illustrative purposes. The "`slam_data_version`" is there because the source code checks that it is correct in some sense, i.e. that it is version 0.2 or 0.3. We put it in just to make sure it passed the check.

The "`root_path`" refers to the root path for the data. We put the data so that the root path was as given in the listing. The "`intrinsics`" refers to the camera matrix. For our data, all pictures have the same intrinsics as they are taken with the same simulated camera, so we only need one entry in this list. The code adds the ability to have several different intrinsic matrices. However, the code does not allow for non-square pixels or a skew parameter. From the source code, we so far do not see the purpose of having the camera intrinsics in so many levels, i.e. "`value`", "`ptr_wrapper`", "`data`".

```
{
    "slam_data_version": "0.3",
    "root_path": "./",
    "intrinsics": [
        {
            "intrinsicId": 0,
            "value": {
                "ptr_wrapper": {
                    "data": {
                        "focal_length": 25,
                        "principal_point": [959.5, 529.5]
                    }
                }
            }
        }
    ],
    "views": [
        {
            "viewId": 1,
            "local_path": "",
            "filename": frame1,
            "width": 1920,
            "height": 1080,
            "id_intrinsic": 0,
            "extrinsic": {
                "rotation" : R1,
                "center": t1
            },
            "observations": [
                {
                    "pointId": 1,
                    "X": [0.1234, 2.44, 3.435]
                },
                {
                    "pointId": 2,
                    "X": [2.234, 3.564, 7.54]
                },
                ...
            ]
        },
        ...
    ]
}
```

Listing 1: Input VSLAM data format

Most of the data is stored in the "`views`" element of the JSON. The element contains a list with one entry for each view in the video stream. The views are selected by the VSLAM method creating the data. Each view is assigned an ID and its picture's corresponding filename. Also, the width and height in pixels must be given. Furthermore, one must link the view to its corresponding intrinsics through the "`id_intrinsic`" parameter. The extrinsics for the view must also be assigned. In listing 1, `R1` is a 3×3 rotation matrix relating the camera frame to the world frame, and `t1` is the translation from the camera frame to the world frame. The observations refer to which points are observed from the current view. The points are given IDs, so that two matched points are registered as the same points. The position must also be given, even if the point ID and position are given from an earlier view. This is because the method allows for updating the position of the points. The ellipsis symbol in the attached listing means that the data pattern repeats. For the observed points, it repeats for the number of observed points at the current view, and for the views it repeats for the number of views in the dataset.

When we had successfully installed the method and resolved the required input format, we wanted to test the method on a custom dataset of a synthetic underwater dataset.

3.1 The Car-Cube Dataset

The dataset used for testing the incremental surface reconstructor will be referred to as the Car-Cube dataset. The dataset contains an image sequence of a synthetic underwater scene containing a car and a cube, accompanied by the ground truth depth images and poses. This let us test the method isolated from errors in these variables. See fig. 6 for some sample images and corresponding depth images. The dataset is a precursor to the VAROS dataset (Zwilgmeyer et al., 2021). We used the Car-Cube dataset instead of the VAROS dataset simply because of issues relating to the size of the VAROS dataset and the hardware at hand. The trajectory of the camera in these datasets is simulated with a physics-based model, and is realistic in terms of an AUV vehicle. However, the lighting conditions in the Car-Cube dataset are unrealistically good. This will perhaps give a better performance of feature detectors than with more non-uniform lighting. At the same time, it lets us test the surface reconstruction method in isolation from poor feature detection. Furthermore, the dataset let us use points not detected by a regular feature detector, such as untextured areas, as we could simply extract the depth information for each pixel.

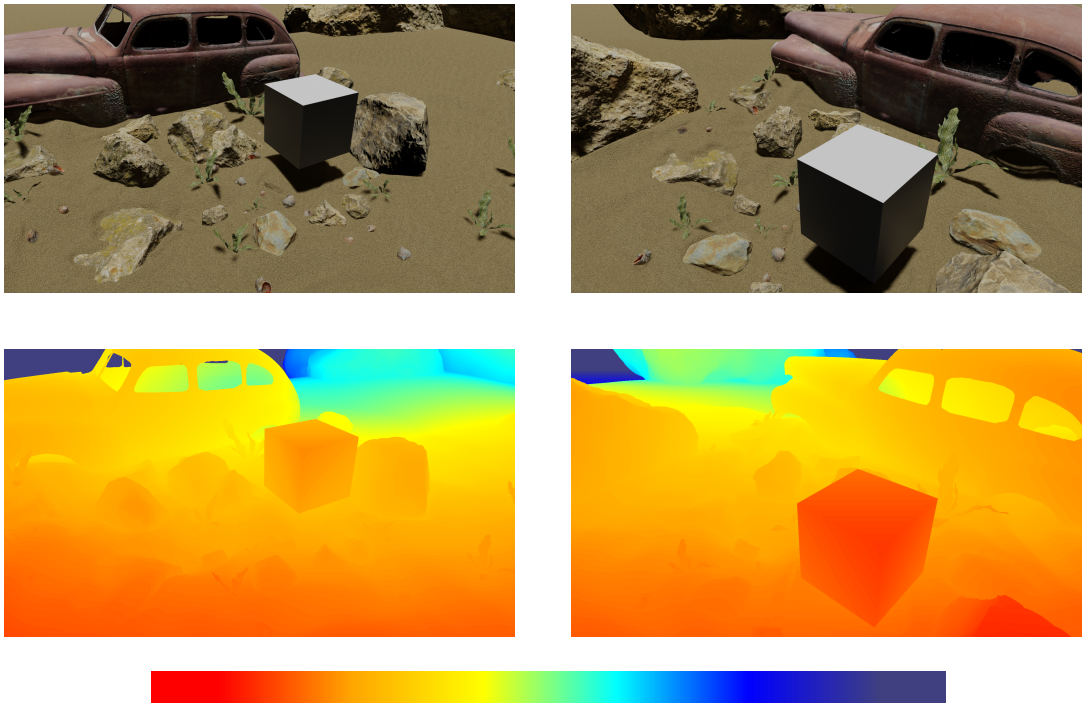


Figure 6: Sample color images and corresponding depth maps. Red is near and violet is far.

3.2 Point Projection

The ground truth depth in the Car-Cube dataset is truncated at 10 meters, so all depths greater than 10 meters have the same value in the depth map. We used the NumPy library (Harris et al., 2020) for implementing points and matrices as arrays. It is imported with the name `np`. We used the OpenCV library (Bradski, 2000) for reading images and performing feature extraction and matching.

The ground truth pose given in the dataset was utilized for the emulated VSLAM. The pose was given as a 6-dimensional vector, where the first three elements give the position, and the following three give the orientation. The ground truth orientation is given in Euler angles. To get the rotation matrix from the data, we used the Scipy library (Virtanen et al., 2020), as such:

```
def get_rotation_matrix(euler_angles)
    r = scipy.spatial.transform.Rotation.from_euler('xyz', euler_angles)
    return r
```

The full transformation could then be reconstructed from the rotation matrix, r, and the translation, t.

```
def compose_T(r, t):
    T = np.eye(4)
    T[:3, :3] = r.as_matrix()
    T[:3, 3] = t
    return T
```

The inverse T matrix was calculated using the same Scipy library:

```
def invert_T(T0):
    T = T0.copy()
    r = Rotation.from_matrix(T[:3, :3])
    t = T[:3, 3]
    r_inv = r.inv()
    T_inv = np.eye(4)
    T_inv[:3, :3] = r_inv.as_matrix()
    T_inv[:3, 3] = - r_inv.apply(t)
    return T_inv
```

For the projection, we assume an ideal camera, with no distortion or skewing. The intrinsic camera matrix then becomes quite simple, i.e.

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

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

$$K = \begin{bmatrix} 1263 & 0 & 959.5 \\ 0 & 1263 & 539.5 \\ 0 & 0 & 1 \end{bmatrix}$$

or in meters,

$$K = \begin{bmatrix} 0.02500 & 0 & 0.01899 \\ 0 & 0.02500 & 0.01068 \\ 0 & 0 & 1 \end{bmatrix}$$

It is crucial to always use correct measurement units for the coordinates, and using wrong units was the main source of error throughout the project. Two logical measurement units were considered in our method: meters and pixels, and we allow for the use of both. Having square pixels of a fixed size, the conversion is simple: One can simply multiply the coordinates given in meters by the number of pixels per meter to obtain the coordinates given in pixels. Conversely, one can divide the pixel coordinates by the pixels per meter to obtain coordinates in meters.

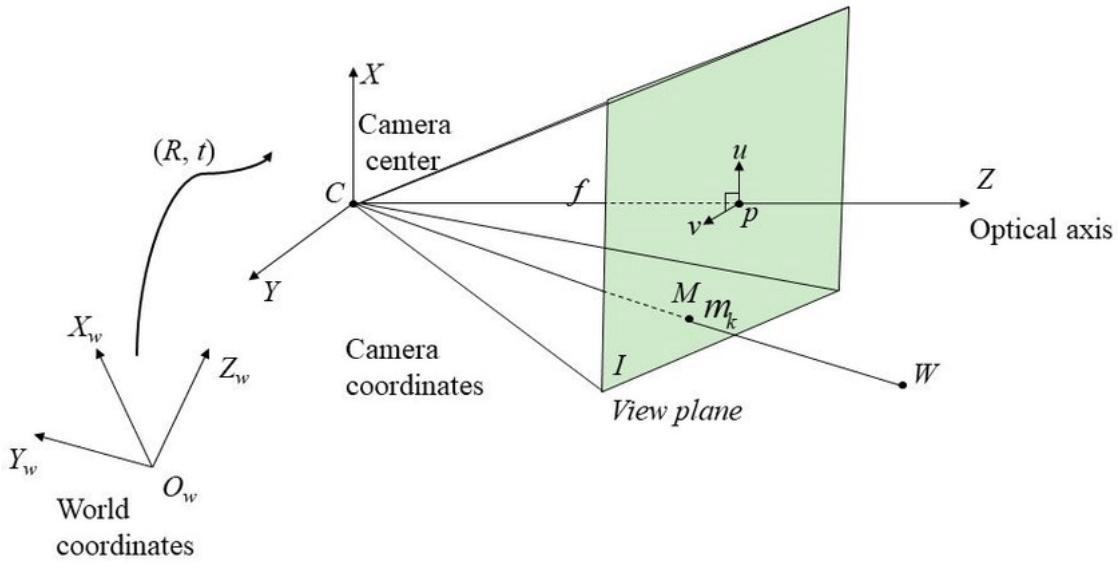


Figure 7: Pinhole camera model. Point W is projected onto image plane, I, to obtain image point m_k, or vice versa. Camera coordinate system is related to world coordinate system by (R, t). Copied from (Huang & Huang, 2020).

Using the pinhole camera model, we could utilize the ground truth depth information to project the detected point at coordinates (u, v) into space. See fig. 7 for an illustration of the pinhole camera model. The projection can be performed using the method of similar triangles, as illustrated in fig. 8. In the illustration, f, y, and Z are known, and we want to find Y. Y can be found as

$$\begin{aligned} \frac{Y}{Z} &= \frac{y}{f} \\ Y &= \frac{y}{f} Z, \end{aligned} \tag{12}$$

with $y = u - c_x$ or $y = v - c_y$. In our method we implemented it in the following function, where K is the intrinsic camera matrix:

```
def project_2d_to_3d(u, v, z, K):
    c_x = K[0, 2]
    c_y = K[1, 2]
    f_x = K[0, 0]
    f_y = K[1, 1]

    x = ((u - c_x) / f_x) * z
    y = ((v - c_y) / f_y) * z
    return [x, y, z]
```

The coordinates of the points are then given in the camera frame of reference,

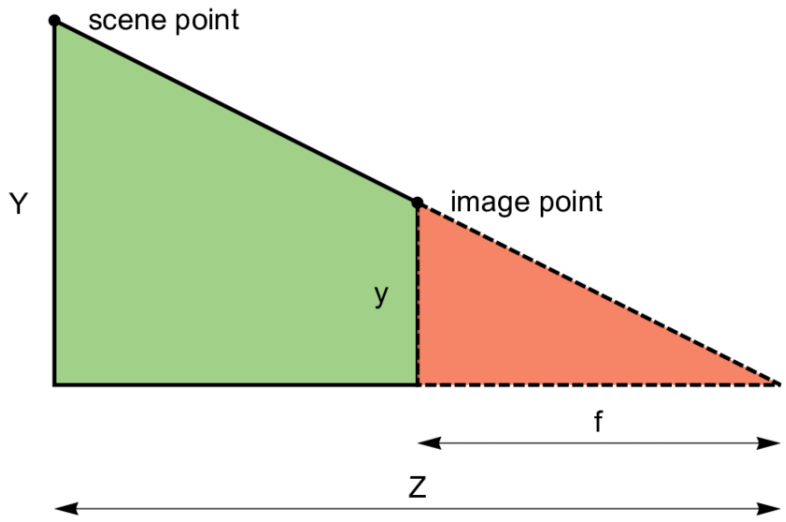


Figure 8: The figure shows a crossection of space, where the Y coordinate of the scene point can be found using the method of similar triangles. Copied from (Szpak, 2013).

centered at the center of the camera with x and y coordinate basis vectors spanning a plane parallel to the image plane and the z vector perpendicular to this plane, i.e. pointing in the direction the camera is viewing. We can use a transformation matrix to transform the coordinates of the points from the camera reference frame to the world reference frame, with the use of the ground truth pose. The reference frame must then be rotated and translated into the world frame, which can be done with a transformation matrix, $\mathbf{T}$, so we simply apply $\tilde{X}_{world} = \mathbf{T}\tilde{X}_{camera}$ to each point, where the tildes imply that the points are given in homogeneous coordinates. In Python, this can be implemented as:

```
def transform_3d_points(T, X):
    return T @ X
```

The question now remained how we could obtain appropriate sample points on the image plane. Two main approaches were considered and tried out: Random sampling and feature detection. The first approach has the advantage that, over many frames, one would get a quite representative sampling of the depths in the images and thus of the scene. The implementation of random sampling is presented in section 3.3. The feature detection method could suffer from a concentration of feature points around strong visual appearances in the image, and ignore points of little visual information. However, it might be more realistic in terms of true VSLAM data. Feature detection should then also imply feature matching to produce realistic VSLAM data. The feature detection and matching is presented in section 3.4.

3.3 Random Sampling

The random sampling of points was implemented by producing random integers in the range of the image width and height, as illustrated in the function `sample_keypoints`:

```
def sample_keypoints(num_kpts, width=1920, height=1080):
    us = np.random.randint(width, size=num_kpts)
    vs = np.random.randint(height, size=num_kpts)
    uvs = np.vstack((us, vs))
    return uvs
```

This function gives a set of points in pixel coordinates. Now, our depth map gives a depth value for each pixel. Thus, we can project each point into space and transform it into world coordinates, and we obtain a point cloud. The projection can be performed with the function `project_2d_to_3d`, as described in section 3.2.

When random sampling is used, one may have the side effect that sampled image points in different views are projected very closely in space, and that they would otherwise be detected as a feature match. This effect could be eliminated, and we did indeed implement a naïve approach to solve this problem by searching through all points and checking if they are near the projected point. The search was very computationally expensive, and we did not spend much time on optimizing it. Therefore it was decided to remove the search altogether, and rather accept the possible reduction of the data quality. The data produced with this random sampling approach did thus not include matching of points across views, which led to quite artificial data in that regard. This would also keep adding points to the system at a high rate, even when standing still, deteriorating the performance of the real-time reconstructor.

3.4 ORB Feature Detection and Matching

We decided to use ORB for feature extraction and description, as it is the feature detector used in the state-of-the-art ORB-SLAM3 method (Campos et al., 2020) and it gives a descriptor that can be used for matching. For matching, we use a FLANN-based matcher (Muja & Lowe, 2009), for its efficiency. As new points are detected, we for each point check if it is in the model. If so, we simply add that point to the list of observed points from this view, and proceed. For debugging purposes, we also added an option of marking the point on the image in order to compare the keypoint coordinates from the feature detection with the coordinates resulting from projecting the 3D point onto the image plane.

If the point is not in the model, i.e. it did not produce a good match with any other point, we add the point to the model with the described method of projection

and also add the point to the list of observed points from this view.

Each unique point is assigned a unique integer, which acts as its identifier (ID). From this, we can easily describe which points are viewed from which frames, which is a part of the emulated VSLAM data, and crucial for the reconstruction process.

The FLANN-matcher is initialized as with the following parameters, inspired by the OpenCV tutorial on feature matching (OpenCV, 2021):

```
index_params = dict(algorithm=6,
                    table_number=6,
                    key_size=12,
                    multi_probe_level=1)
search_params = dict(checks=50)

flann = cv.FlannBasedMatcher(index_params, search_params)
```

Now, if we define `des_i` as the descriptors in the current image, and `previous_des` as descriptors in the previous images we can do feature matching in the following way:

```
matches = flann.knnMatch(des_i, previous_des, k=2)
```

and filter out the poor matches by performing Lowe's ratio test (Lowe, 2004). We use a distance of 0.75 in the ratio test, as follows:

```
good = []
for m, n in matches:
    if m.distance < 0.75 * n.distance:
        good.append(m)
```

Then we add the ID of the matched point to the list of observed point in the current view, and note that it is previously detected. The ID corresponds to the index of the matched point, so we can add it as such:

```
for match in good:
    old_point = match.trainIdx
    new_point = match.queryIdx
    observations[current_view].append(old_point)
    prev_detected[new_point] = old_point
```

Using ORB features also included the possibility of false matches. However, in our code, this will not lead to outliers. The reason is that we do not triangulate the

points. Rather, points are projected using the ground truth depth map as already outlined.

Nevertheless, to force the selected ORB features to comply with an outlier rejection method used by a triangulation, we choose to exclude points with false matches from the model. To do this, we project the 3D points, `X`, to the image frame in pixel coordinates and compare with the detected features. This is done with the following function, where the point coordinates given to the function are not homogeneous. The function allows for projecting one or several points at once:

```
def project_3d_to_2d(K, X, pose):
    num_points = X.shape[1]
    ones = np.full(num_points, 1)
    X1 = np.append(X, [ones], axis=0)
    uvw = K @ (pose @ X1)[:3, :]
    uvw /= uvw[2, :]
    return uvw[:2, :]
```

Then, one can compare a detected point in the current image, `(u, v)`, with its corresponding point in the model projected into the current frame, `(u_hat, v_hat)`, and mark the point as an outlier if it is above some defined threshold. If the units are converted from meters into pixels, we considered a suitable threshold to be 5 pixels:

```
if abs(u-u_hat) > 5 or abs(v-v_hat) > 5:
    pm.mark_as_outlier(train_idx)
```

The points are marked as outliers in an instance of the `PointManager` class, `pm`. This is where points are added to, retrieved from and edited. For the implementation of this class, we refer to appendix A.

3.5 Combination of Random Sampling and ORB

As a compromise to the problems discussed with purely using ORB-features or random sampling, we lastly decided to implement a combination of the two approaches, using some randomly sampled points and some detected features in each frame. In our code, the use of random samples can be toggled on and off, and one can specify the number of keypoints and frames to run:

```
USE_RANDOM_KPTS = True
NUM_RANDOM_KPTS = 20
NUM_FRAMES = 500
```

When we use the random sampling and ORB in combination, we perform feature matching on the ORB features, while the randomly sampled points are projected as it otherwise would, without any feature matching or "duplicate removal". With this combination, we will therefore continue to add points to the model for each frame, even when viewing the scene from a static viewpoint, just as we would using only random sampling.

4 Results

Unless otherwise mentioned, the Steiner point step length is set to 10m; the inverse cone heuristic weights are set to setup 1, as specified in table 5; and the scene is reconstructed using random sampling only. Generally, the triangles surrounding the reconstructed scene arise from the Steiner points surrounding the scene and are not visual artifacts.

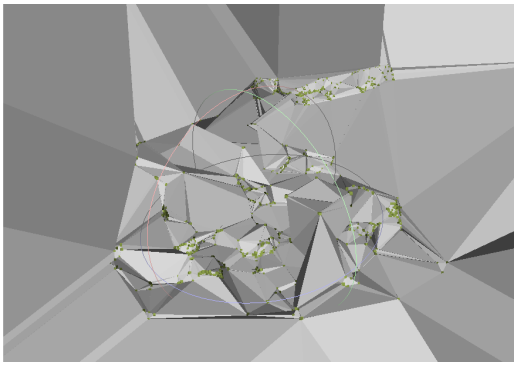
4.1 Point Selection

We begin by comparing the different keypoint selections: ORB features only, random sampling only, and a combination of ORB features and random sampling. The result is shown in fig. 9. The green dots show the keypoints in space. Recall that even if we use ORB features, we use the ground truth depth for 3D localization of the point. A color image from the scene is given in fig. 9d for comparison. All reconstructions are based on 30 frames of video.

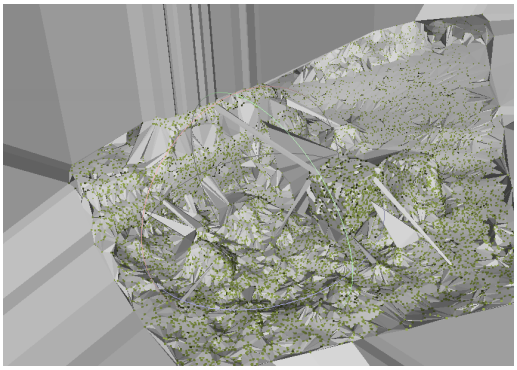
The ORB-only feature extraction extracts 1000 features per frame. Due to feature matching and depth truncation, the total number of points in the model is 11490. The result is shown in fig. 9a.

The random sampling is done with 400 keypoints per frame over 30 frames. However, we do not project points where the depth is truncated. The resulting number of points in the model is then 10746. The result is shown in fig. 9b.

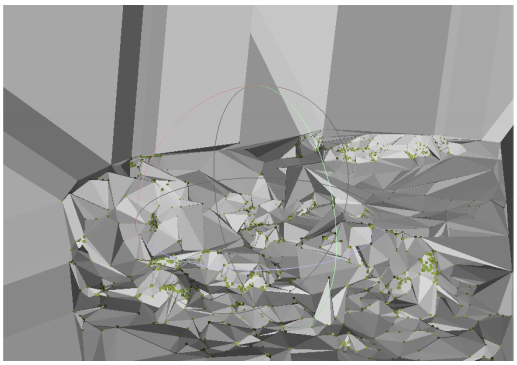
The combination of ORB features and random sampling uses 1000 ORB features per frame and 20 random points per frame. The resulting number of points is 12011, 521 more than from ORB alone. The result is shown in fig. 9c.



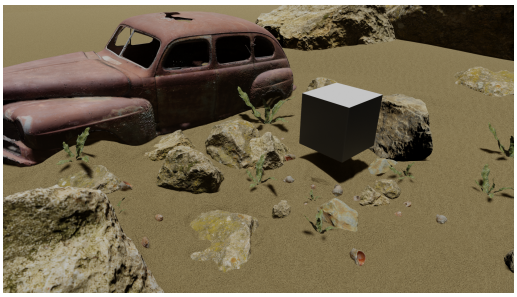
(a) ORB features only



(b) Random sampling only



(c) ORB and random sampling

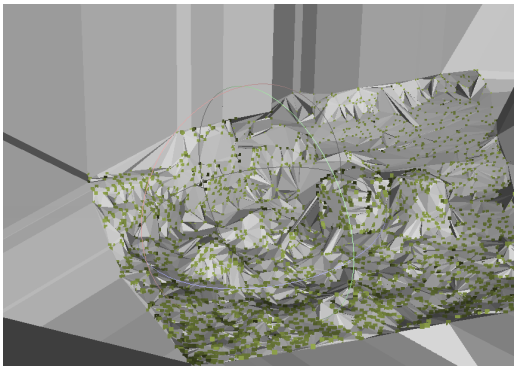


(d) Color image of scene

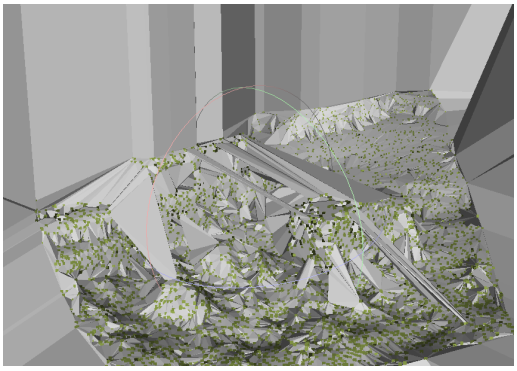
Figure 9: Comparison between the different image points used for reconstruction, with color image for reference. The green dots are the points in the model.

4.2 Reconstruction Progression

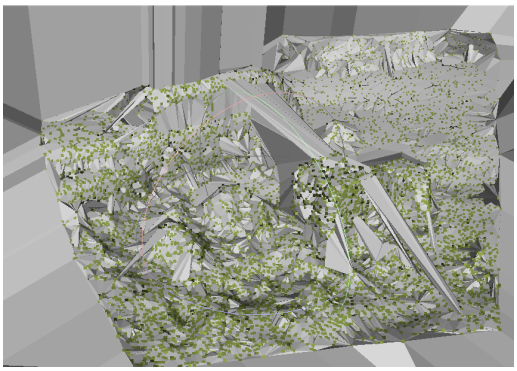
We include some increments in the reconstruction process. The points are extracted from random sampling only. We sample 400 keypoints per frame over 30 frames. Figure 10 shows the reconstruction for some selected frames. Figures 10a to 10d shows the reconstruction after 8, 16, 24 and 30 frames, respectively. The green points are the points in the model. The frames used in the reconstruction are the same as in section 4.1.



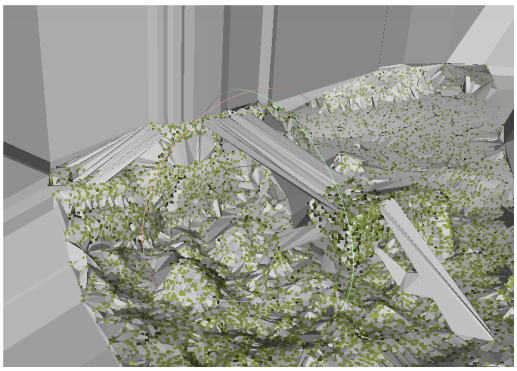
(a) After 8 frames



(b) After 16 frames



(c) After 24 frames



(d) After 30 frames

Figure 10: Reconstruction progression as more and more frames are incrementally added to the model.

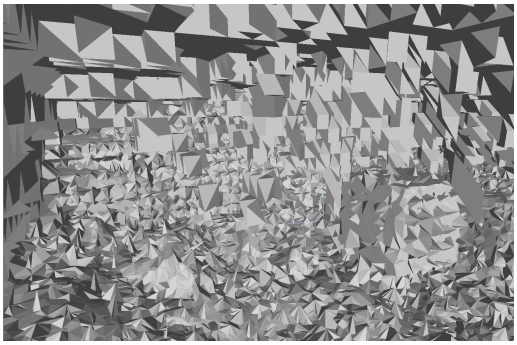
4.3 Steiner Points

In this section, we test the influence of the density of Steiner points in the reconstruction. The density of Steiner points is defined as the step length between each Steiner point. Recall that the step length between Steiner points is the shortest distance between two Steiner points, which is equal for all points as they are distributed evenly in a grid in space.

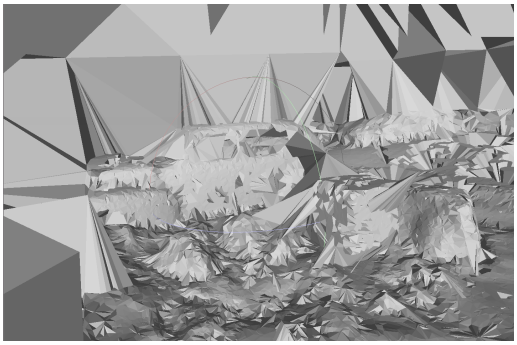
The visual results are shown in fig. 11. In this test, we again use random sampling only, 400 keypoints per frame and 30 frames. The resulting mesh is the result after all frames have been processed. The results on running time are summarized in table 1.

Steiner point step length (m)	0.1	1	10	100
Reconstruction time (s)	12.8844	4.81415	4.21665	4.15063
Reconstruction time per frame (s)	0.42948	0.16047	0.14056	0.13835

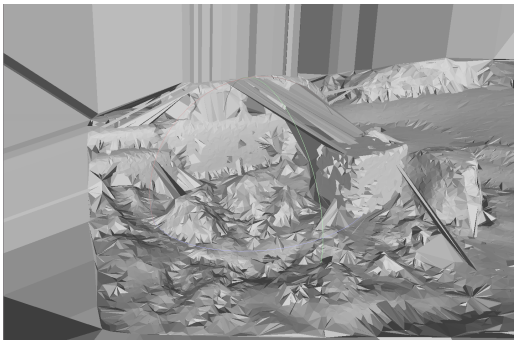
Table 1: Step length of Steiner points influence on running time



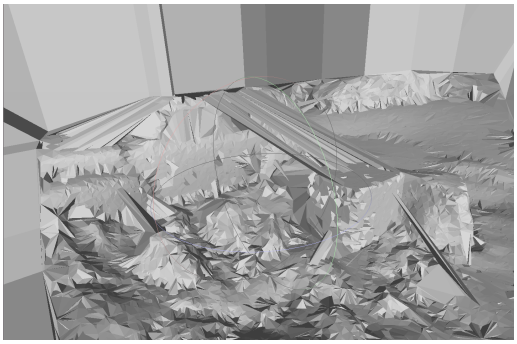
(a) Step length = 0.1m



(b) Step length = 1.0m



(c) Step length = 10m



(d) Step length = 100m

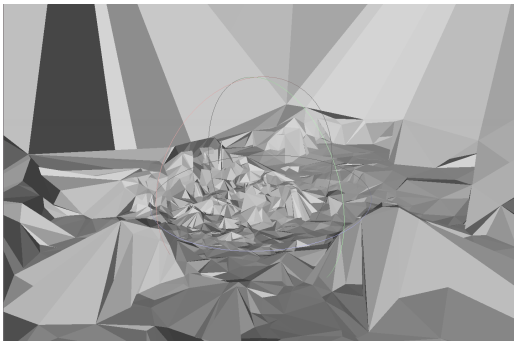
Figure 11: Influence of the step length of the Steiner grid on the reconstruction quality.

4.4 Number of Keypoints

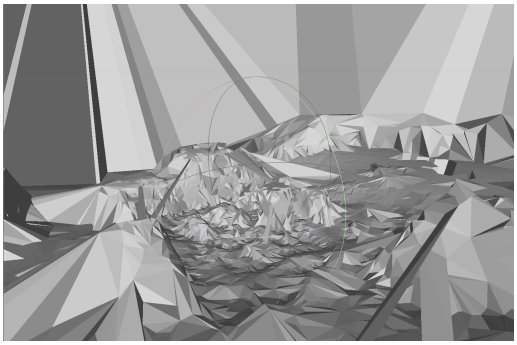
Here we present the influence of the number of keypoints in the visual reconstruction as well as the reconstruction time. All surfaces presented are constructed from the same 800 frames. The visual results for 2, 8, 15, and 50 keypoints per frame are presented in figs. 12a to 12d, respectively. The results on running time are presented in table 2

Number of points per frame	2	8	15	50
Reconstruction time (s)	3.71947	22.5017	51.1882	274.777
Reconstruction time per frame (s)	0.004649	0.028127	0.063985	0.34347

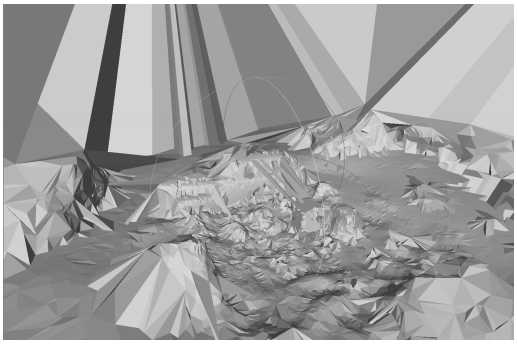
Table 2: Number of sampled points per frame influence on running time



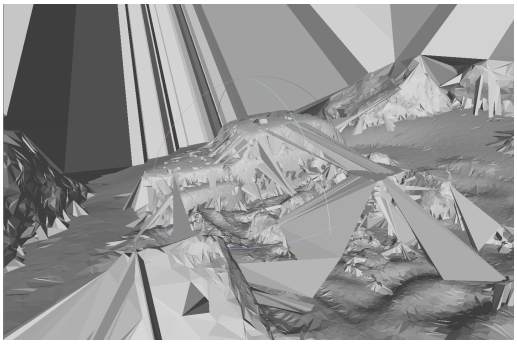
(a) 2 points per frame



(b) 8 points per frame



(c) 15 points per frame



(d) 50 points per frame

Figure 12: Influence of number of keypoints sampled at each frame on the quality of the reconstruction.

4.5 Number of Frames

Here we present the number of frames' influence on running time. We randomly sample 50 points per frame. The results are presented in table 3.

Number of frames	100	200	400	800
Reconstruction time (s)	3.21262	12.6656	60.2912	274.777
Reconstruction time per frame (s)	0.0321262	0.063328	0.150728	0.343471

Table 3: Number of frames influence on running time using random sampling only

An important result can be summarized with the following example. For the first 100 frames, the reconstruction time is about 0.03 seconds per frame. For the last 400 frames, i.e. from frame 401 through 800, the reconstruction time per frame is approximately $\frac{275-60}{400}s \approx 0.54s$, which is about 18 times more than for the first 100 frames.

Exchanging random sampling with ORB feature detection, we obtain the results presented in table 4. We detect 1000 features per frame.

Number of frames	100	200	400	800
Reconstruction time (s)	46.7501	190.371	821.87	3380.83
Reconstruction time per frame (s)	0.46750	0.95186	2.054675	4.22603

Table 4: Number of frames influence on running time using ORB features only

4.6 Inverse Cone Heuristic

In this section, we present the effect of the inverse cone heuristic. We only present the visual effects in a qualitative manner. The reconstruction time does not differ significantly between the different weights.

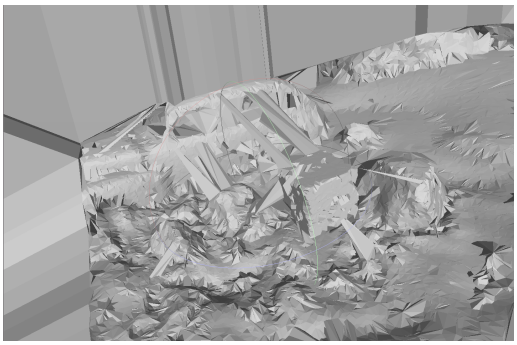
We refer to weights of the intersected tetrahedra as w_1, weights of their neighbors as w_2, and weight of the neighbors' neighbors as w_3.

The different weight setups are presented in table 5. Setup 4 is simply to not use the inverse cone heuristic, in the same manner as previous methods, such as in (Litvinov & Lhuillier, 2013). The results for setup 1, 2, 3 and 4 are presented in figs. 13a to 13d respectively. The reconstructed is run with 400 randomly sampled points per frame over 30 frames.

	Weights		
	w_1	w_2	w_3
Setup 1	10	2.0	1.0
Setup 2	1.0	0.8	0.2
Setup 3	2.0	0.8	0.2
Setup 4	1.0	-	-

Table 5: Different weight setups for the inverse cone heuristic

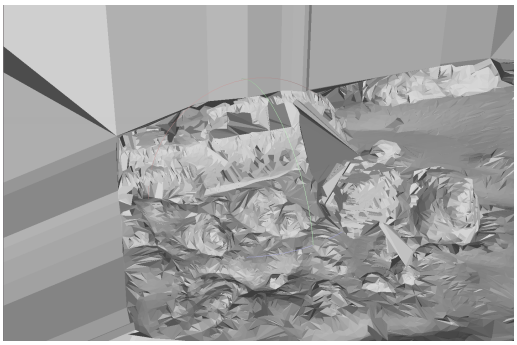
Since the random sampling of points can affect the reconstruction, we include results from four randomized models with the same parameters. These are presented in fig. 14. The number of frames equals 30 and the number of points per frame equals 400. The frames are the same in all reconstructions. The inverse cone heuristic for the reconstruction uses weight setup 1. Inspection reveals that the tetrahedra in front of the cube in fig. 14d are created somewhere in the last two frames of the reconstruction. Updating the manifold more often than every 4 frames removed the artifact, but introduced other artifacts.



(a) Weight setup 1

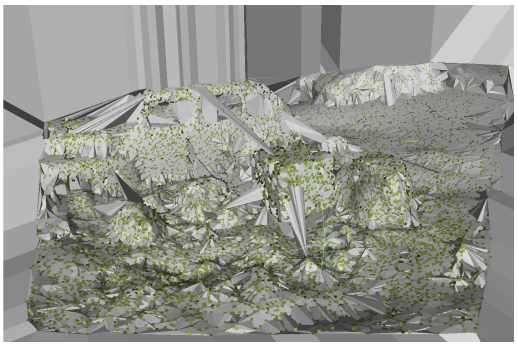
(b) Weight setup 2

(c) Weight setup 3

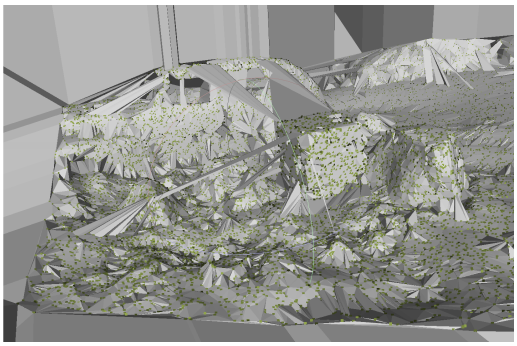


(d) Weight setup 4

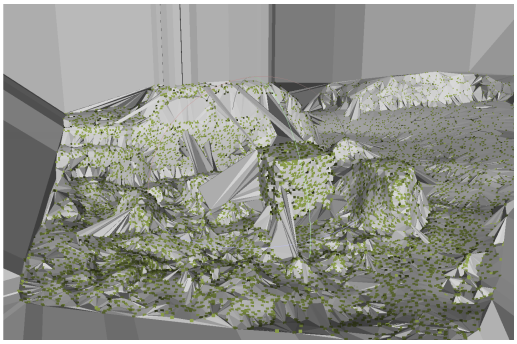
Figure 13: Influence of different weights used in the inverse cone heuristic on the reconstruction quality and artifact removal.



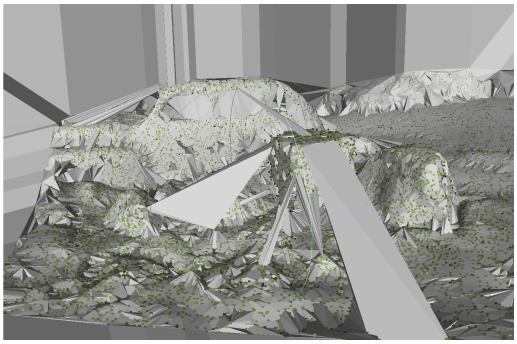
(a) Random sampling 1



(b) Random sampling 2



(c) Random sampling 3



(d) Random sampling 4

Figure 14: Four different models created of the same scene, using the same frames and the same parameters, but with randomly sampled points from each frame. This illustrates the effect of randomness in our sampling method on the reconstruction.

5 Discussion

5.1 Point Selection

We see that using ORB features alone do not give more than the very general structure of the scene, even though 1000 features were extracted at each frame over 30 frames. One issue with the ORB feature extraction on the simulated scene that we noticed quite early is that it tended to find a lot of features concentrated around small areas. Meanwhile, large portions of the images did not have any features extracted.

In addition, the feature matching of the ORB features was not really robust. The matching procedure produced a lot of false negative matches. This resulted in multiple points at approximately the same location, that should really have been one point. One reason for this artifact might be that we did not impose the necessity of a point being visible from multiple frames, as is usually the case in a VSLAM system. This might have filtered some of the false negative matched points away, in that we would not project them into space.

One can vaguely see these effects in fig. 9a, where there are dense clusters of green dots, which correspond to features. In contrast, see fig. 9b, where the green dots are quite evenly spread out. Interestingly, the number of points in the two models are almost the same: 11490 using only ORB features and 10746 using random samples only. Actually, the model with only random samples has fewer points, but gives far more visually interpretable results. When points are more evenly selected over multiple frames, this effect is strengthened further. For this reason, we used random sampling for testing other aspects of the reconstruction process. Nevertheless, we do see quite a few visual artifacts in the reconstruction. This will be elaborated upon further in section 5.6.

5.2 Reconstruction Progression

From the reconstruction process, we can see the car and the cube appear more clearly as more frames are processed. This is naturally the behavior we expect. We also see that the density of green points gets higher and higher, as more points are added. Roughly the same amount of points are added to the reconstruction for each frame when using the random sampling method as we do here.

However, a surprising result is that visual artifacts appear to be introduced. For example, there are a number of visual artifacts created between frames 8 and 16. The ray tracing should prevent such artifacts from occurring, especially by introducing the inverse cone heuristic. As this was quite surprising to us, we decided to test different weights on the inverse cone heuristic, as well as without the heuristic, using

only ray tracing. This was tested in section 4.6, and will be discussed in section 5.6.

5.3 Steiner Points

From fig. 11a, where the step length of the Steiner points is set to 10cm, we see that the scene is almost unrecognizable. The first explanation is how the result is presented, with only a single image of the scene without texture. A video moving the resulting mesh around the view would probably give more information of the geometry in the model, as the shading of the small triangles gives a lot of visual clutter in a still image. The report format does, however, restrict us to the use of images for presentation of the results.

Besides the still image representation, the model is still quite poorly represented. There are a lot of tetrahedra floating in arbitrary positions in space, and the car is not easily recognizable, no matter how much we studied the model. The reason for the floating tetrahedra might be that there are no, or few, rays passing through these tetrahedra. As the Steiner points get denser, and resultingly the tetrahedra smaller, the rays would have to pass through more tetrahedra to mark all of them as free space. However, depending on the number of points observed, there will not be enough points distributed evenly in space to pass rays through all tetrahedra. We also believe this effect would be further strengthened if using the ORB feature extraction only. As discussed, these features tend to cluster in small areas, and this would further reduce the number of tetrahedra marked as free space.

Qualitatively, there is a big shift in the quality of the reconstruction when moving from 0.1m step length to 1.0m step length. However, the difference is not so large when moving from 1m to 10m or 100m. This may be because when the step length of the Steiner points is this large, the ray tracing does not suffer from the problem discussed in the previous paragraph. The quality of the reconstruction is then mainly defined by the points in the model, as they are much denser than the Steiner points.

We also tested the effect of the size of the Steiner points on the running time. From table 1, we see that there is a significant drop when increasing from 0.1m to 1.0m. Further, there is a slight drop when increasing from 1.0m to 10m. However, the drop is not that significant when increasing the step length further. The running time also slightly varies from run to run, and sometimes, having 100m step length was faster than 10m.

There are two conflicting mechanisms relating the size of the Steiner step length to running time: ray tracing and the enclosing set. As the Steiner step length is larger, the rays have to pass through fewer tetrahedra resulting in faster computation. This is especially the case when the robot is exploring new territory, where

the Steiner points are largely defining the size of the tetrahedra. However, when the step length increases, so does the number of tetrahedra in the enclosing set. Recall that the size of the enclosing set in (Romanoni & Matteucci, 2018) is proportional to $\sqrt{3}l_{Steiner}$. Thus, increasing $l_{Steiner}$ increases the enclosing set, resulting in more shrinking, and thus more growing, both of which result in more computation.

5.4 Number of Keypoints

We see from table 2 that the reconstruction time increases a little more than linearly with the number of new points per frame. These results are not so surprising, as we reconstruct from 800 frames of roughly the same scene. If the robot was moving in a straight line, as when driving a car on a road, we would expect the reconstruction time to increase roughly linearly with the number of sampled points per frame. However, when the robot travels in a loop with the camera always facing the cube, there will also be an effect of point clustering in this area of the model slowing down the running time.

5.5 Number of Frames

The results in tables 3 and 4 are a little more surprising. The reconstruction becomes extremely slow after a while. When using random sampling, this can be explained by the fact that we keep adding points to the model for each frame. After 800 frames one would then have almost 40 000 points in the model, closely located.

We thus also tried using ORB only, but the effect was the same, and the reconstruction time was in fact even worse. One explanation for this when using ORB is the number of false negative matches as discussed before, so that we keep adding points to the model also here, even though they are actually previously added.

Some causes for this behavior might be the same as described in section 5.4. The simulation loops around the same scene with the camera roughly facing the box the entire time. Thus, it may the case that the first 100 frames are more representative of a continuously moving robot. However, the results are not satisfactory for a robot involved in inspection and other tasks that require a thorough investigation of a local scene.

5.6 Inverse Cone Heuristic

Clearly, there is a problem with visual artifacts in the results. In the same manner as (Romanoni & Matteucci, 2015), (Romanoni & Matteucci, 2018) uses the inverse cone heuristic. However, the standard values for the weights as distributed in the code in (Romanoni & Matteucci, 2018), does not comply with the specifications of

(Romanoni & Matteucci, 2015). Specifically, the weights in the distributed code are

$$w_1 = 10.0, \quad w_2 = 2.0, \quad w_3 = 1.0.$$

However, (Romanoni & Matteucci, 2015) specifies that the following relation should be satisfied between the weights: $w_3 = \frac{w_2}{4}$, because 4 is the maximum number of neighboring tetrahedra that may have received weights for a single array. Thus, we should never be able to get a higher weight in the neighbors of neighbors, than in the neighbors of intersecting tetrahedra. Because of this peculiarity, we wanted to test for different values for these weights. Luckily, this was easily adjustable in the source code of the surface reconstructor.

We see from the results that none of the combinations really remove the visual artifacts, even though they are in perfect compliance with the values suggested in (Romanoni & Matteucci, 2015). When we remove the inverse cone heuristic, the car is more deteriorated but there appear to be fewer visual artifacts. This is a somewhat surprising result. The visual artifacts appearing from the different values of the weights appear quite similar, but the artifacts that appear without the inverse cone heuristic look quite different. This might be a peculiarity of this dataset, and it may be that the artifacts with the heuristic are more "correct" when viewed from a ray tracing procedure. However, it is hard to see whether this is actually the case.

The viewpoint from which the reconstructed scene is presented is quite similar to the view from which the first of the 30 frames is taken. Thus, the artifacts appearing in front of the car should not really be there, especially considering that we sample 400 points over 30 frames. The density is comparable to that in fig. 10d, where we see that there should be enough points behind the artifacts, i.e. enough rays passing through the artifact tetrahedra, to remove them.

The periphery of the scene may influence the reconstruction and introduce some artifacts. For example, in fig. 11d some artifacts appear to be connected to the periphery of the scene. However, this cannot explain most of the visual artifacts appearing. As an example, most of the artifacts in fig. 12d are far away from the periphery of the scene.

There is also an element of randomness in the model creation, as we randomly sample points from the image to project into space. The three models using the inverse cone heuristic in fig. 13 happened to be quite similar. However, this is not always the case. Figure 14 illustrates this. Here we run the model-creation with the same parameters, and thus only the sampled points are different across the reconstructions, due to the random sampling. We see that the reconstructions give quite different visual artifacts. Nevertheless, we are not able to explain why some of them occur. For example, Figure 14d shows a large artifact in front of the cube,

i.e. between the cube and the camera. This artifact should have been removed due to ray tracing. It is also stunning that it is in fact so large.

As stated in section 4.6, the tetrahedra in front of the cube are created somewhere in the last two frames. (Litvinov & Lhuillier, 2013) states that when new tetrahedra are created, old rays possibly passing through that tetrahedra should be retraced. This is why it remains an open question to us how this artifact is introduced. Interestingly, the artifact was removed when the manifold is updated more frequently, but then other artifacts were introduced. Hence, we can not state that this is the root of the problem.

5.7 Error Metric

In this report, we have only presented qualitative results with respect to the reconstruction quality. This is because we consider the task of finding an accurate and descriptive error metric challenging.

First, the synthetic dataset does not come with the ground truth surfaces. However, it does come with the ground truth depth for each pixel in each frame. Thus, one could define an error metric similarly to (Romanoni & Matteucci, 2015) or (Romanoni & Matteucci, 2018). The two methods define two errors metrics that appear quite similar, but that could give significantly different results. Both are based on Velodyne LIDAR measurements, which are comparable to our ground truth depth.

The error metric in (Romanoni & Matteucci, 2015) is defined as "the average of the distances between each Velodyne point and the nearest mesh triangle". We interpret that to mean that the depth at each lidar point measurement is projected into space, yielding a 3D coordinate, $\mathbf{x}$, of the measurement. Then, one finds the Euclidean distance between $\mathbf{x}$ and the mesh triangle with the smallest Euclidean distance to $\mathbf{x}$. They then take the average over all these measurements errors.

In (Romanoni & Matteucci, 2018), they define the error metric as "comparing the depth images of the reconstruction rendered in each frame, against the distance of the Velodyne points projected on the same image". We interpret this error metric as follows. For each camera frame, the 3D reconstructed surface is rendered onto the frame, creating a pixel-wise depth map of the scene as predicted from the model. Then this prediction is compared to the lidar measurements on the pixels that also have associated lidar measurements. Then the average is taken over all these errors.

In a near-perfect reconstruction, these metrics should yield approximately the same result. However, due to occlusions and visual artifacts, they will generally not give the same result. Consider for example the visual artifact in front of the cube in fig. 14d. Using the error metric in (Romanoni & Matteucci, 2015) would probably

give a smaller error than the error metric in (Romanoni & Matteucci, 2018). This is because when the lidar measurement is projected into space it will be located close to the surface of the cube. However, if the 3D model is rendered onto the current frame and one extracts the depth, one will get the depth to the visual artifact, while the measurement will measure depth to the cube. Thus, we will get a large error in this case.

The difference between the two error metrics illustrates an important obstacle in defining an accurate and consistent error metric: One does generally not know which point on the reconstructed surface corresponds to which point on the real surface, without distinct texture. Taking the Euclidean distance between a point on the reconstructed surface and its closest point on the real surface clearly does not give an accurate error metric: Several points on the reconstructed surface may be mapped to the same point on the real surface and vice versa. Also, one would not know how to properly align the two surfaces for comparison. Furthermore, focusing on one axis direction, e.g. simply taking the error in the z-axis, would not solve the issue either, because it would not accommodate for shifts and errors in the x- and y-direction.

One way to try to accommodate for this problem could be to provide texture to the reconstruction. This would, however, in the general case not solve the issue. In areas of the surface with unclear texture it will be difficult to find correspondence to the real surface. Furthermore, it may be difficult to texture the scene model consistently, especially in the presence of visual artifacts.

Consider a theoretical scene, which is a single, infinitely large plane in a single color. Consider then a bounded, textured surface reconstruction of the plane, which is slightly bumpy and shifted in a direction perpendicular to the plane. Finding a general, precise, and consistent error metric of this scene is in fact an ill-posed problem, in that it is no *one true way* of doing this.

6 Further Work

One line of further work is to define an error metric on the synthetic dataset, by somehow measuring the distance between the actual surface and the reconstruction. One possible way to do this is as follows. If one has a ground truth depth map for each frame, one can obtain a very dense point cloud of the scene. Then, an accurate surface reconstruction method can be applied to the dense point cloud, e.g. Poisson surface reconstruction, which will likely give a good approximation of the ground truth surface. Now, for each frame in the reconstruction process, we match the pixels involved in point projection to the pixels used for creating the ground truth, to know which 3D points are the same in the two reconstructions. Then, when comparing the surfaces, we can align them by "anchoring" these matched 3D points to each other, and one can e.g. measure the distance in the z-axis between the actual surface and the incrementally constructed surface.

Another line of further work is to improve the method of emulating the VSLAM data. Especially, we see an improvement in removing duplicate points. As discussed in section 3.3, we abandoned the idea in this report, due to the time complexity of our naïve search. One way to improve this, inspired by the boundary spatial hashing in (Romanoni & Matteucci, 2018), is to provide a spatial hashing of the points created from random sampling. This way, one can more quickly retrieve points that are near each other. Another way is to store the k nearest neighbors of the point.

Furthermore, there remains a need to study the appearance of the visual artifacts in the reconstructions performed in this report. Along with this, one may want to test the VSLAM emulation and the reconstruction method on more diverse synthetic data, e.g. on the entire VAROS dataset (Zwilgmeyer et al., 2021).

Lastly, one can test the identified real-time reconstruction method on real underwater video streams, to see how well it performs. To do this, one would first have to run a VSLAM algorithm on the data, and this will be more prone to errors than the emulated VSLAM data from simulated scenes and trajectories. Nevertheless, the final goal of the scene reconstruction is to make it work in a real underwater setting, and as such one may need to evaluate the different methods on real data.

7 Conclusion

In this report, we have performed a review of selected literature on surface reconstruction from point clouds, and identified a real-time manifold surface reconstruction method that can handle moving points. This method has been tested on a synthetic underwater dataset, and evaluated qualitatively and quantitatively. To perform these tests, we emulated VSLAM data on the dataset. The emulated data took use of the ground truth depth at each pixel and the ground truth pose of the camera, and had the ability to both detect ORB features in the images and sample random points at the image, and project these. We noticed that visual artifacts were an issue in the reconstruction, and that the reconstruction time could get slow as more and more frames were added, and lastly, we discussed why it is challenging to define an accurate error metric on the reconstruction.

Acknowledgements

I would like to thank Annette Stahl, Rudolf Mester, and Andreas Teigen for helpful guidance and input throughout the project, and I am especially thankful for the help provided by Mauhing Yip from start to end.

Bibliography

Alliez, P., Cohen-Steiner, D., Tong, Y. & Desbrun, M. (2007). Voronoi-based variational reconstruction of unoriented point sets. *Symposium on Geometry processing*, *7*, 39–48.

Berger, M., Tagliasacchi, A., Seversky, L., Alliez, P., Levine, J., Sharf, A., Silva, C., Berger, M., Tagliasacchi, A., Seversky, L., Alliez, P., Guennebaud, G. & Sur-, A. (2016). A Survey of Surface Reconstruction from Point Clouds To cite this version : HAL Id : hal-01348404 A Survey of Surface Reconstruction from Point Clouds. *Computer Graphics Forum*.

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

Biniaz, A. & Dastghaibyfard, G. (2012). A faster circle-sweep delaunay triangulation algorithm. *Advances in Engineering Software*, *43*(1), 1–13.

Boissonnat, J.-D. (1984). Geometric structures for three-dimensional shape representation. *ACM Transactions on Graphics (TOG)*, *3*(4), 266–286.

Bradski, G. (2000). The OpenCV Library. *Dr. Dobb's Journal of Software Tools*.

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*.

Carr, J. C., Beatson, R. K., Cherrie, J. B., Mitchell, T. J., Fright, W. R., McCallum, B. C. & Evans, T. R. (2001). Reconstruction and representation of 3d objects with radial basis functions. *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, 67–76.

Eelume AS. (2021). *Eeulume*. Retrieved 10th December 2021, from https://eelume.com/

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., ... Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, *585*(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2

Hoppe, H., DeRose, T., Duchamp, T., McDonald, J. & Stuetzle, W. (1992). Surface reconstruction from unorganized points. *SIGGRAPH Comput. Graph.*, *26*(2), 71–78. https://doi.org/10.1145/142920.134011

Huang, H.-Y. & Huang, S.-Y. (2020). Fast hole filling for view synthesis in free viewpoint video. *Electronics*, *9*, 906. https://doi.org/10.3390/electronics9060906

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

Kazhdan, M. & Hoppe, H. (2013). Screened poisson surface reconstruction. *ACM Trans. Graph.*, *32*(3). https://doi.org/10.1145/2487228.2487237

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

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

Litvinov, V. & Lhuillier, M. (2013). Incremental solid modeling from sparse and omnidirectional structure-from-motion data. *BMVC 2013 - Electronic Proceedings of the British Machine Vision Conference 2013*, 1–11. https://doi.org/10.5244/C.27.61

Litvinov, V. & Lhuillier, M. (2014). Incremental solid modeling from sparse structure-from-motion data with improved visual artifacts removal. *2014 22nd International Conference on Pattern Recognition*, 2745–2750.

Lorensen, W. E. & Cline, H. E. (1987). Marching cubes: A high resolution 3d surface construction algorithm. *ACM siggraph computer graphics*, *21*(4), 163–169.

Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, *60*(2), 91–110.

Muja, M. & Lowe, D. G. (2009). Fast approximate nearest neighbors with automatic algorithm configuration. *International Conference on Computer Vision Theory and Application VISSAPP'09)*, 331–340.

OpenCV. (2021). *Feature Matching tutorial*. Retrieved 19th December 2021, from https://docs.opencv.org/4.x/dc/dc3/tutorial_py_matcher.html

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.

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

Su, P. & Drysdale, R. L. S. (1997). A comparison of sequential delaunay triangulation algorithms. *Computational Geometry*, *7*(5-6), 361–385.

Szpak, Z. (2013). *Constrained parameter estimation in multiple view geometry* (Doctoral dissertation).

Tagliasacchi, A., Zhang, H. & Cohen-Or, D. (2009). Curve skeleton extraction from incomplete point cloud. *Acm siggraph 2009 papers* (pp. 1–9).

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

Yu, S. & Lhuillier, M. (2012a). Genus refinement of a manifold surface reconstructed by sculpting the 3d-delaunay triangulation of structure-from-motion points. *Proceedings of the 21st International Conference on Pattern Recognition (ICPR2012)*, 1021–1025.

Yu, S. & Lhuillier, M. (2012b). Incremental reconstruction of manifold surface from sparse visual mapping. *2012 Second International Conference on 3D Imaging, Modeling, Processing, Visualization & Transmission*, 293–300.

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.

Appendix

A The PointManager Class

The PointManager class creates a layer of abstraction when dealing with the points and their descriptor and ID, as well as whether they are marked as outliers. The class is implemented as follows:

```
import numpy as np


class PointManager:
    def __init__(self, tot_max):
        # store it like np array to get quick
        # access to all descriptors
        self._points = np.ndarray((tot_max, 1 + 32 + 3))
        self._id_counter = 0

    def add_point(self, des, loc) -> int:
        self._points[self._id_counter, :] = [
            self._id_counter, *des, *loc
        ]
        point_id = self._id_counter
        self._id_counter += 1
        return point_id

    def get_everything(self):
        return self._points

    def get_inliers(self) -> np.ndarray:
        inliers = np.asarray(
            self._points[:self._id_counter, 0] != -1
        ).nonzero()[0]
        return self._points[inliers, :]

    def get_descriptors(self):
        inliers = np.asarray(self._points[:, 0] != -1).nonzero()[0]
        return self._points[inliers, 1:33].astype(np.uint8)
```

```
    def mark_as_outlier(self, index):
        self._points[index, 0] = -1

    def is_outlier(self, index):
        return self._points[index, 0] == -1
```