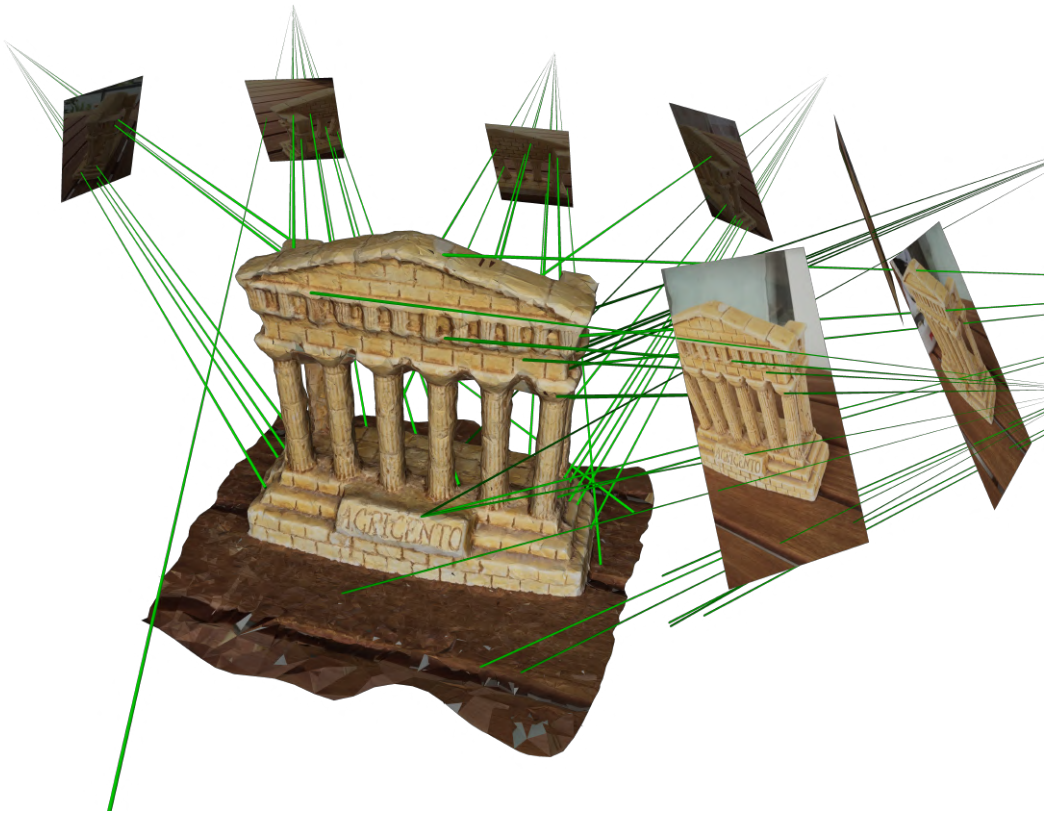
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Incremental and Adaptive 3D Reconstruction in the Large



Master Thesis CVG Lab
SS 2015

Ioannis Mariggis

Advisors: Torsten Sattler, Amaël Delaunoy
Professor: Prof. Marc Pollefeys

Abstract

This thesis proposes a set of algorithms for fast 3D reconstruction of a textured surface model using the SfM pointcloud, which are fast enough to enable real time reconstruction when using SLAM for tracking and computation of the 3D vertices. The sparse SfM pointcloud is used to construct a 3D Delaunay triangulation, and the visibilities of vertices are used to reverse trace rays from vertices to viewing cameras. Considering that in the physical world rays traverse in free space, tetrahedra intersected by rays are carved away from the triangulation. The surface can be reconstructed as the boundary between carved out and remaining tetrahedra. Contrary to related work utilizing similar approaches, the proposed method uses reprojection errors to formulate a probabilistic carving approach, where vertices are 3D Gaussian distributions. This enables a more sensible and accurate representation of the visibility based carving technique. Furthermore, photoconsistency and surface smoothness is also measured and considered in the reconstruction. The carving process is formulated as an energy functional in a graph representation and is efficiently optimized by finding the minimum cut. Unlike similar approaches, this thesis proposes to normalize the weights of the various graph costs based on previous reconstruction statistics. After the graph cut, remaining artefacts are cleared based on a spikiness measure and surface texture is computed. A partial textured surface reconstruction is available after every processed frame and can be rendered on the screen to provide live reconstruction feedback to the user.

Acknowledgments

I would like to thank my thesis advisors, Torsten Sattler and Amaël Delaunoy, for providing many good ideas and suggestions that helped a lot in forming the algorithms implemented as part of this thesis. Sharing their expertise on the subject helped a lot in getting on the right track for implementation of the various algorithms and helped avoid potential mistakes. Their active involvement in debugging helped speed things up. Particularly, the probabilistic tetrahedra carving technique, unique to this thesis compared to related work, that utilizes reprojections errors to estimate the probabilistic distribution of vertices in 3D space would not have been even considered for implementation without the input of Torsten Sattler. I would also like to thank Petri Taskanen for showing interest in the thesis and taking the time for constructing a dataset from the mobile device. Unfortunately bugs in the current conversion of the dataset format does not allow to include the results in this thesis. I am planning to debug as soon as possible after completion of the thesis however and share the results with the group, as it can provide a very accurate representation of the capabilities of the reconstruction pipeline on smartphones.

Contents

List of Figures

1 Introduction

The fundamental problem addressed by this thesis is the reconstruction of real-world scenes from image sequences recorded by a single camera. The three dimensional scanned scene is projected on the different camera frames of the image sequence with various poses. Thus the projected scene on the image sequences can be matched and combined to find the orientation of the camera in the real world. Matching points on different images can be triangulated to three dimensional coordinates. Reconstructing 3D coordinates for a large number of vertices forms a pointcloud, which ideally is sufficiently accurate and dense to give to represent the real scene. This problem is specifically described as Structure from Motion (SfM) and there are various techniques and algorithms focusing on problems such as speed, accuracy and density of the SfM pointcloud. Algorithms described as Simultaneous Localization and Acquisition (SLAM) generate an SfM pointcloud in real-time, using the spacial continuity of the images.

In many applications, the density of the pointcloud acquired from SfM is not high enough, in which case dense reconstruction techniques are implemented that allow for reconstruction of a large number of additional vertices. Often, the desired end result of the scene reconstruction problem is to obtain a surface that approximates the real scene as good as possible.

1.1 Overview

In the first chapter the introduction of the live reconstruction problem and its setting in the wider context is discussed. Related work is introduced and the advantages of the proposed reconstruction algorithms is briefly explained.

The second chapter explains the basic carving algorithm. It illustrates how tracing rays between cameras and visible vertices allows restriction of the scene's volume and explains the necessary steps to efficiently computing visibility based carving in an online fashion. It also illustrates that the Delaunay triangulation is a good choice for tessellating the volume into a tetrahedral grid suitable for the reconstruction purposes.

In the third chapter, the probabilistic model is introduced that uses reprojection errors to implement tetrahedra carving in a more sensible manner. In the simple carving process each tetrahedron is either part of the volume or not. Instead, the probabilistic formulation assigns probabilities to each tetrahedron. This allows for a more accurate visibility based carving algorithm and ultimately results to better reconstruction results.

The fourth section explains how graph cut based minimization framework is used to globally optimize an energy functional that considers tetrahedra probabilities and face costs. Face costs are newly introduced in this chapter and benefit the reconstruction pipeline by attempting to maintain high photoconsistency of reconstructed triangles, avoiding formation of large triangles at insufficiently explored regions of the scene.

The fifth chapter explains how artefacts that appear in spike formation can be removed from the reconstruction. These algorithms measure the spikiness of the reconstructed volume on individual tetrahedra and run after the graph cut minimization to clear remaining artefacts.

In the sixth chapter explains how the surface is reconstructed and textured and shows how the individual algorithms are executed in the reconstruction pipeline.

The last chapter shows experimental results that illustrate the overall reconstruction performance and the performance of individual algorithms in the pipeline. The quality and speed of reconstruction is tested under different modifications of the main experimental datasets, and the applicability of the reconstruction pipeline to different scenes is illustrated on a wider range of scene datasets.

1.2 Related Work

Related work in this area that specifically targets smartphones is presented in the papers *Turning Mobile Phones into 3D Scanners* [1] and *Live Metric 3D Reconstruction on Mobile Phones* [2], both done at the Computer Vision and Geometry (CVG) group at ETH Zurich. It is about a complete in-device reconstruction pipeline to generate dense poinclouds using stereo matching and rendering the results in real time. The implementation uses its own SLAM tracker that also produces a sparse pointcloud. However, in its current implementation is does not produce a textured surface model.

The probably most related project is *ProFORMA: Probabilistic Feature-based On-line Rapid Model Acquisition* [3] from Cambridge university, which is a simpler version of the pipeline described in this thesis, and was the initial source of inspiration for the thesis. ProFORMA uses the 3D Delaunay triangulation of the sparse SLAM pointcloud, which produces a tetrahedral grid forming the convex hull of the scanned scene. Tetrahedra are then carved away based on the visibilities of the vertices in the pointcloud from the cameras. The basic idea is that rays between vertices and cameras must be traversing empty space, and thus tetrahedra intersected by rays are classified as empty space and carved away. ProFORMA's latest algorithm version is implemented with a probabilistic model, that describes vertices as observations of the surface. For every triangle that is intersected from a ray, the probability of a hypothesis is formed, that in simple words reads: "if this triangle were to be part of the surface, what is the probability of the noisy vertex appearing behind that triangle". They assume a distribution of the vertex's location centred on the model surface.

ProFORMA is able to reconstruct in real time and produce texture mapped models. However, their reconstruction algorithms are only considering vertex visibilities and a threshold condition on tetrahedra probability to decide which tetrahedra are part of the object's solid volume and which are empty space. Although the probabilistic model improves the results, the underlying assumptions are not directly related to the uncertainly of the SfM accuracy of vertex reconstruction.

There are several projects and publications that use the Delaunay triangulation and visibility information to reconstruct a scene. Work in this thesis drew inspiration from the publication *Efficient Multi-View Reconstruction of Large-Scale Scenes using Interest Points, Delaunay Triangulation and Graph Cuts* [4]. This publication mentions the use of graph cuts to minimize an energy functional that includes costs from intersections of tetrahedra with rays, as well as smoothness and photoconsistency terms. This leads to a more qualitative surface reconstruction. The algorithms implemented as part of this project however, do not consider a probabilistic model that describes vertex noise, and it is not designed for real-time reconstruction. It does however inspire to use similar surface and photoconsistency costs for the reconstruction algorithms in this thesis.

The reconstruction pipeline minimizes an energy functional by expressing it in a graph and computing a minimum cut. The graph cut algorithm developed by Yuri Boykov and Vladimir Kolmogorov [5] is used for this purpose, as it is a vision specific implementation. Experiments show, that its use for graph cuts in the reconstruction pipeline is of sufficient speed. Further open source software used in code are OpenCV library [6] functions for image blurring and conversion from sRGB to L*a*b* colourspace.

1.3 Motivation

Traditionally, reconstruction from images has been done by separating the processes of collecting data and evaluating the data, reason for this being that the amount of computations associated with reconstruction vastly exceeded the capabilities of mobile devices. This kind of approach is often problematic when processing the data does not return sufficient reconstruction quality and in hindsight, the scene exploration needs to be cover certain regions in more detail. Historically, there was no alternative, however recently, smartphones have entered the market, which come equipped with a camera and fast CPUs. The most recent smartphones contain multicore CPUs and GPUs, and together with a high quality camera are excellent candidates for reconstruction. Since smartphone users in general constantly carry their devices with them, they offer the option of spontaneously recording videos and reconstructing on the device.

Such spontaneous reconstruction applications require the algorithms to run fast enough to be able to provide results to the user before physically leaving the scene. Ideally, the reconstruction runs in real time and is rendered live on screen. That way, users get instant feedback of the reconstruction and can decide on the spot if certain regions in the scene require more exploration. In offline implementations, waiting is required for the reconstructed surface to be generated and rendered. if in hindsight it is decided to capture more scene frames, it may not be a simple task to continue building a reconstruction that was halted, due to potential changes in the scene from objects moving away or changes in illumination.

In grant scope, the motivation to develop real-time reconstruction on a mobile phone is the high amount of flexibility and user friendliness it offers to capture and explore scenes. User friendliness arises form the fact, that during a real-time reconstruction with live rendering a user can decide on the spot which regions on the scene need further exploration and does not have to do many attempts and plan exploration beforehand.

Although the SLAM tracker can provide a sparse pointcloud, this does usually not contain sufficient detail. To obtain a more detailed scene reconstruction, producing a surface model from the SLAM pointcloud is required. This thesis proposes a set of algorithms in a pipeline that is capable of reconstructing surfaces from live video with a better quality than is noticed on similar projects.

The fact that the resolution of the reconstruction is not necessarily pre-determined and that there could be regions with large and regions with low resolution motivates the usage of the Delaunay triangulation instead of a regular voxel grid.

1.4 Contributions

The core reconstruction notion of carving tetrahedra away from the Delaunay triangulation based on SfM vertex visibilities has many appearances in various projects and publications.

What is unique to ProFORMA that this thesis draws inspiration form, is the utilization of a probabilistic model for handling the free space carving of tetrahedra. Compared to ProFORMA however, the algorithms designed in this thesis are more accurate model of the true uncertainties involved with SfM vertex reconstructions. In the probabilistic model used in this thesis, vertices are distributed in 3D space according to a trivariate Gaussian distribution, centred at the location of the vertex's observation. The re-projection errors of vertices on the various frames are directly used to estimate the parameters of the triavariate Gaussian distribution. This is then in turn used, to compute a probability of ray intersections with tetrahedra and ultimately probabilities of tetrahedra being part of the final volume or carved out and become free space.

In more contrast to ProFORMA, which employs a threshold based tetrahedra labelling, the algorithms here express the labelling process as a global minimization problem. This is very efficiently solved by expressing it in form of a graph end performing a minimum graph cut.

A further difference with ProFORMA is that the algorithms in this thesis also include photoconsistency costs for the faces between tetrahedra. Surface area of triangles is also used to avoid crude reconstruction of unexplored scene regions. These costs are inspired from [4], however in contrast to this these, that project does not focus on real-time implementation, and it doesn't use a probabilistic model either. In [4], the weighting of the three different costs, visibility, photoconsistency and surface area, is required to be provided by the user and optimal values are likely to differ between different scenes. The photoconsistency and area cost weightings in this thesis however, are normalised automatically based on reconstructed surface statistics from previous iterations.

Both these mentioned projects and other publications in related topics mention the use of *The Computational Geometry Algorithms Library* (CGAL) [7] for the Delaunay triangulation, a reason for this being that it is very efficient. CGAL however, does not compile for mobile OSes and since the algorithms developed in this thesis target mobile platforms, CGAL cannot be used. Initial development made use of *The Visualization Toolkit* (VTK), but because the Delaunay triangulation with VTK was becoming the reconstruction bottleneck, the triangulation has been re-implemented.

The vertices that are triangulated for use in scene reconstruction are not simple 3D coordinates, as it is usually the case when building the Delaunay triangulation, but also vertex visibilities from cameras are known. This extra information was used to re-design a different tetrahedron locator of the Delaunay triangulation algorithm, which seems to perform even better than CGAL. The triangulation algorithm in this thesis, contrary to related work, is capable of compiling on mobile platforms and is faster.

The reconstruction process is not perfect, and although many artefacts are carved away by a probabilistic formulation and expressing photoconsistency costs, some artefacts remain visible. Almost all of the remaining noticeable artefacts are spike formations. One last algorithm in the reconstruction pipeline of this thesis, that has not made an appearance on related works, is the utilization of solid angles spanned by the reconstructed volume on individual vertices to detect and remove spikes. This approach seems to perform well on clearing the remaining artefacts and badly reconstructed regions, increasing the overall quality of the reconstruction.

2 Core Algorithm

This chapter explains the general idea of free space carving tetrahedra from a Delaunay triangulation. The first section will show that 3D Delaunay triangulations provide a suitable tetrahedra grid as the base for reconstruction. The simple carving procedure is then explained and incrementally formulated to be computed online and updated at every new frame.

2.1 3D Delaunay Triangulation and Convex Hull

Given a set $\mathcal{P}$ of points in Euclidean 3D space, the Delaunanay triangulation decomposes the convex hull of $\mathcal{P}$into tetrahedra. A tringulation in 3D space is strictly speaking a tetrahedralization, such that vertices of tetrahedra belong to $\mathcal{P}$ and intersection of two tetrahedra is either empty or a vertex or a face or an edge. The 3D Denaunay triangulation has the specific property, that the circumsphere of avery tetrahedron in the triangulatios is empty, i.e. does not contain any of the points in $\mathcal{P}$. [8]

The goal is to use the sparse pointcloud obtained from SfM to create a surface mesh. Key concept is to turn the pointcloud into voluminous grid of elementary thee dimensional shapes that represents as good as possible the real volume of the scanned scene. The Delaunay Triangulation produces an unstructured grid of tetrahedra with some desirable properties for the application. In an unstructured grid, the tetrahedra can be of strongly varying size, which is a desirable property favouring reconstruction of scenes with strongly varying resolution. The two main benefits of an unstructured tetrahedral grid is that resolution is almost exclusively bound by the accuracy of the floating point variables used and the size of the grid is defined by the number of tetrahedra, rather than the range of a coordinate system. This means that memory usage to represent the volume is independent of resolution and Cartesian size of the scene and only depends on the number of tetrahedra. An unstructured tetrahedral mesh is therefore very desirable and provides good support for arbitrary scenes.

Such Triangulations usually return a grid that forms a convex hull of the volume of the entire scene. To obtain the actual volume, tetrahedra have to selectively be carved away, leaving only the ones that are located in the same space as the scene's solid volume and representing the real volume as good as possible. For this to be even possible, the real volume has to be able to be constructed from the available tetrahedra. Ultimately, every tetrahedron in the grid is either solid or empty. This means that regardless of the quality of the labelling process, given a triangulation, the reconstruction of the volume can be only as good as the grid allows. The triangulation should as much as possible allow separation of real volume from the rest of the convex hull. Figure 2.1 illustrates the importance of the triangulation algorithm and shows how the Delaunay Triangulation constructs useful grids.

The Delaunay Triangulation of a set of points in 3D is a unique (if no degenerate points) tetrahedral grid that offers further benefits. In 3D space, it minimizes the maximum radius of a tetrahedron's enclosing circumsphere. A similar property in two dimensional space is that the Delaunay Triangulation maximizes the minimum angle of all triangles. This avoids sharp

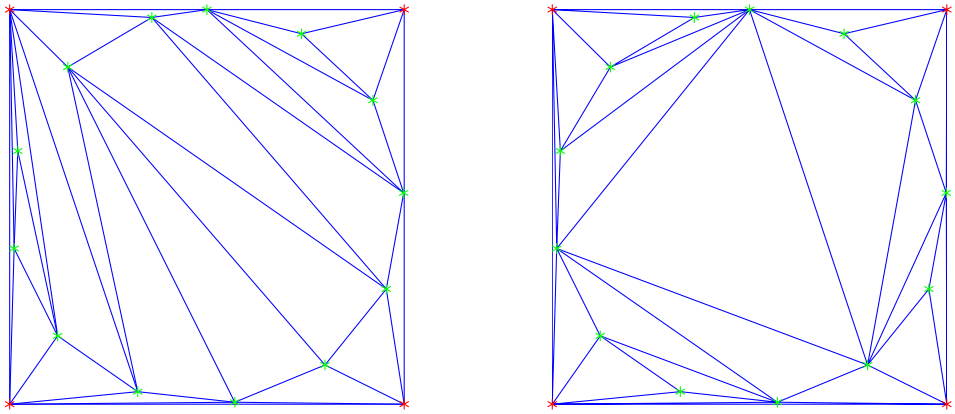


Figure 2.1: Delaunay (right) vs a random (left) triangulation of same data in 2D

triangles and analogously, sharp tetrahedra in the 3D case. Therefore, in presence of noisy vertex observations of the true surface, the 3D Delaunay Triangulation is a sensible choice.

Figure 2.1 above illustrates that the Delaunay triangulation is more likely to have an accurate boundary surrounding the real solid volume of the scene than a random triangulation. 2D triangulations are shown here, as they give clearer insight; the situation is similar in 3D space. The green points on the figure are samples from the hull of a circle and the red points are outliers. Although both triangulations use the same set of points, only in the Delaunay Triangulation the shape of the circle can be accurately recovered.

Delaunay Condition: In 3D, a triangulation of a set of points is a Delaunay triangulation if the circumsphere of all tetrahedra is empty, i.e. the circumsphere of every tetrahedron does not contain any vertices of other tetrahedra.

2.2 Notation

The notations `T` and `F` are generally used to describe tetrahedra and faces in 3D space. $\mathbf{T}_i = \begin{pmatrix} T_{ix} & T_{iy} & T_{iz} \end{pmatrix}^\top$ denotes the i^{th} vertex of a tetrahedron, and similar expressions denote vertices of faces. $\mathbf{V} = \begin{pmatrix} V_x & V_y & V_z \end{pmatrix}^\top$ is generally used to refer to a general vertex. Vertices of faces of tetrahedra are numbered in a consistent way, such that the orientation of faces with respect to a tetrahedron is consistent. Correct indexing has to be ensured when implementing the algorithms.

2.3 Simple Tetrahedra Carving

The tetrahedral grid has been formed and the true volume has to be extracted from the triangulated convex hull. In the SfM pointcloud, the set of cameras that lead to the reconstruction of every 3D vertex is known. This visibility information is used to carve tetrahedra away from the Delaunay Triangulation. Ideally all the ones that are not part of the scanned object's solid volume are carved away, i.e. tetrahedra that are free space; hence in literature, such a procedure is occasionally referred to as free space carving.

At start all the tetrahedra are assumed solid volume. Since the positions of the cameras

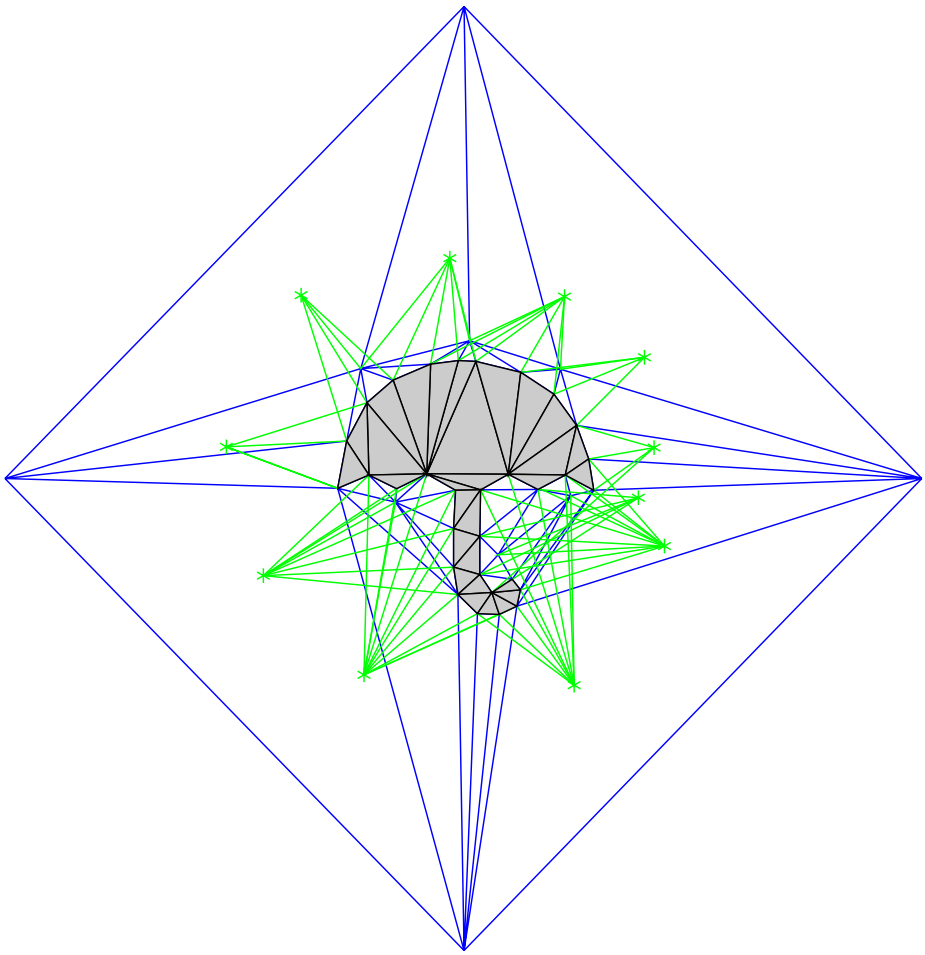


Figure 2.2: 2D Illustration of Simple Tetrahedra Carving

and the vertices of the triangulation are known, it can be assumed that the line segment connecting a camera with the vertices registered to that camera does not intersect any solid volume. With this logic, all tetrahedra that are intersected by a ray from a vertex to a camera must be free space. Surface reconstruction can be achieved by tracing all rays in the scene and relabelling intersected tetrahedra to empty and defining the surface as the the boundary between empty and solid tetrahedra.

Figure 2.2 is illustrating this procedure in 2D. In this illustrative set-up, each vertex in the triangulation has two camera viewing it. The green lines in the figure represent rays from vertices to cameras. All tetrahedra that are intersected by one or more rays are relabelled empty and the ones that are not intersected form the solid area of the scanned object. In this example, the triangles labelled solid are shaded grey and the 2D umbrella can be clearly recognised. In 3D space, the same concept applies.

2.3.1 Ray Processing

1: Find exiting adjacent tetrahedron to vertex (search blue set)
2: Keep moving through exiting faces till camera reached (yellow)

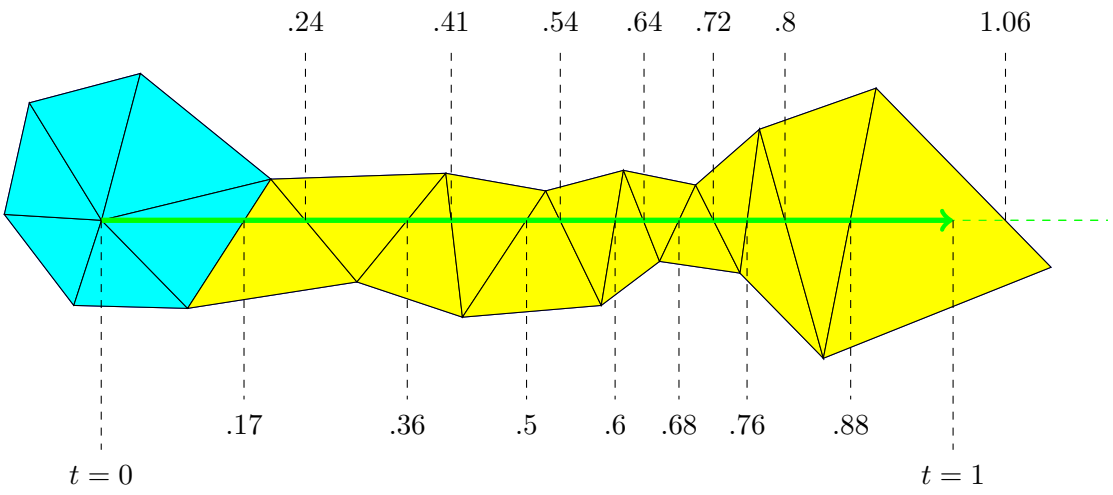


Figure 2.3: Processing Tetrahedra on a Ray

The scan for all intersections of a ray with tetrahedra can be efficiently implemented; vertices are aware of adjacent tetrahedra and every tetrahedra is aware of its neighbour through each of its face (and is also aware if there is no neighbour). Processing a ray comes down to two steps. First, from the set of adjacent tetrahedra to the vertex, the tetrahedron that is intersected by the ray originating on that same vertex needs to be found. The triangulation contains four boundary vertices, whose hull contains all inserted vertices. This means that reconstructed vertices are surrounded by tetrahedra in all directions and that a ray originating on a vertex is guaranteed to intersect a tetrahedron adjacent to that vertex.

Figure 2.3 summarizes how tetrahedra on a ray are traced and processed. Contrary to the common ray tracing technique in computer graphics, ray tracing in the algorithms in the thesis is done in reverse, from the vertex to the camera. A ray intersects with a tetrahedron if at least one of its triangular faces is intersected. The computation of the intersection between a ray and a triangle in 3D space is described in section 2.3.2. Since the ray origin is part of the tetrahedron, the three faces formed with the origin cannot possibly intersect the ray. Hence, only the face opposite the vertex needs to be checked for intersection with the ray. The size of the set of adjacent tetrahedra to a vertex is small enough to very quickly brute force search for the intersected tetrahedron. Experiments show that the number of adjacent tetrahedra to a vertex in average is 26 (see figure 7.17; slightly over 6 tetrahedra per vertex times 4 vertices per tetrahedron).

Once the first intersected tetrahedron is found, the second step is to continue tracing the ray till the camera is reached. If the ray enters a tetrahedron on one face, it has to exit on another face or terminate on the camera. Since the exiting face cannot be the same as the entering face, there are three candidate faces of which one is intersected. If it is not the first two, then it must be the third one. So the search for the exiting face will at most compute 2 intersections and is hence executed very fast too. All encountered tetrahedra during the trace are carved away. The ray trace terminates of the tetrahedron is reached, that contains the camera.

2.3.2 Ray Triangle Intersection

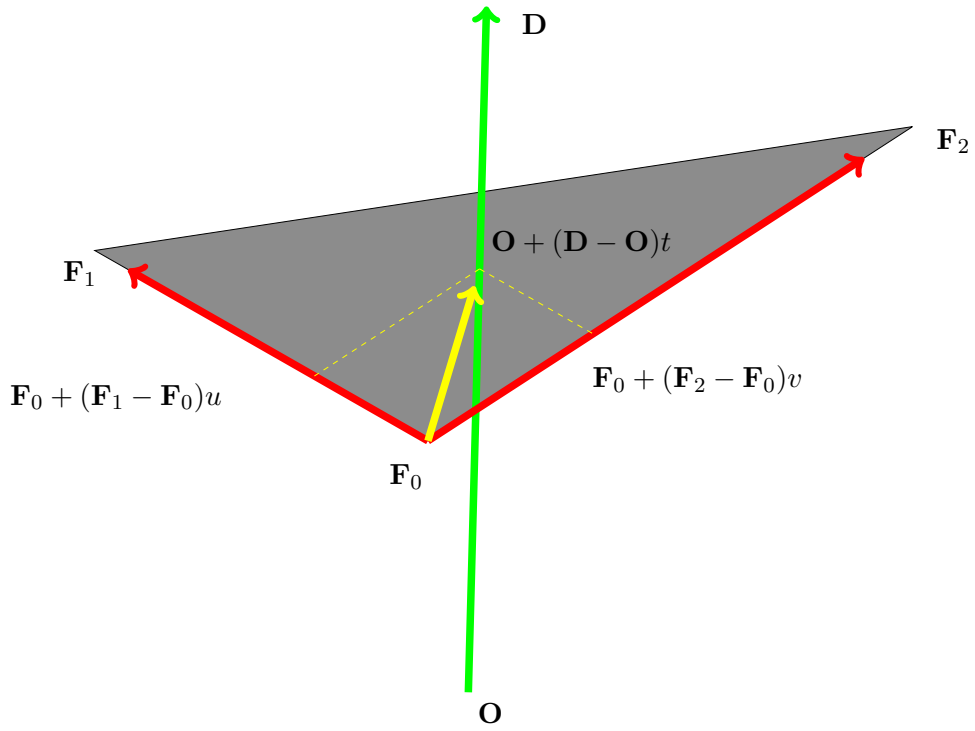


Figure 2.4: Intersection of a ray with a triangle

The geometry of any ray-triangle intersection is shown on figure 2.4. The triangle vertices

are $\mathbf{F}_i = \begin{pmatrix} F_{ix} & F_{iy} & F_{iz} \end{pmatrix}^\top$ with i in $(0, 2)$, the ray origin is $\mathbf{O} = \begin{pmatrix} x_O & y_O & z_O \end{pmatrix}^\top$ and the ray destination is $\mathbf{D} = \begin{pmatrix} x_D & y_D & z_D \end{pmatrix}^\top$. Note that the superscript ^{3} denotes that the vertices are part of a triangle; the notation is introduced in section 2.2. At the intersection point between the ray and the plane spanned by the triangle, following applies

$$\mathbf{O} + (\mathbf{D} - \mathbf{O})t = \mathbf{F}_0 + (\mathbf{F}_1 - \mathbf{F}_0)u + (\mathbf{F}_2 - \mathbf{F}_0)v \tag{2.1}$$

$$\mathbf{O} - \mathbf{F}_0 = (\mathbf{O} - \mathbf{D})t + (\mathbf{F}_1 - \mathbf{F}_0)u + (\mathbf{F}_2 - \mathbf{F}_0)v. \tag{2.2}$$

The vectors from $\mathbf{F}_0$ to $\mathbf{F}_1$ and $\mathbf{F}_0$ to $\mathbf{F}_2$ are used as the direction vectors of the plane. Using $\mathbf{B} = \mathbf{O} - \mathbf{F}_0$ and

$$\mathtt{A} = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} x_O - x_D & F_{1x} - F_{0x} & F_{2x} - F_{0x} \\ y_O - y_D & F_{1y} - F_{0y} & F_{2y} - F_{0y} \\ z_O - z_D & F_{1z} - F_{0z} & F_{2z} - F_{0z} \end{bmatrix}. \tag{2.3}$$

the equation becomes $\mathbf{B} = \mathtt{A} \begin{pmatrix} t & u & v \end{pmatrix}^\top$ and the solution

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \mathtt{A}^{-1}\mathbf{B}. \tag{2.4}$$

The inverse of the 3x3 matrix $\mathtt{A}$ is given by

$$\mathtt{A}^{-1} = \frac{1}{|\mathtt{A}|} \begin{bmatrix} a_{11}a_{22} - a_{12}a_{21} & a_{02}a_{21} - a_{01}a_{22} & a_{01}a_{12} - a_{02}a_{11} \\ a_{12}a_{20} - a_{10}a_{22} & a_{00}a_{22} - a_{02}a_{20} & a_{02}a_{10} - a_{00}a_{12} \\ a_{10}a_{21} - a_{11}a_{20} & a_{01}a_{20} - a_{00}a_{21} & a_{00}a_{11} - a_{01}a_{10} \end{bmatrix}. \tag{2.5}$$

Since the determinant $|\mathtt{A}|$ is only from a 3×3 matrix, it can efficiently be computed directly as

$$|\mathtt{A}| = a_{00}(a_{11}a_{22} - a_{12}a_{21}) - a_{01}(a_{10}a_{22} - a_{12}a_{20}) + a_{02}(a_{10}a_{21} - a_{11}a_{20}). \tag{2.6}$$

The point defined with u and v is on the triangle's plane with $\mathbf{F}_0 + (\mathbf{F}_1 - \mathbf{F}_0)u + (\mathbf{F}_2 - \mathbf{F}_0)v$. It is located within the triangle's parallelogram if both u and v are positive. For ray-plane intersections that are within the triangle, additionally $u + v < 1$ must hold.

Considering that the ray starts on $\mathbf{O}$, which is the vertex, if the sign of t is negative, the intersection is located on the line, that is a superset of the ray, but not on the ray itself, which is a line segment. So if $t < 0$, then the ray does not intersect the triangle. Similarly, if $t > 1$, the intersection happens behind $\mathbf{D}$, which is the camera, in which case the triangle is not intersected either.

The ray intersects the triangle on the shaded area of figure 2.4 if

$$u > 0 \quad \wedge \quad u < 1 \quad \wedge \quad v > 0 \quad \wedge \quad v < 1 \quad \wedge \quad u + v < 1 \quad \wedge \quad t > 0 \quad \wedge \quad t < 1. \tag{2.7}$$

Since none of u, v and t are used in equations other than computing the ray-triangle intersection, if the intersection does not occur within the triangle (i.e. shaded area on figure 2.4), the calculations of $\mathtt{A}^{-1}$, u, v and t do not occur in the above described order. Instead, the rows of $\mathtt{A}^{-1}$ are individually formed to individually calculate u, v and t (and in this order). Then, the conditions from (2.7) that can be evaluated with the partial results are computed and if for example $u < 1$ is violated, the code returns immediately without calculating v and t.

2.4 Delaunay Initialization

The triangulation is initialized by defining 6 boundary vertices $\mathbf{V}_i^B$ with integer i in $(0, 5)$ (B denoting boundary) such that

$$\begin{bmatrix} \mathbf{V}_0^B & \mathbf{V}_1^B & \mathbf{V}_2^B & \mathbf{V}_3^B & \mathbf{V}_4^B & \mathbf{V}_5^B \end{bmatrix} = \begin{bmatrix} -L_B & L_B & 0 & 0 & 0 & 0 \\ 0 & 0 & -L_B & L_B & 0 & 0 \\ 0 & 0 & 0 & 0 & -L_B & L_B \end{bmatrix}. \tag{2.8}$$

L_B is the length from the centre of the world coordinates system to the boundary vertices. These vertices span a convex hull by forming a very large octahedron, that contains all points that are going to be inserted. The boundary vertices are just for use in the algorithms and do not become part of the output model. L_B should be a very large value, so that new points are guaranteed to be within the bounding octahedron. It can be arbitrary large but since the code works with floating point numbers it should not exceed the accuracy limits; the boundary vertices are still going to be used in some computations and a too high L_B could potentially cause instability. (For an illustration, figure 2.2 shows a 2D Delaunay triangulation with boundary vertices with extremely small L_B). In practice, L_B can be set for example to 10000 times (or more) the distance between the first and second camera frame, as it is likely, that exploration will not exceed such large distances.

The bounding octahedron is split into 4 tetrahedra that are a valid Delaunay triangulation. Since the boundary vertices are a degenerate case for the Delaunay triangulation, there are a number of different constellations that provide a valid structure. The tetrahedra are in code always defined in the same way, but it is not mentioned here since any valid constellation produces the same result.

2.5 Incremental Delaunay Triangulation

When inserting a new vertex to the triangulation, some of the tetrahedra get removed, and new ones are created, re-triangulating the volume of the removed tetrahedra. In most cases there are more new tetrahedra after the insertion, but in some cases there are less enclosing faces to the re-triangulated volume than tetrahedra replaced, in which case an insertion reduces the number of tetrahedra in the triangulation. The process of updating a given triangulation by adding a vertex is split into two main steps. The first one is to locate a tetrahedron whose circumsphere includes the new vertex and the second step is to update the tetrahedral grid. After that, rays intersecting the changed volume are retraced in that volume to find which new tetrahedra are intersected.

In practice, in a given triangulation of realistic size, the number of tetrahedra whose circumsphere contains the new vertex is very small compared to the total number of tetrahedra and hence the localisation step of finding the tetrahedra can become the bottleneck for large triangulation sizes. The implementation in this thesis uses the visibility information provided by SfM to speed up the localization step of a tetrahedra whose circumsphere contains the new vertex.

2.5.1 Localization of Enclosing Tetrahedron

Instead of searching for any tetrahedron whose circumsphere contains the new vertex, the localization step in this code searches to find the tetrahedron that includes the vertex inside its volume. Although the enclosing tetrahedron is just one compared to a potentially larger set of tetrahedra that violate the Delaunay circumsphere condition, the following approach requires less computations and outperforms other approaches that were tested (figure 7.15 in section 7.7).

In SfM, at least two cameras need to have visibility on a newly reconstructed vertex. In real-time SfM, one of the cameras is the newest frame and the other camera (referring to it as auxiliary camera) from a previous frame. In code, every camera is aware of the tetrahedron in the grid where it is located. Ensuring an always up-to-date state of a camera's enclosing tetrahedron can be done by updating it prior to exiting the ray trace loop. The final tetrahedron in the trace of a ray is the one enclosing the ray's camera. (Refer to figure 2.3 for an illustration in 2D).

Prior to every new vertex insertion, all cameras except the newest one are guaranteed to have correct awareness of their location in the grid. Hence, it is possible to locate the new vertex containing tetrahedron by tracing the ray from the auxiliary camera to the vertex (only here the trace starts at the camera, elsewhere it always starts on the vertex). If the tetrahedron that contains the auxiliary camera does not contain the new vertex, then the reverse trace traverses all tetrahedra on the ray's path. When finally a tetrahedron is entered that does not have an exiting face, it means that the vertex is contained within that tetrahedron. In the more rare case of the auxiliary camera's tetrahedron also containing the new vertex, there is no need for a trace as the location is already found; this case is detected if the search through the 4 faces does not return and positive intersection result (defined in section 2.3.2 equation (2.7), which describes the intersection conditions).

This way, the same equations and code can be used as for ray tracing, without the need of calculating (2.9) for testing if the new vertex lies within the traversed tetrahedra. If there is an exiting face during trace, it is implied that the tetrahedron is not enclosing the new vertex, otherwise it is. The method is illustrated in a 2D Delaunay triangulation in figure 2.5.

This method of tracing a ray through the triangulation could be also used as a general incremental Delaunay triangulation technique, where instead of the trace's origin being the auxiliary camera, it could be the previously inserted vertex or a more intelligently chosen vertex. The auxiliary camera however is a good choice for the ray's origin as it guaranteed to be relatively close to the new vertex without the need of having a further computation decide which vertex to use as the trace origin (it also outperforms choosing the previous vertex).

2.5.2 Vertex in Tetrahedron Circumsphere Test

The tetrahedron that contains the vertex is definitely violating the Delaunay condition, which states, that the circumsphere of any tetrahedron must not contain any vertices from other tetrahedra. Formula (2.10) can be used to test efficiently whether a tetrahedron's circumsphere contains a vertex.

Given `T` as the tetrahedron and $\mathbf{V}$ as the vertex, the result if $\mathbf{V}$ is within the circumsphere of `T` is determined by the orientation of `T` and the sign of the determinant (formula proposed in [9])

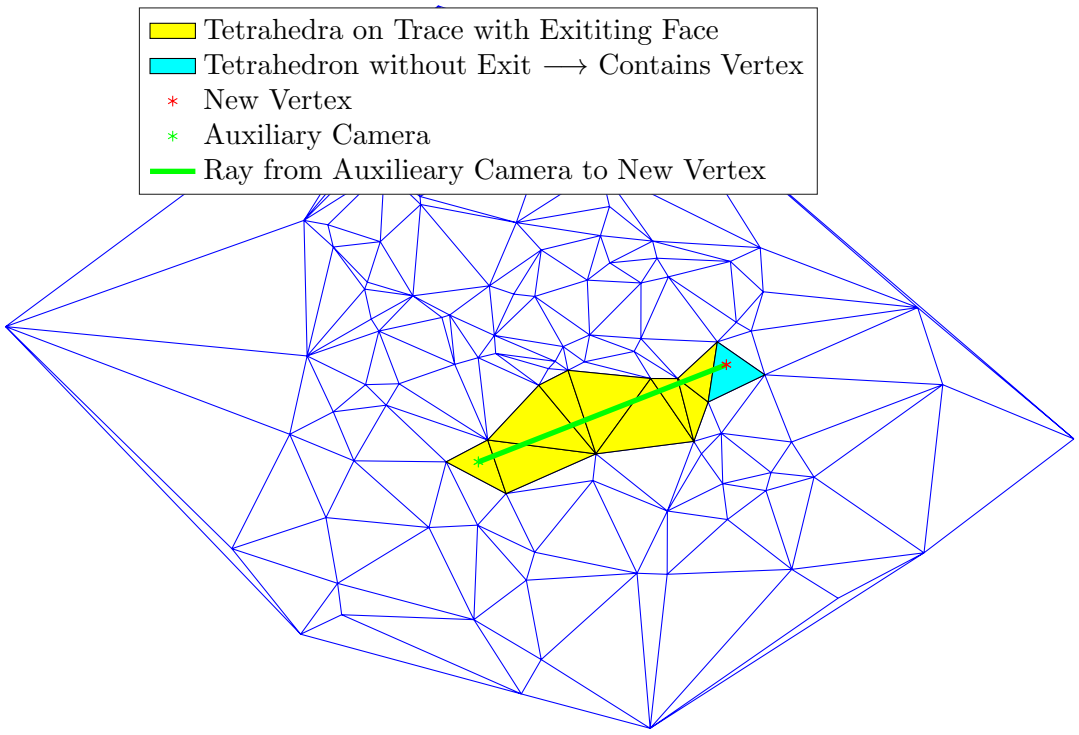


Figure 2.5: Localization of Tetrahedron Containing New Vertex

$$D_{5\times5} = \left|\mathtt{A}\right| = \begin{vmatrix} 1 & T_{0x} & T_{0y} & T_{0z} & \|\mathbf{T}_0\|^2 \\ 1 & T_{1x} & T_{1y} & T_{1z} & \|\mathbf{T}_1\|^2 \\ 1 & T_{2x} & T_{2y} & T_{2z} & \|\mathbf{T}_2\|^2 \\ 1 & T_{3x} & T_{3y} & T_{3z} & \|\mathbf{T}_3\|^2 \\ 1 & V_x & V_y & V_z & \|\mathbf{V}\|^2 \end{vmatrix}. \tag{2.9}$$

Given that all tetrahedra in the triangulation have clockwise oriented faces when looking from the opposite vertex, the vertex $\mathbf{V}$ is inside the tetrahedron's circumsphere if the determinant is $D_{5\times5} < 0$.

Since for a tetrahedron the vertices remain the same over its lifetime, by expanding the determinant along the 4^{th} row (numbering starts at 0) the computation becomes (2.10). A minor $M_{i,j}$ is the determinant of the 4×4 matrix formed by removing row i and column j from matrix $\mathtt{A}$. Minors $M_{i,j}$ are computed by expansion on any row or column (row 0 in code) and using (2.6) for their sub-minors of 3×3 matrices.

$$D_{5\times5} = \begin{pmatrix} 1 & V_x & V_y & V_z & \|\mathbf{V}\|^2 \end{pmatrix} \begin{pmatrix} M_{4,0} & -M_{4,1} & M_{4,2} & -M_{4,3} & M_{4,4} \end{pmatrix}^\top. \tag{2.10}$$

All the minors are determinants of $\mathtt{A}$ without the last row and hence are independent from the coordinates of the vertex that is tested. So they are calculated at most once for every

tetrahedron and are reused for further tests. This way, from the second sphere test onwards, the computations reduce to only forming $\|\mathbf{V}\|^2$ and calculating a dot product.

2.5.3 Find Collision Set and Enclosing Faces

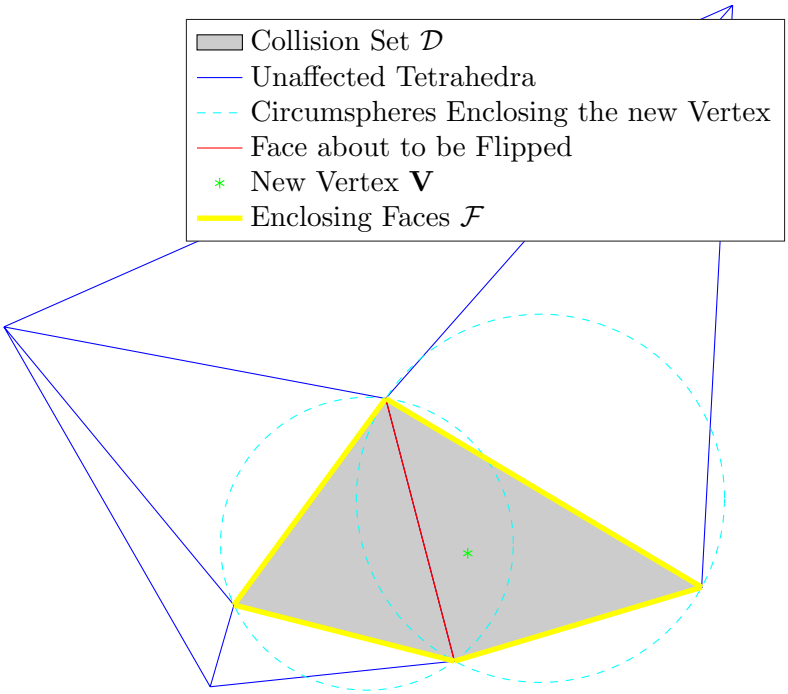


Figure 2.6: Finding Delaunay Conflicts at Insertion

Figure 2.6 illustrates in 2D the geometric components relevant to preparing a point insertion in a Delaunay triangulation. When the new Vertex is about to become part of the Delaunay triangulation, all tetrahedra, whose circumsphere captures the new vertex, do not satisfy the Delaunay condition when the new vertex is introduced to the triangulation; their circumsphere would not be empty. These tetrahedra are forming the collision set $\mathcal{D}$ and will be removed during the grid update process of including the new vertex in the triangulation. The gap left by removing the collision set needs to be filled up by new tetrahedra that satisfy the Delaunay circumsphere condition.

For being able to efficiently create the new tetrahedra in the next step, the set of enclosing faces $\mathcal{F}$ needs to be captured. This is the set of triangles that forms the closed surface that separates the new, re-triangulated part of the tetrahedral grid from the remaining part of the grid. If the re-triangulated volume is on the boundary, the boundary triangles adjacent to new tetrahedra are also part of $\mathcal{F}$.

These sets can be efficiently found using a recursive search as defined in algorithm 1. Denoting the new vertex as $\mathbf{V}$ and the tetrahedron that encloses that vertex as `T`, the function is called the first time as

$$\texttt{FindDF}(\texttt{T}, -1, \mathbf{V}). \tag{2.11}$$

If the collision set is larger than 1 tetrahedron, the function calls itself and upon return of

the first call, both sets $\mathcal{D}$ and $\mathcal{F}$ are fully populated.

Algorithm 1: Finding Delaunay Conflicting Tetrahedra and Enclosing Faces

Input: `T` is the tetrahedron in $\mathcal{D}$ that is being scanned
Input: F is the face number that has already been checked
Input: $\mathbf{V}$ is the new vertex that will become part of the triangulation
Result: Every call of this function contributes to populating the sets $\mathcal{D}$ and $\mathcal{F}$

`FindDF` (`T`, F, $\mathbf{V}$)
1 Mark current tetrahedron `T` for removal, by adding it to set $\mathcal{D}$
2 **forall the** *face numbers* $f \neq F$ **do**
3 **if** *`T` has no neighbour though face number* f **then**
4 Add `T`'s f^{th} face to set of enclosing faces $\mathcal{F}$
5 **continue**
  end
6 Let `N` be `T`'s neighbour though face f
7 **if** *`N`'s sphere has been checked* **then**
8 Reuse result
9 **else**
10 Compute in-sphere test
  end
11 **if** $\mathbf{V}$ *is in `N`'s circumsphere* **then**
12 **if** *FindDF has not been called on `N`* **then**
13 $F_N \leftarrow$ numbering of `T`'s f^{th} face as seen by `N`
14 `FindDF` (`N`, F_N, $\mathbf{V}$)
  end
15 **else**
16 Add `T`'s f^{th} face to set of enclosing faces $\mathcal{F}$
  end
 end

`FindDF` is first called on the tetrahedron `T` that is enclosing the new vertex that is about to become part of the grid. -1 is passed as the face number argument in the recursive `FindDF` search in order not to skip checking any of the four faces of `T`. The recursive search takes care to not include duplicates in $\mathcal{D}$.

2.5.4 Update Tetrahedral Grid

Figure 2.7 illustrates the corresponding updates in the grid for the same data as in figure 2.6. The newly formed tetrahedra are captured in the set $\mathcal{N}$.

The logic behind the update is simple, the set of tetrahedra mark for removal $\mathcal{D}$ is removed along with any links that point to them. The void volume is filled by forming new tetrahedra by using the newly inserted vertex $\mathbf{V}$ and all the enclosing faces $\mathcal{F}$.

This update corresponds to splitting the tetrahedron that contains $\mathbf{V}$ to three smaller tetrahedra by using the original tetrahedron's faces and the new vertex. Then, if there are any adjacent tetrahedra that are marked for removal, the common face between these and the

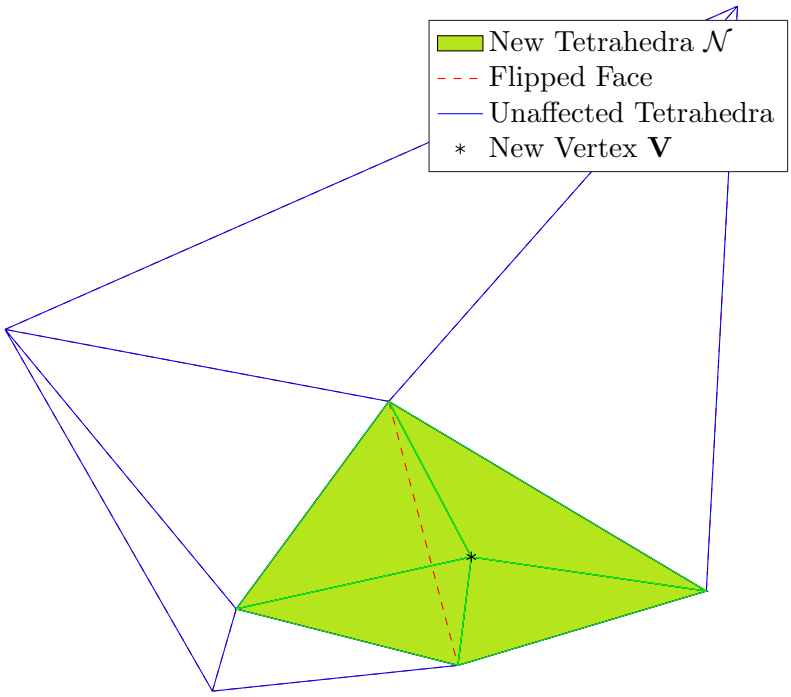


Figure 2.7: Updating the Tetrahedral Grid

adjacent smaller tetrahedron created in the first step is flipped. Flipping in an operation on two adjacent tetrahedra, sharing a common face. Flipping these two tetrahedra is essentially as basic as creating two different tetrahedra using the same vertices; visually it looks like the face, splitting a pentahedron into 2 distinct tetrahedra, is flipped. Flipping is also illustrated in figure 2.7. After performing flips on all tetrahedra in $\mathcal{D}$, the result would be exactly the same as forming the new tetrahedra straight away with the enclosing faces (in fact this is how the validity of the approach with using the enclosing faces can be derived). Re-triangulation of the volume formed with tetrheahedra in $\mathcal{D}$ is implemented efficiently in code by using the set of enclosing faces in $\mathcal{F}$.

In order to completely remove the tetrahedra in $\mathcal{D}$, their references must be removed from all adjacent tetrahedra. This however is done automatically when the references are updated as described in the next section 2.6.1; they override the old references in the neighbouring tetrahedra. This is guaranteed, since the grid's volume is exactly the same as prior to the insertion. The only thing associated with removal of tetrahedra in $\mathcal{D}$, other than reusing the memory, is to remove reference from the entries in the vertex lists that contain reference to all adjacent tetrahedra.

2.6 Recovering Ray Traces

After inserting a new vertex to the triangulation and re-triangulating, the new tetrahedra have no ray intersections accounted for. Rays that were intersecting the volume prior to the grid update need to be traced again but only within that volume, as the grid update is only impacting tetrahedra in $\mathcal{N}$. In order to be able to know which rays have to be retraced, tetrahedra are aware of the rays that intersect them and which face each ray is intersecting.

It is sufficient to only log exiting intersections.

Figure 2.8 illustrates the two different ray scenarios that are handled differently in code, and the usefulness of the distinction becomes clear in follow up sections.

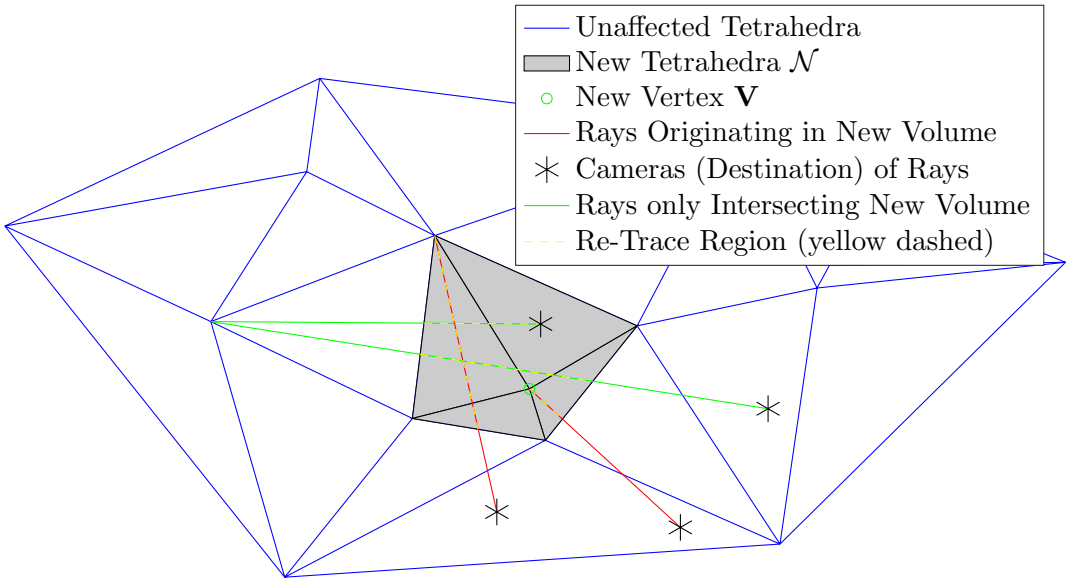


Figure 2.8: Types of Rays w.r.t. the Re-Triangulated Volume

2.6.1 Neighbouring Links in Grid

It was mentioned before that all tetrahedra in the grid are aware of their neighbouring tetrahedra. They are aware which neighbouring tetrahedron is adjacent through each of the four faces; and they are also aware what that face's number is, as seen from the neighbouring tetrahedron. This allows for a quick traversal of the grid during tracing. A tetrahedron is naturally aware of its vertices, but vertices are also aware of the tetrahedra they are adjacent to. This is important as it allows fast traversal through the tetrahedral grid and thus a fast implementation of ray tracing.

With a routine as shown in 2 that uses the set of enclosing faces $\mathcal{F}$ it is possible to efficiently update the links between new and remaining tetrahedra and among the faces shared between new tetrahedra.

By introducing the set of new tetrahedra $\mathcal{N}$ and having removed tetrahedra listed in $\mathcal{D}$, the links within the grid need update. The links through the enclosing faces in $\mathcal{F}$ are between new tetrahedra in $\mathcal{N}$ and tetrahedra that were unaffected by the new insertion. For readability, the unaffected tetrahedra in the Delaunay triangulation are referred to as the old tetrahedra. To implement updates efficiently, each enclosing face is aware of which tetrahedron has been created with it and with which old tetrahedron the face is shared with. The face number as seen by the old tetrahedron is also known, and by definition, the enclosing face is face number 0 in the new tetrahedron. Updating links through the enclosing faces with this information is straight forward.

The rest of the faces of the new tetrahedra are shared among themselves. Since the cardi-

nality $|\mathcal{N}|$ is relatively small, a brute force search through the tetrahedra in $\mathcal{N}$ that searches for matching faces could form the internal links very quickly. However, since the enclosing faces are aware of which tetrahedron in $\mathcal{N}$ they are part of, and since all shared faces in $\mathcal{N}$ share the common vertex $\mathbf{V}$, the brute force search for shared faces between tetrahedra in $\mathcal{N}$ can be easily reduced to a search for shared edges of triangles in $\mathcal{F}$.

When the tetrahedra in $\mathcal{N}$ get created, their links read $\varnothing$. When an edge match in the search is found, then both sides of the link get written, so that the same match does not have to be searched a second time. Hence, when looping through the edges of the enclosing faces, by skipping edges whose correspondent face does not link to $\varnothing$, redundant search is avoided.

Algorithm 2: Updating Links within the Grid after Insertion

```
   UpdateGridLinks(F)
1  forall the triangles F in set of enclosing faces F do
2  |  O ← old tetrahedron adjacent to T
3  |  N ← new tetrahedron adjacent to T
4  |  Link O and N as neighbours
5  |  forall the edges E of F with unassigned neighbours do
6  |  |  Search exhaustively in F to find triangle sharing the edge
7  |  |  M ← new tetrahedron adjacent to neighbouring triangle through edge E
8  |  |  Link M and N as neighbours
   |  end
   end
```

2.6.2 Origin Rays

A ray whose source vertex is part of a tetrahedron's vertices and which intersects the tetrahedron is described as originating in that tetrahedron. Every tetrahedron is aware of the rays that are originating from it. The information is fed into the tetrahedra when the first exiting tetrahedron is found at the beginning of a ray trace, which described in section 2.3.1 and illustrated in figure 2.3; prior to continuing the ray trace after the first exit, the tetrahedra logs that the ray originates within it. In that figure, the origin tetrahedron of the ray would be the one among the light blue coloured tetrahedra which intersects the ray.

When it comes to recovering traces, what needs to be considered is that the newly formed volume had been formed by removing the tetrahedra in set $\mathcal{D}$ and re-triangulating the volume to form the tetrahedra in $\mathcal{N}$. This means, that all rays originating in one of the tetrahedra in $\mathcal{D}$ must also originate in one of the tetrahedra in $\mathcal{N}$. Figure 2.8 illustrates that origin rays in the replaced volume could originate from any vertex in that volume.

In order to be able to recover the ray in cases of originating rays in the volume, references to origin rays is captured from every tetrahedron in $\mathcal{D}$ before tetrahedra in $\mathcal{D}$ are removed and are used to form the set $\mathcal{R}_{\mathcal{O}}$ of all rays originating in the modified volume that need to be re-traced.

Since the originating tetrahedron of rays in $\mathcal{R}_{\mathcal{O}}$ has been removed, it is unknown where these rays originate after the update. However, the origin tetrahedron of a ray must be part of the new volume and hence it must be in $\mathcal{N}$, and it also must be adjacent to the vertex that defines the origin of the ray. The options are therefore narrowed down to the intersection of

the sets $\mathcal{N}$ and set of tetrahedra adjacent to the ray's vertex.

Retracing the rays in $\mathcal{R}_\mathcal{O}$ for tetrahedra in $\mathcal{N}$ can be efficiently done by using the same code that processes intersections of rays. To reuse the algorithm for retraces it is sufficient to amend the trace termination condition to stop if the ray reaches the unmodified part of the tetrahedral grid. To allow this additional termination condition, the boolean *RetraceOption* argument has to be **true**.

For retracing all origin rays, the function described in algorithm 3 is invoked for every ray $(\mathbf{V}, \mathbf{C}) \in \mathcal{R}_\mathcal{O}$ by

$$\texttt{ProcessRay}(\mathbf{V}, \mathbf{C}, \mathbf{true}, \mathcal{N}). \tag{2.12}$$

Algorithm 3: Starting a Ray Trace Process

Input: $\mathbf{V}$ is the vertex of the ray, where the trace is started
Input: $\mathbf{C}$ is the camera of the ray, where the trace terminates
Input: $\mathcal{N}$ is the set of the new tetrahedra from the last point insertion
Input: *RetraceOption* is **true** to constrain tracing in set $\mathcal{N}$ only or **false** otherwise

`ProcessRay` ($\mathbf{V}, \mathbf{C}, \mathit{RetraceOption}, \mathcal{N}$)

```
1  forall the tetrahedra T adjacent to V do
2      if RetraceOption and T ∉ N then
3          continue
       end
4      f ← find number of vertex V in T
5      if Ray from V to C intersects the face opposite vertex f then
6          Carve the current tetrahedron T to empty
7          T adds (V, C) to its origin rays
8          T adds (V, C) to its list of rays through face f
9          N ← findT's neighbour on face f
10         f_n ← numberfT's f^th face as seen by N
11         if Trace has not reached camera then
12             TraceRay (N, f_n, V, C, RetraceOption)
           end
13         return
       end
   end
```

2.6.3 Rays through Enclosing Faces

The majority of rays that intersected tetrahedra in the former set $\mathcal{D}$ are originating in tetrahedra in the unaffected part of the grid. By having all tetrahedra being aware of the rays that are intersecting them and are also aware of the faces through which the rays exit them, retracing of such rays is very convenient.

The enclosing faces are the staring point from which these rays enter the volume formed with tetrahedra in $\mathcal{N}$ (illustrated by green rays in figure 2.8. Enclosing faces are aware of their adjacent old tetrahedra and the face number that they represent, hence a list of all

rays entering the volume of tetrahedra in $\mathcal{N}$ through any particular face is directly available. Simultaneously, enclosing faces are also aware of which tetrahedron among the ones in $\mathcal{N}$ they are adjacent to, and that the enclosing face is per definition face number 0. This is all the information that is needed for running the retrace and it can be obtained without the need of any search routines at all.

The rays described here are retraced from new tetrahedra of enclosing faces, till either the end of the volume defined by tetrahedra in $\mathcal{N}$ or the ray's camera has been reached (both such cases are illustrated in figure fig:RetraceTypes). For this, the retrace function that is illustrated by the yellow (2D) tetrahedra of figure 2.3 is reused with the additional return condition for when a tetrahedron $\notin \mathcal{N}$ is reached. The code in algorithm 4 retraces the rays through all faces by reusing the trace code from algorithm 5.

Algorithm 4: Using Trace Algorithm for Re-Tracing all Rays through $\mathcal{F}$ in $\mathcal{N}$

```
   RetraceFromFaces (N)
1  forall the tetrahedra T in N do
2  |  N ← neighbour through face 0. N is a tetrahedron in the unchanged Delaunay
   |  volume
3  |  f_n ← face number of T's face 0 as seen by N
4  |  forall the Rays (V, C) exiting tetrahedron N through N's f_n^th face do
5  |  |  TraceRay (T, 0, V, C, true)
   |  end
   end
```

Algorithm 5: Tracing Tetrahedra Intersections on a Ray

Input: `T` is the currently processed tetrahedron
Input: f is the face number where the ray entered the cell
Input: $\mathbf{V}$ is the vertex of the ray
Input: $\mathbf{C}$ is the camera of the ray
Input: *RetraceOption* is **true** if retracing in $\mathcal{N}$ only or **false** otherwise

`TraceRay` (`T`, $f, \mathbf{V}, \mathbf{C}$, *RetraceOption*)

```
1  while true do
2  |  if RetraceOption and T ∉ 𝒩 then
3  |  |  return
   |  end
4  |  Label T as an empty tetrahedron
5  |  f ← find face through which the ray exits tetrahedron T
6  |  N ← T's neighbour through face f
7  |  f_n ← T's f^th face's number as seen by N
8  |  if Trace has reached the camera at C then
9  |  |  return
   |  end
10 |  T ← N and f ← f_n        // continue trace by moving to next tetrahedron
   end
```

3 Probabilistic Carving

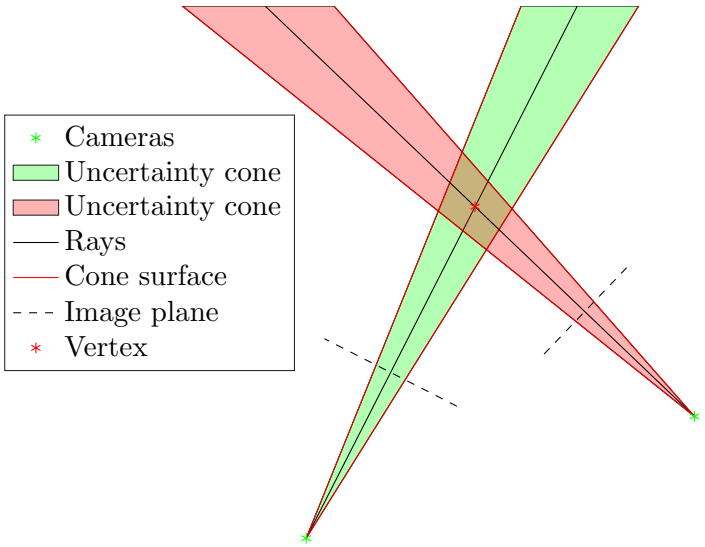


Figure 3.1: Uncertainty Region for the True Location of the Vertex.

The simple carving procedure from the previous section carves all tetrahedra intersected by rays. Due to the noise in the pointcloud from SfM however, rays from misplaced vertices carve away tetrahedra that should have been kept. In an approach to overcome this, the uncertainty of reconstructed vertices is taken into account to formulate the carving algorithm in a probabilistic way. A consequence of noisy vertices is, that the origin of the rays that intersect tetrahedra and carve them is also susceptible to noise and hence the intersections themselves are not guaranteed to happen. Instead, intersections have a certain probability of occurence, and hence tetrahedra have a certain probability to be carved away or remain part of the volume.

The uncertainty of the location in 3D space of a vertex $\mathbf{V}$ is closely linked to the reprojection errors on the image plane of cameras reconstructing $\mathbf{V}$. On the image plane, there coordinates registered by SfM differ from the coordinates obtained when projecting $\mathbf{V}$ using the estimated camera pose (i.e. the reprojection error). The probability distribution of $\mathbf{V}$'s SfM coordinates on a camera's image plane is assumed to be a 2D Gaussian, with circular contours.

Figure 3.1 shows an example topology. Because the true projection of $\mathbf{V}$ is a random variable, the true line between $\mathbf{V}$ and the camera is a random variable too. Given a Gaussian model (assuming for now that the Gaussian's parameters are correctly known; which is not the case) it is possible to know the the line between the vertex and the camera is with a probability of 0.997 bound within the cone that intersects the image plane on the distribution's

3σ contour (this cone is illustrated on figure 3.1 for both cameras). Therefore, the region in 3D space of the true location of the vertex is bound with at least 0.997^2 probability (more given correlation) by the intersection of the two cones in the example of figure 3.1.

The distribution of **V**'s true location in 3D space is modelled as a 3D Gaussian distribution, centred at the observed location of **V**.

3.1 Probabilistic Model

The uncertainty region of vertices in 3D space is three dimensional, i.e. an observed vertex deviates from the truth in X, Y and Z. However, for the purpose of making the mathematic formulation and algorithmic implementation simpler, given a vertex and a camera, the vertex's uncertainty is only considered on the ray. In other words, given a ray from a vertex **V** to a camera **C**, as far as the ray is concerned, **V**'s true location is somewhere on the ray.

The 1 dimensional space on the ray is defined in such way that the observed vertex is at $t = 0$ and the camera is at $t = 1$. Modelling uncertainty of the vertex on the ray means that the location of the vertex's observation is at $t = 0$ and the true location of the vertex remains a random variable, with a probability density function on the dimension of the ray.

The distribution is assumed to be Gaussian with standard deviation σ and with the assumption that the observed vertex location at $t = 0$ to be the mean (i.e. a zero mean Gaussian distribution is assumed), as displayed on figure 3.2. Note that on the figure the axis on the ray is displayed and hence the vertex V and camera C have a one-dimensional coordinate on the ray; in fact $V = 0$ and $C = 1$ on the ray.

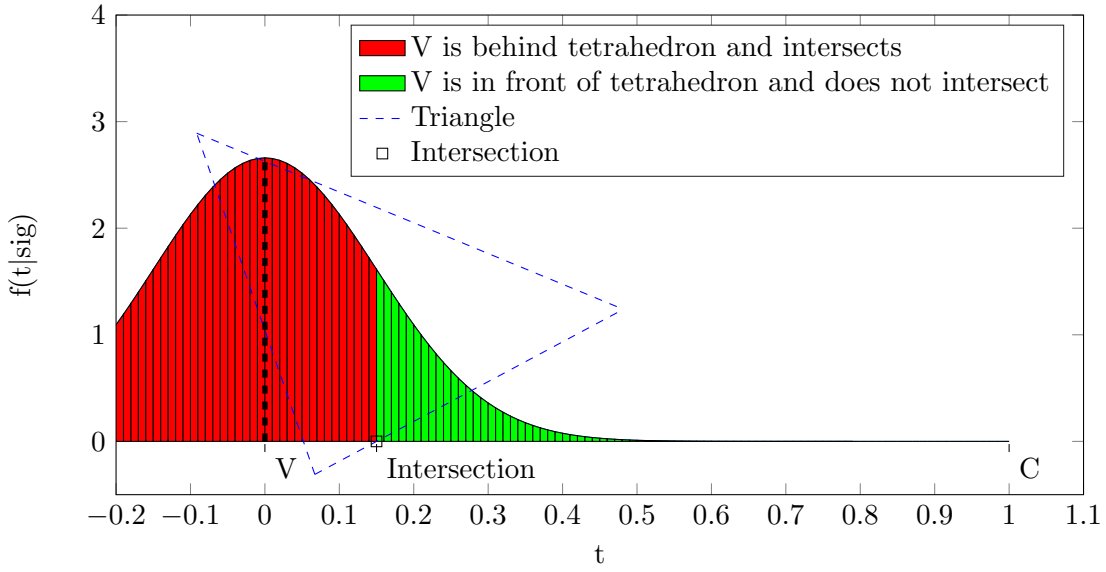


Figure 3.2: Zero Mean Normal PDF of true Vertex Location

The implications on the carving algorithm are such, that given the uncertainty of the vertex on the ray, an intersection with a tetrahedron has a certain uncertainty too. If for example the ray from the observed vertex to the camera intersects a tetrahedron on the region $0.05 \leq t \leq 0.15$, and the location of the true vertex were to be at 0.3, the intersection with the tetrahedron only occured due to the noise involved with the vertex's observation. In

this example, the ray intersects the tetrahedron if the true location of the vertex on the ray is behind 0.15. Figure 3.2 illustartes this example.

Given the location of the vertex is normally distributed with $t \sim N(0, \sigma^2)$, the probability of intersection in this example is the total probability that t is less that 0.15 and therefore $P(t < 0.15)$. And respectively the probability of a tetrahedron to not be intersected and remain part of the volume is $P(t > 0.15)$.

Moving away from the example to a general case where the intersection of the ray occurs at t_0 (instead of 0.15) and denoting $F(t|\sigma)$ the cumulative distribution function of a zero mean Gaussian with standard deviation σ, the probability of a tetrahedron remaining part of the volume given an intersection with a ray formed with an observed vertex is

$$\begin{aligned} P(t > t_0) &= 1 - P(t \leq t_0) = 1 - F(t_0|\sigma) \\ &= 1 - (1 - F(-t_0|\sigma)) \\ &= F(-t_0|\sigma). \end{aligned} \tag{3.1}$$

In reality, many rays intersect a tetrahedron, and by assuming N rays that intersect the tetrahedron are mutually independant, the total probability of a tetrahedron being part of the volume is

$$P_{\mathtt{T}} = \prod_{i=1}^{N} F(-t_i|\sigma_i). \tag{3.2}$$

Where t_i is the intersection coordinate on the ray i and σ_i si the standard deviation of the zero mean Gaussian distribution on the i^{th} ray.

3.2 Distribution Parameters

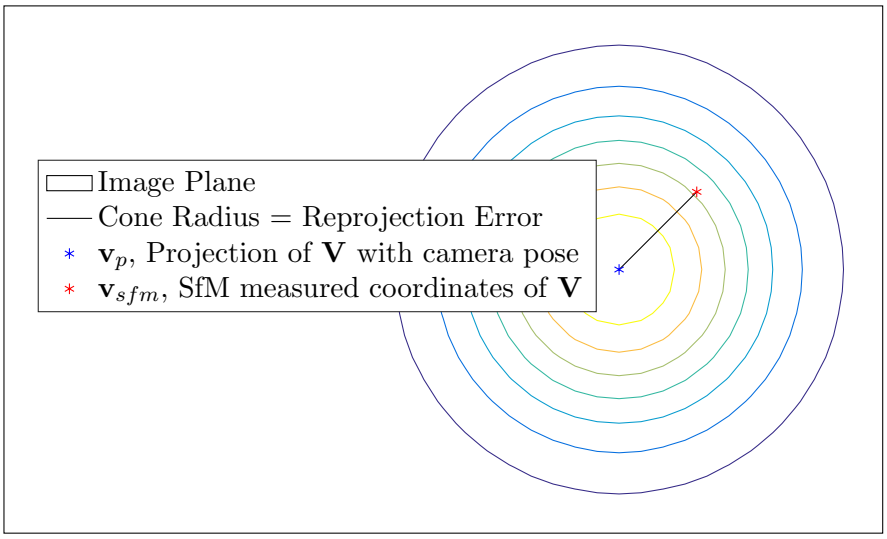


Figure 3.3: Gaussian Distribution on Image Plane

First, lets consider the uncertainty related with the reconstruction of a vertex $\mathbf{V}$. The reprojection of $\mathbf{V}$ on the image plane of a camera using the estimated pose does not match the measured SfM image coordinates. This suggests some uncertainty regarding the true projected coordinates of the vertex. For convenience reasons, the coordinate system used to define the distribution of $\mathbf{V}$ is a translation of the world coordinate system centred at $\mathbf{V}$. $\mathbf{V}$'s distribution in 3D space is modelled as a trivariate Gaussian with zero mean as $\mathbf{V} \sim N(\mathbf{0}, \Sigma)$. Samples to estimate the covariance matrix are taken from the intersections of the uncertainty cones with the rays. As usual, the covariance matrix is defined as

$$\Sigma = \begin{bmatrix} \sigma_x^2 & \Sigma_{xy} & \Sigma_{xz} \\ \Sigma_{xy} & \sigma_y^2 & \Sigma_{yz} \\ \Sigma_{xz} & \Sigma_{yz} & \sigma_z^2 \end{bmatrix}. \tag{3.3}$$

Given $\mathbf{V}_i$ are N measurements of $\mathbf{V}$, Σ is estimated as

$$\Sigma = \frac{1}{N} \sum_{i=1}^{N} \mathbf{V}_i \mathbf{V}_i^\top. \tag{3.4}$$

Lets denote as $\mathbf{v}_p$ and $\mathbf{v}_{sfm}$ the reprojected usng the camera pose and the SfM coordinates of $\mathbf{V}$ on the image plane. The distance between $\mathbf{v}_p$ and $\mathbf{v}_{sfm}$ is the reprojection error (see figure 3.2). The true projected coordinates $\mathbf{v}$ of the vertex must be a random variable with a certain distribution on the image plane (2D Gaussian). However, this distribution does not need to be known exactly. Instead, the only thing assumed is that $\mathbf{v}_p$ is the centre of the distribution, and that the circle with $\mathbf{v}_p$ as a centre and $\mathbf{v}_{sfm}$ at the radius defines an area on the image plane that is likely to contain observations of the true coordinates $\mathbf{v}$.

The circle on the image plane together with the camera centre defines a cone in 3D space, that is likely to contain the true ray, which is displayed on figure 3.4 in yellow. Given a ray with another camera viewing the same vertex $\mathbf{V}$, the cone intersects with this ray at two points, $\mathbf{A}$ and $\mathbf{B}$. As displayed on the figure, $\mathbf{W}$ denotes the world coordinates of $\mathbf{v}_{sfm}$, $\mathbf{V}$'s SfM pixel on the image plane. $\mathbf{W}$ is formed by rotating and translating $\mathbf{v}_{sfm}$ from camera coordinates to world coordinates. Given this geometry, the lengths of the vectors from $\mathbf{V}$ to $\mathbf{A}$ and from $\mathbf{V}$ to $\mathbf{B}$, $L_A = \|\mathbf{A} - \mathbf{V}\|$ and $L_B = \|\mathbf{B} - \mathbf{V}\|$ respectively can be derived to be

$$L_A = \frac{r(1)}{\sin\beta + \cos\beta \tan\alpha} \tag{3.5}$$

$$L_B = \frac{r(1)}{\sin\beta - \cos\beta \tan\alpha}. \tag{3.6}$$

Where the function $r(t)$ gives the radius of the cone on the circular section of the cone that cuts the ray $\mathbf{C}_1 + t(\mathbf{V} - \mathbf{C}_1)$ at a specific parameter t. The Radius is given by

$$\begin{aligned} D_W &= \frac{\|(\mathbf{V} - \mathbf{C}_1) \times (\mathbf{C}_1 - \mathbf{W})\|}{\|\mathbf{V} - \mathbf{C}_1\|} \\ t_W &= \frac{|(\mathbf{V} - \mathbf{C}_1) \cdot (\mathbf{C}_1 - \mathbf{W})|}{\|\mathbf{V} - \mathbf{C}_1\|^2} \\ r(t) &= \frac{D_W}{t_W} |t|. \end{aligned} \tag{3.7}$$

and closely linked to the world coordinates $\mathbf{W}$ of the SfM projection of $\mathbf{V}$ on the image plane. α is the angle between the cone's central axis and its outer surface, and β is the angle between the vectors $\mathbf{C}_1 - \mathbf{V}$ and $\mathbf{C}_2 - \mathbf{V}$ and are given more explicitly by

$$\begin{aligned} \tan\alpha &= \frac{r(1)}{\|\mathbf{C}_1 - \mathbf{V}\|} \\ \cos\beta &= \frac{(\mathbf{C}_1 - \mathbf{V}) \cdot (\mathbf{C}_2 - \mathbf{V})}{\|\mathbf{C}_1 - \mathbf{V}\| \, \|\mathbf{C}_2 - \mathbf{V}\|}. \end{aligned} \tag{3.8}$$

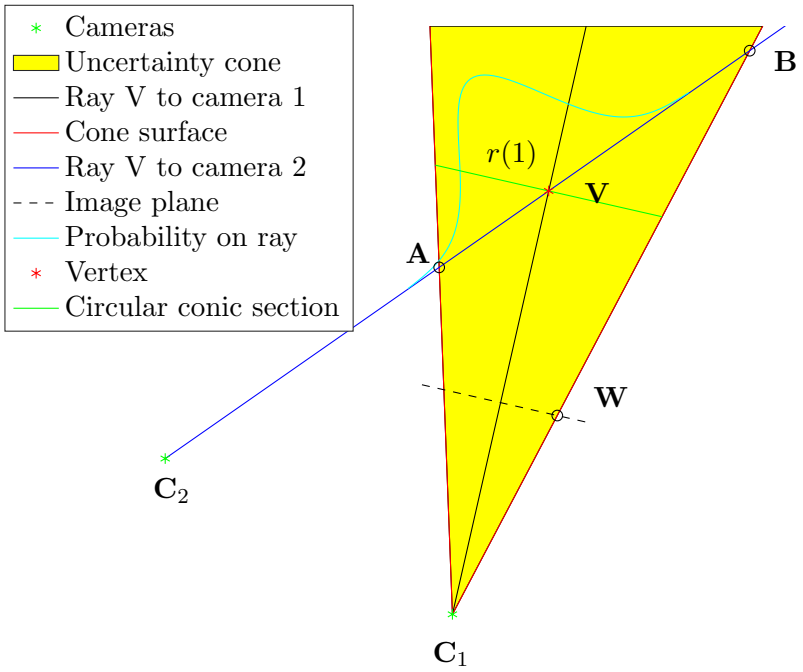


Figure 3.4: Uncertainty Cone Intersections Produce Samples **A** and **B**

Having L_A and L_B makes it possible do determine the location of the points **A** and **B** with

$$\begin{aligned} \mathbf{A} &= L_A \frac{\mathbf{C}_2 - \mathbf{V}}{\|\mathbf{C}_2 - \mathbf{V}\|} \\ \mathbf{B} &= L_B \frac{\mathbf{V} - \mathbf{C}_2}{\|\mathbf{C}_2 - \mathbf{V}\|}. \end{aligned} \tag{3.9}$$

The coordinates of **A** and **B** are given on a translated world coordinate system that is centred at **V**. Note, that **A** or **B** do not necessarily exist. The coordinates **A** and **B** can be used as data points to estimate the covariance matrix in equation (3.4). If there are more than two cameras viewing the vertex, more points can be used to estimate the covariance matrix. A cone of a camera creates two datapoints (one if **B** does not exist) on each of the rays from all other cameras viewing the vertex.

When tracing a ray, the probability distribution on the ray itself is used to described tetrahedra probabilities, i.e. probabilities, that tetrahedra are not empty. Univariate distributions on rays that intersect the centre can be derived from the 3D trivariate distribution.

In the trivariate distribution, the coordinates of a vector $\begin{pmatrix} X & Y & Z \end{pmatrix}^\top$ are correlated random variables. Any direction has components in X, Y and Z with weightings a_x, a_y and a_z respectively, such that the location of the vertex on a ray of that direction is a random variable $T = a_x X + a_y Y + a_z Z$. The variance $\sigma_T^2 = \mathrm{Var}[a_x X + a_y Y + a_z Z]$ of the linear combination of correlated random variables is given by

$$\sigma_T^2 = a_x^2\sigma_X^2 + a_y^2\sigma_Y^2 + a_z^2\sigma_Z^2 + 2a_x a_y \Sigma_{XY} + 2a_x a_z \Sigma_{XZ} + 2a_y a_z \Sigma_{YZ}. \tag{3.10}$$

Similarly as in principal component analysis, the weightings are normalized such that $a_x^2 + a_y^2 + a_z^2 = 1$. Then σ_T^2 gives the directional variance of the trivariate distribution. Note that for specific constellations of a_x, a_y, a_z, the principal components are formed and σ_T^2 is the variance of principal components. The directional variance is used to for the univariate Gaussian distributions of the vertex's true location on the rays. Given the ray from $\mathbf{V}$ to $\mathbf{C}_2$ in the geometry of figure 3.4, the weightings should be normalized as

$$\begin{pmatrix} a_x & a_y & a_z \end{pmatrix}^\top = \frac{\mathbf{C}_2 - \mathbf{V}}{\|\mathbf{C}_2 - \mathbf{V}\|} \tag{3.11}$$

in order to meet the unity condition $a_x^2 + a_y^2 + a_z^2 = 1$.

4 Graph Cut Optimization

The probabilistic formulation of the tetrahedra carving technique allows for a more accurate, probability based representation of the scanned volume. However, the process of reconstructing a surface has become more complex now, as unlike in the simple carving algorithm, tetrahedra are not simply solid or empty, but have a probability of being solid. A simple approach employed by ProFORMA [3] is to reconstruct the surface by carving tetrahedra that have probability smaller than a given threshold. This thesis proposes a global optimization instead. Particularly, an energy functional that considers costs associated with carving tetrahedra is formulated and minimized.

Furthermore, there is information available than only what is offered by the SfM pointcloud. The images of the video sequence are used to check for photoconsistency of triangles. The basic idea is that, if a triangle is part of the real surface, its projection on frames of cameras viewing the triangle should have to a certain the same content, referring to pixel colours. If a triangle's projection on different frames produces different colours, there is a high chance that it is not part of a real surface.

In this chapter, photoconsistency measures are quantitatively formulated and included in the global energy minimization. Tetrahedra carving is formulated as an energy functional and is shown how it can be efficiently minimized using graph cut algorithms.

4.1 Graph Cut Tetrahedra Labelling

4.1.1 Minimum Cuts for Solving Energy Minimization Problems

In the background section the min cut max flow theorem was described, that allows the minimum cut to be computed efficiently by finding the maximum flow. The minimum cut problem itself can be used to minimize a graph based energy function such as

$$E(\mathcal{L}) = \sum_{i \in \mathcal{V}} D_i(L_i) + \sum_{\forall (p,q \in \mathcal{V} \times \mathcal{V}) : L_p \neq L_q} E_{p,q}. \tag{4.1}$$

Where $\mathcal{V}$ is the set of all nodes and $\mathcal{L}$ is the set of labels for all nodes in the graph; every node is having one label that is either 1 or 0. $L_i \in \mathcal{L}$ denotes the label of node i. $D_i(L_i)$ is the data cost and describes the tendency of a node to be assigned to either one of the labels, i.e.

$$D_i(L_i) = \begin{cases} a_i, & \text{if } L_i = 1. \\ b_i, & \text{if } L_i = 0. \end{cases} \tag{4.2}$$

In the above expression (4.2) a_i and b_i are the cost associated with labelling node i as 1 and 0 respectively. Labelling i as 1 adds a_i to the total cost for example, and if $a_i < b_i$, then the node inherently has a tendency to be labelled 1.

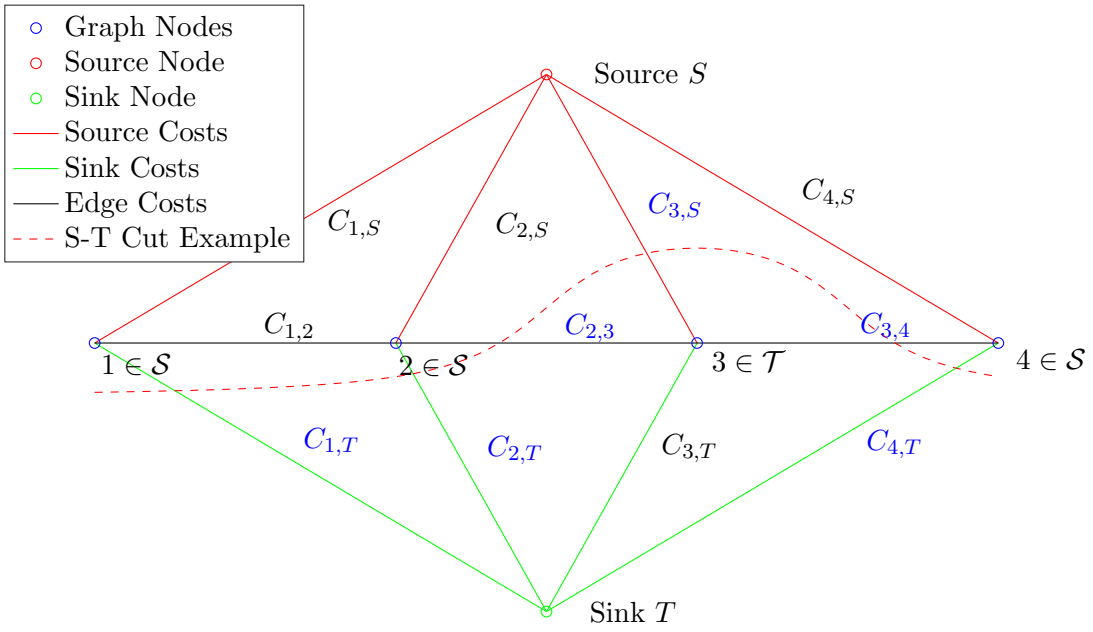


Figure 4.1: S-T Cut on Graph with Global Connectivity to S and T

Simultaneously, $E_{p,q}$ describe edge costs between neighbouring nodes that penalize neighbours having different labels. If two nodes $p \in \mathcal{V}$ and $q \in \mathcal{V}$ share an edge with very high $E_{p,q}$, it is likely that they will be labelled the same, even if their individual data costs have a tendency towards different labels. Edge costs in the graph formulation in this thesis are undirected, such that $E_{p,q} = E_{q,p}$.

Formulating (4.1) as a minimum cut problem can be done by introducing two additional nodes for the source and the sink, S and T respectively, which are connected to all other nodes in the graph. A graph cut partitions the graph into two non-intersecting sets of nodes $\mathcal{S}$ and $\mathcal{T}$. $\mathcal{S}$ is the set of all nodes that remain attached to the source and $\mathcal{T}$ is the set of all nodes that is attached to the sink. The source and sink are also part of these sets, such that $S \in \mathcal{S}$ and $T \in \mathcal{T}$. However, source and sink nodes S and T are not considered part of set of all nodes, $\mathcal{V}$.

A simple illustration of such an example is shown on figure 4.1, which also shows the capacities and graph partitioning of the cut; capacities in blue colour are contributing to the total cut capacity. Capacity values in the graph are loaded with the energy function's cost coefficients as indicated in equation (4.5). In this set-up, all nodes have an edge capacity to S and T, denoted as $C_{i,S}$ and $C_{i,T}$. The nodes still maintain their original edge capacities between each other, denoted as $C_{p,q}$ between two neighbouring nodes $p \in \mathcal{V}$ and $q \in \mathcal{V}$.

The minimum cut between the source and the sink minimizes the sum of all cut capacities. The cut separates every node from either S or T, and the capacity contribution to the total cut's capacity is

$$C_i = \begin{cases} C_{i,S}, & \text{if } i \in \mathcal{T}. \\ C_{i,T}, & \text{if } i \in \mathcal{S}. \end{cases} \tag{4.3}$$

When a cut assigns a node p to set $\mathcal{T}$ and thus separates it from the source S, while one of node p's neighbours, $q \in \mathcal{S}$, is getting separated from T, then the edge between these two nodes must also be part of the minimum cut and edge (p, q)'s capacity $C_{p,q}$ is added to the total minimum cut capacity. By adding all the capacities by pairs $(p, q) : p \in \mathcal{S} \setminus S, q \in \mathcal{T} \setminus T \vee p \in \mathcal{T} \setminus T, q \in \mathcal{S} \setminus S$ (i.e. undirected cut edges between nodes that are not S or T), together with the capacities from cutting edges to S or T described in (4.3) the total capacity C_{mc} of the minimum cut becomes

$$C_{mc} = \sum_{i \in \mathcal{V}} C_i + \sum_{p,q \in \mathcal{E}} C_{p,q}. \tag{4.4}$$

Here $\mathcal{E}$ is the set of all cut edges between nodes other than S or T, which can be expressed as $\mathcal{E} = \{(p, q \in \mathcal{V} \times \mathcal{V}) : p \in \mathcal{T}, q \in \mathcal{S} \vee p \in \mathcal{S}, q \in \mathcal{T}\}$

A comparison of equations (4.1) and (4.4) shows that the format is essentially the same. Capacities to the source and sink nodes S and T correspond to the data costs of choosing a label. If nodes in $\mathcal{T}$ are chosen to represent nodes labelled 1, and nodes in $\mathcal{S}$ would represent nodes labelled 0. Therefore, assigning the capacities to source and sink as $C_{i,S} = a_i$ and $C_{i,T} = b_i$ leads to $C_i = D_i(L_i)$.

When formulating the energy functional of equation (4.1) as a graph, capacities $C_{p,q}$ between nodes $(p \in \mathcal{V}, q \in \mathcal{V})$ correspond to edge costs $E_{p,q}$ between the nodes $(p \in \mathcal{V}, q \in \mathcal{V})$. Just like $E_{p,q}$ are costs associated with neighbours having a different label, $C_{p,q}$ are capacities that contribute to the total minimum cut cost C_{mc} if p and q belong to different sets $\mathcal{S}$ or $\mathcal{T}$.

The equations in (4.5) summarize how the graph has to be constructed so that an energy minimization problem as given by (4.1) can be efficiently solved using a maxflow algorithm (since equivalent to min-cut due to max flow min cut theorem).

$$\begin{aligned} C_{i,S} &= a_i \\ C_{i,T} &= b_i \\ C_{p,q} &= E_{p,q}. \end{aligned} \tag{4.5}$$

Loading the graph with these capacities and solving the minimum cut problem, the total min-cut capacity is equivalent to the minimized cost of the energy function $C_{mc} = E(\mathcal{L})$. The partitioning of set $\mathcal{V}$ into $\mathcal{S}$ and $\mathcal{T}$ provides the node labelling $\mathcal{L}$ that minimizes the energy function of (4.1).

4.1.2 Graph-Cut Energy Minimization on Tetrahedral Grid

Tetrahedra in the triangulation have up to four neighbours, hence by expressing tetrahedra as graph nodes, nodes can have up to four undirected edges connecting them with other nodes. The graph is slightly more difficult to visualize than the simple illustration in figure 4.1, as there are more graph edges. Figure 4.2 shows the graph connectivity of a 2D Delaunay triangulation. The main property, that allows to efficiently express an energy minimization in the form (4.1) still holds; if neighbouring nodes are labelled differently, the edge between them is added to the total min-cut capacity and hence to the total energy. In figure 4.2, the yellow shaded area separates nodes that remain attached to the sink from nodes that remain attached to the source after the graph cut.

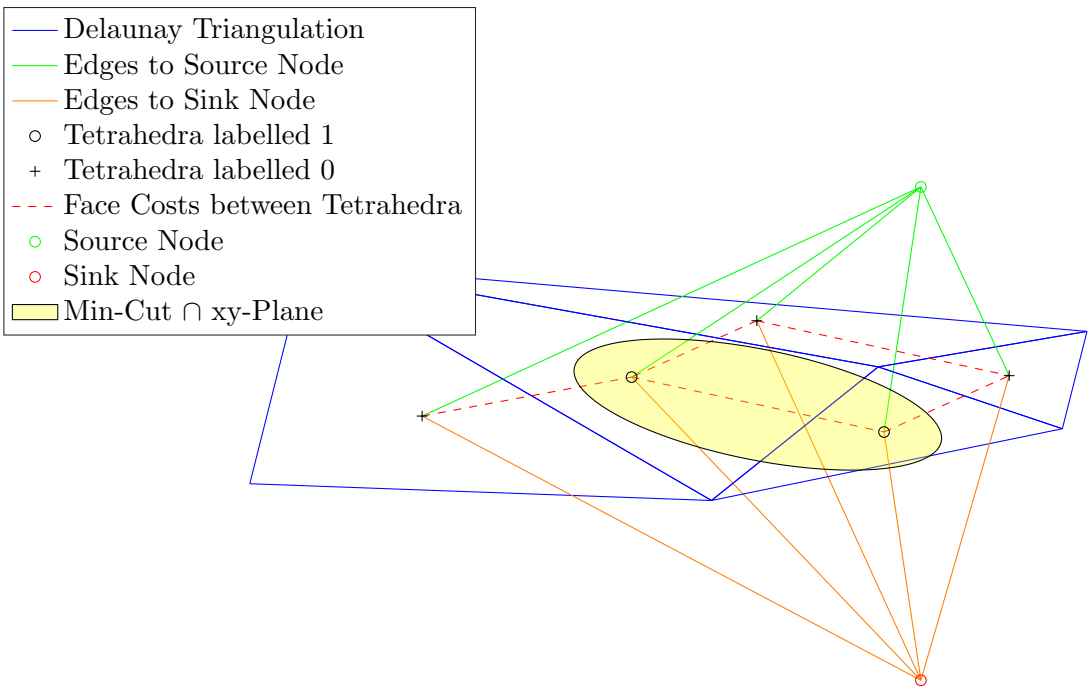


Figure 4.2: Set up for Max Flow Min Cut on Delaunay Grid

The grid given by the Delaunay triangulation is used to form a graph, where every tetrahedron is represented by a node and every face that is shared between two tetrahedra is the graph edge between the nodes corresponding to the tetrahedra.

The labelling problem of the tetrahedra in the Delaunay triangulation can be written as an graph based energy potential in the form of equation (4.1) by using the tetrahedra probabilities as the data costs and edge costs based on desirable properties of surface triangles such as photoconsistency and small surface area.

$$E(\mathcal{L}) = \sum_{\mathtt{T} \in \mathcal{D}_\mathcal{T}} D_C(\mathtt{T}) + \sum_{\mathtt{F} \in \mathcal{D}_\mathcal{F}} F_C(\mathtt{F}). \tag{4.6}$$

Where $\mathtt{F} \in \mathcal{D}_\mathcal{F}$, is the common face between the two tetrahedra with different labels; $\mathcal{D}_\mathcal{F}$ denoting the set of all faces between tetrahedra with different labels. The second summation sums all faces that are between tetrahedra with different labels. $\mathcal{L}$ is the set of labels for all tetrahedra in the triangulation, one label for each tetrahedron; the labels that are used are are L_{in} and L_{out}, for tetrahedra that are part of the inside and tetrahedra that are part of the outside volume. The set of all tetrahedra in the Delaunay triangulation is denoted $\mathcal{D}_\mathcal{T}$. $L(\mathtt{T}) \in \{L_{in}, L_{out}\}$ denotes the label for tetrahedron ${}^{i}\mathtt{T}^{4}$. The data cost is the probability of a tetrahedron, such that

$$D_C(\mathtt{T}) = \begin{cases} 1 - P(\mathtt{T}), & \text{if } L(\mathtt{T}) = L_{in}. \\ P(\mathtt{T}), & \text{if } L(\mathtt{T}) = L_{out}. \end{cases} \tag{4.7}$$

Hence, tetrahedra with a probability > 0.5 are more favourable to become part of the solid volume (i.e. labelled L_{in}); and as it intuitively makes sense, the higher the probability the

stronger the attachment to the L_{in} label and the lower the cost for labelling the tetrahedron L_{in}.

The face-costs are denoted by $F_C(\mathtt{F})$ for a given face $\mathtt{F}$. The faces between tetrahedra are the candidates for becoming part of the reconstructed surface.

4.2 Face Costs

The face costs are made up by two components, the photoconsistency cost P_C and the area cost A_C, which are added to form the total face cost $F_C = P_C + A_C$.

4.2.1 Area Cost

This is the simpler of the two costs and has a piecewise linear relationship given in (4.9) to the surface area of individual faces. The surface area of a face, given its three vertices $\mathbf{F}_0$, $\mathbf{F}_1$ and $\mathbf{F}_2$ can be simply computed by calculating half the length of the cross product of any two of the triangles plane defining vectors (half because the area of the triangle is half the area of the parallelogram).

$$A = \frac{1}{2} \left\| (\mathbf{F}_1 - \mathbf{F}_0) \times (\mathbf{F}_2 - \mathbf{F}_0) \right\|. \tag{4.8}$$

At every new frame the grid changes and so does the reconstructed surface. Given the mean A_μ and standard deviation A_σ of the area of the individual triangles of the surface reconstructed at the previous frame (at this point it still is the most recent surface available), the area cost of a face is defined as

$$A_C = \begin{cases} \frac{1}{3A_\sigma}(A - (A_\mu + 3A_\sigma)), & \text{if } A > A_\mu + 3A_\sigma. \\ 0, & \text{else.} \end{cases} \tag{4.9}$$

For the very first frame there is no reconstructed surface available yet, and in this case all area costs are set to $A_C = 0$. From the second iteration onwards it is extremely unlikely and practically impossible to not have a single reconstructed triangle, so equation (4.9) can be used to calculate the area costs. In the very unlikely event however that there can be no surface reconstructed at all after the previous frame, the costs are assigned $A_C = 0$, as in the first iteration of the reconstruction process.

The equation in (4.9) is motivated by the observation, that the vast majority of the triangles inspected at a reconstruction without any area cost penalties have a significantly smaller surface area than some of the very distinct artefacts. Generally, artefacts that can be removed by introducing the area cost have a very large surface area compared to other triangles and generally appear in regions of the scene that are not explored well. Most triangles are not like that however and only triangles with large surface areas should be penalized.

Thus, triangles, that have an area less than three standard deviations more than the mean area A_μ, are not penalized and only the few ones that exceed the $A > A_\mu + 3A_\sigma$ threshold are penalized. Normalizing the excess area by $1/3A_\sigma$ makes sure that area penalties, compared to the data penalties calculated by tetrahedra probabilities and given by (4.7), are neither too large for only small excess area, neither too small for a large excess area.

4.2.2 Idea of Photoconsistency Costs

The basic idea when introducing photoconsistency costs, is, that if a triangle becomes part of the reconstructed surface, then the colours of the projections of the triangle to the frames of the cameras that can see it should be very similar. The photoconsistency cost is a measure of the colour difference among the projections of the triangle's centroid to all cameras that can see at least one of the triangle's vertices. Since for colour differences on pixel level the computations tend to be strongly affected by noise, prior to forming the photoconsistency cost, the images are blurred and the colourspace converted, which is explained in the following to sections in more detail.

4.2.3 Image Blurring

A very effective, yet very simple option for reducing image noise is to blur the images with an average filter. For this, the colour at every pixel is simply replaced with the average colour of all pixels from a square patch given a radius R. Consider the image to be represented by a $H \times W$ matrix of colours, where W is the picture's width and H is the pictures height, and indexing starts at 0. The colours imported are given by their RGB (i.e. Red, Green, Blue) values and a pixel is represented as a vector $\mathbf{p}_{i,j} = \begin{pmatrix} R & G & B \end{pmatrix}^\top$.

Averaging around a square patch with radius R means that every pixel $\mathbf{p}_{i,j}$ is replaced by the average colour of all pixels $\mathbf{p}_{p,q}$ with $p \in (i-R, i+R)$ and $q \in (j-R, j+R)$. Averaging is implemented with the relevant OpenCV [6] function which uses reflected boundaries such that an index of -1 for example returns the same colour as accessing index 1. Averaging while using reflected boundaries can be expressed as replacing every pixel of the image by

$$\begin{aligned} \mathbf{p}_{p,q} &= \frac{1}{(2R+1)^2} \sum_{i=p-R}^{R} \sum_{j=q-R}^{R} \mathbf{p}_{f_H(i), f_W(j)} \\ f_L(k) &= \begin{cases} k, & \text{if } 0 \leq k < L. \\ -k, & \text{if } k < 0. \\ 2L-k-2 & \text{else.} \end{cases} \end{aligned} \tag{4.10}$$

The visual effect of averaging around a patch is blurring the image, the colours however show much less variation as noise induced offsets with different signs cancel each other out during the averaging. For the purpose of reducing the amount of noise a relatively wide range of radii provide good results, so that it is sufficient to choose a radius of $R = 5$ and not try to optimize R in code. The blurring functionality that is available in OpenCV is used to compute (4.10).

4.2.4 Conversion to L*a*b* colorspace

An approach that comes immediately to mind is to use the RGB values and compute the colour difference of two pixels as the Euclidean distance between their RGB values. Although this gives some sort of measure of colour difference, the better approach is to convert the image to the L*a*b* colour space, which is designed with the intention to represent colours in a more intuitive manner. The particularly useful property of converting to the L*a*b* colour space is that the colour difference expressed in L*a*b* colours is much more accurate.

The conversion expressions that convert from RGB to L*a*b*, with an intermediate conversion to the XYZ colour space, are [6]

$$\begin{aligned}
\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} &\leftarrow \begin{bmatrix} 0.4340 & 0.3762 & 0.1898 \\ 0.2127 & 0.7152 & 0.0722 \\ 0.0178 & 0.1095 & 0.8728 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \\
L^* &\leftarrow \begin{cases} 116Y^{1/3} - 16, & \text{if } Y > 0.008856. \\ 903.3Y, & \text{else.} \end{cases} \\
a^* &\leftarrow 500(f(X) - f(Y)) + 128 \\
b^* &\leftarrow 200(f(Y) - f(Z)) + 128 \\
f(t) &= \begin{cases} t^{1/3}, & \text{if } t > 0.008856. \\ 7.787t + 16/116, & \text{else.} \end{cases}
\end{aligned} \tag{4.11}$$

All pixels of the image, prior to being averaged by the operation in (4.10), are converted to the L*a*b colour space (reconstructions shown in the last chapter are done with an implementation that first averages and then converts to L*a*b*; visually there is no noticeable difference however). For the conversion, the OpenCV functionality was used that implements exactly the RGB to L*a*b* conversion described in (4.11). Now, the pixels instead of containing RGB values are $\mathbf{p}_{i,j} = \begin{pmatrix} L^* & a^* & b^* \end{pmatrix}^\top$. In the L*a*b* colour space, the Euclidean distance between the colour of pixels $\|\mathbf{p}_{i,j} - \mathbf{p}_{p,q}\|$ is a much more accurate measure for the colour difference between two pixels, even more so between two pixel's on a picture that has been average-filtered.

4.2.5 Photoconsistency Costs

Figure 4.3 illustrates how badly reconstructed triangles can be removed by introducing the potoconsistency cost. The real object's volume is represented by the grey shaded area and the object's real surface is represented by the colourful perimeter. If there are none or only a few rays only that are intersecting the volume with concavities, then some tetrahedra in it are labelled part of the solid volume, although they should be ideally labelled as empty space. Similar situations arise if there are hollow spaces within the object (like for example in the temple used for the experiments) or if two different objects are placed close together in the real world scene. A sensible method to avoid such occurrences, is to measure the photoconsistency of triangles among the different frames that can see them. In the figure, the rays to the centroid of triangles is projected onto the different frames, and large difference in the colour indicates a badly reconstructed triangle.

Every triangle in the Delaunay grid is a candidate to become part of the reconstructed surface. The reconstructed surface should ideally be matching the real surface of the object being scanned. Of course, the real surface is not made by triangles, but reconstructed triangle should be ideally small enough and located correctly in the scene such that the real surface is planar enough and well represented by the triangle. Simultaneously, based on the knowledge that the vertices in the triangulation are reconstructed from features, which generally are formed at distinct points such as edges or high texture gradients and regions with high variability, the space between vertices that is part of the scanned object's real surface, has less variability in texture and structure that the vertices themselves, otherwise there would

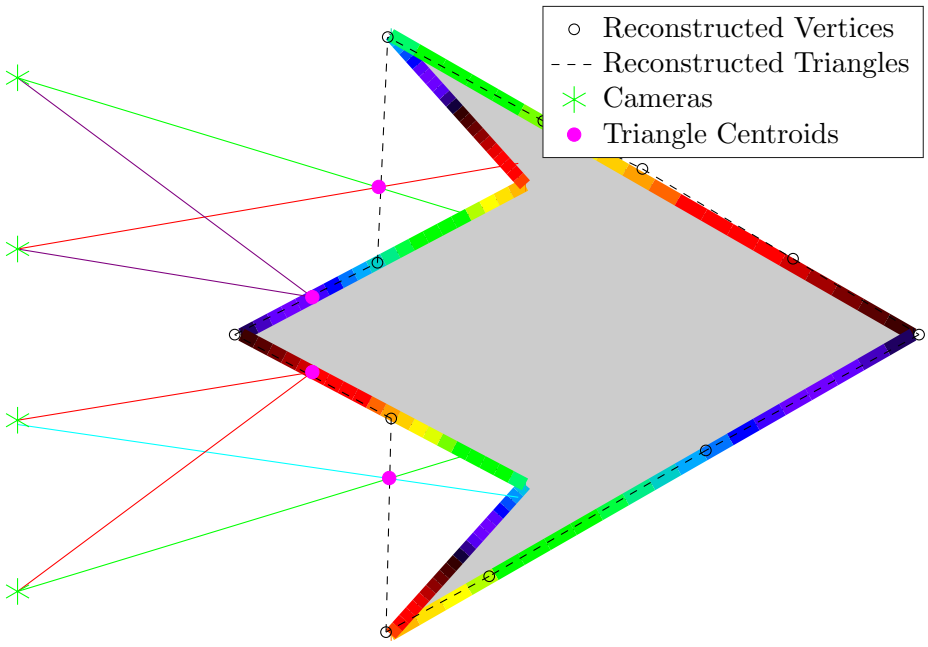


Figure 4.3: Motivation for using Photoconsistency Costs

have been more feature points formed in that space. With this in mind, the texture inside the body of the triangles that approximate the real surface has relatively uniform colour.

The colour at the centroid of the triangle can be assumed to be a good representation of the colour of the entire triangle, especially now that the images have been averaged, accessing the colour on a single pixel not only contains less noise that the original pixel colours, but is also implies that the colour value read comes from a larger patch than a single pixel on the image (and due to the conversion to L*a*b* space the colour differences are more accurately represented by Euclidean distances). If a triangle exists in the real 3D world, the projection of it on the various frames should be containing the same structural content, and the same colour.

Although theoretically if the object is sufficiently reflective and illumination comes from unfortunate angles, the colour perceived in different frames could show large variation, in practice, given exposure time or similar settings of the camera have not been changed from one frame to another, the projection of a real triangle on the different frames will be essentially having the same colour.

If the triangle is badly reconstructed and does not exist in the real scene, its projection on different images will contain different texture and colour. Figure 4.3 illustrates this by colouring rays from cameras to centroids of reconstructed triangles with the perceived colour. For badly reconstructed triangles the perceived colour is different among the projections on different frames.

A triangle can in general be assumed to be visible by all cameras that can see its individual vertices. Every vertex in the SfM pointcloud is visible from at least two cameras, so a triangle is at least visible by two cameras and hence a difference of colours can be measured as $\|\mathbf{p}_{i,j} - \mathbf{p}_{p,q}\|$. In the rare case that the triangle is not visible by at least two camera it can

be assumed that the reason for that is that it is a badly reconstructed triangle with such a large area or at such a remote location that it is not seen by at least two cameras.

For valid triangles, the colour difference can be expected to be reasonably low. However, when it comes to badly reconstructed triangles, that form reconstruction artefacts, given enough diversity in the background, the colour differences can be expected to be significantly larger than for valid triangles. This is because for wrong triangles, even if the vertices are well reconstructed (think of the space between two columns) the centroid of the triangle is not. Projecting the world coordinates of the centroid of a wrong triangle on the camera frames is in fact projecting the physical material that is located at different world coordinates. The light that is forming the pixel at the centroid's projection is the light that comes from a point on the real world, whose world coordinates are such, that the line between the light origin and the camera centre intersected the world coordinates of the centroid.

The pixel formed by the projection of the centroid onto the different frames is therefore a different point in real space for every frame. If the background is non-uniform and provides sufficient colour variability, which in most outdoor and indoor environments it does, the perceived colours on each camera are very different leading to a high $\|\mathbf{p}_{i,j} - \mathbf{p}_{p,q}\|$ when computed. This is illustrated on figure 4.3, where the perceived colours of the centroid's projection can be seen on the colour of the line segments between the cameras and the scanned object; the perceived colours are different.

Since there are usually more than two cameras viewing a triangle, a reasonable formulation of the photoconsistency cost is therefore one, that considers the root mean square distance to the average colour (weighted RMS of weighted average colour in fact) of the centroid of the triangle projected on all frames whose cameras can see the triangle (i.e. can see at least one of its vertices and the projection of the triangle centroid is within the image boundaries). Such an expression can be formulated as

$$\begin{aligned} P_C &= \sqrt{\frac{1}{N}\sum_{i=1}^{3}\sum_{c\in\mathcal{C}_i}\|\mathbf{p}_c - \mathbf{p}_\mu\|^2} \\ \mathbf{p}_\mu &= \frac{1}{N}\sum_{i=1}^{3}\sum_{c\in\mathcal{C}_i}\mathbf{p}_c \\ N &= |\mathcal{C}_1| + |\mathcal{C}_2| + |\mathcal{C}_3|. \end{aligned} \tag{4.12}$$

Here, $\mathcal{C}_i$ denotes the set of cameras that can see the triangle's i^{th} vertex, and the vector of colours of the triangle's centroid projection on the c^{th} camera of a set $\mathcal{C}_i$ is denoted $\mathbf{p}_c$. $|\mathcal{C}_i|$ denotes the cardinality of the set. The first sum iterates over the 3 vertices of the triangle and the second sum iterates over all cameras that can see the triangle. The expression (4.12) implies, that if a camera sees more than one of the triangle's vertices, the colour contribution made by the projection of the triangle's centroid on that camera's frame is weighted 2 or 3 times more than the projection on a camera's frame that sees only one of the vertices. It makes sense to use this weighting and have cameras that see more vertices have a stronger impact on forming the triangle's photoconsistency cost. Furthermore, the weighted photoconsistency cost definition can be implemented more easily, since the vertices are treated independently and all cameras seeing an individual vertex are used, in contrast to first having to check if a camera in $\mathcal{C}_i$ has already been processed when $\mathcal{C}_{i-1}$ was processed. This also helps for the online computation of P_C described in the next section.

4.2.6 Online Computation of Photoconsistency Costs

When running the incremental algorithm, the visibility of vertices changes, as new cameras are viewing existing vertices. This would mean that the photoconsistency cost of the triangles adjacent to that vertex needs to be recalculated, as the set of cameras increases. The expression of the photoconsistency cost as formulated in (4.12) would require recalculation from scratch (or almost from scratch if $\mathbf{p}_c$ are retained, which is also not very efficient).

Calculating the weighted average colour $\mathbf{p}_\mu$ incrementally can be done easily. However, there is no straightforward way to implement online computation of P_C, due to the square root. Instead, $\tilde{P_C} = NP_C^2$ can be efficiently defined in an incremental fashion, and the photoconsistency cost could then simply be calculated as $P_C = \sqrt{(\tilde{P_C})/N}$. $\tilde{P_C}$ is essentially the sum of all $\|\mathbf{p}_c - \mathbf{p}_\mu\|^2$. Due to the incremental fashion, camera views of one of the triangle's vertices are included one at a time, such that projections of the centroid can be more conveniently expressed in terms of a $3 \times n$ matrix $\mathbf{p}^n = \begin{bmatrix}\mathbf{p}^1 & ... & \mathbf{p}^n\end{bmatrix}$ (which is only shown here to illustrate the concept and is not actually stored in memory), where n is the iteration number , and here $\mathbf{p}^n \in \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3$ is the colour of the projection on the n^{th} iteration of computing the photoconsistency cost. Using this notion it is more convenient to show that $\tilde{P_C}$ can be alternatively expressed as

$$\begin{aligned}\tilde{P_C^n} &= \sum_{i=1}^{3}\sum_{c\in\mathcal{C}_i}\|\mathbf{p}_c - \mathbf{p}_\mu\|^2 = \sum_{i=1}^{n}\left\|\mathbf{p}^i - \mathbf{p}_\mu\right\|^2 \\ &= \sum_{i=1}^{n}(L^{*i} - L^{*\mu})^2 + \sum_{i=1}^{n}(a^{*i} - a^{*\mu})^2 + \sum_{i=1}^{n}(b^{*i} - b^{*\mu})^2.\end{aligned} \tag{4.13}$$

Superscripts denote the sequence numbering. Using for all three sums of (4.13) a general equivalence expression; $\bar{x}_n$ being the mean of the first n elements,

$$\sum_{i=1}^{n}(x_i - \bar{x}_n)^2 - \sum_{i=1}^{n-1}(x_i - \bar{x}_{n-1})^2 = (x_n - \bar{x}_{n-1})(x_n - \bar{x}_n). \tag{4.14}$$

$\tilde{P_C}$ can be implemented incrementally for $n > 1$ as

$$\begin{aligned}\mathbf{p}_\mu^n &= \frac{1}{n}((n-1)\mathbf{p}_\mu^{n-1} + \mathbf{p}^n) \\ \tilde{P_C}^n &= \tilde{P_C}^{n-1} + (\mathbf{p}^n - \mathbf{p}_\mu^{n-1}) \cdot (\mathbf{p}^n - \mathbf{p}_\mu^n).\end{aligned} \tag{4.15}$$

For $n = 1$, there is only one colour available, hence the RMS distance of all colours to the mean is the distance of the one available colour to itself, and is hence zero.

For the online calculation of $\tilde{P_C}$, the only variables that need to be retained are $\mathbf{p}_\mu$ and n and everything else can be discarded after the update, so only a small amount of memory is actually needed for computing the photoconsistency costs online. Another benefit with using the expressions of (4.15) is that no computations have to be recalculated for updating $\tilde{P_C}$, leading to good computational performance.

When the actual photoconsistency cost of a triangle is needed, $\tilde{P_C}$ is used to return $P_C = \sqrt{(\tilde{P_C})/N}$. In order to save even more computational time, a flag is used to indicate if P_C

is up-to-date. If the flag is 1 when P_C is needed, the stored value of P_C is simply returned, otherwise P_C is computed first and then returned, while also setting the flag to 1 so that the next attempt to retrieve P_C does not need to be recalculated again. This way, redundant square root operations are avoided.

When visibility to a vertex $\mathbf{V}$ becomes available, the impacted triangles that need a photoconsistency update are all adjacent triangles to the vertex. Since the vertex is aware of the adjacent tetrahedra, the 3 of the 4 faces of the adjacent tetrahedra, which are adjacent to $\mathbf{V}$, are isolated and updated as expressed in (4.15).

When the point in code is reached, where the volume is about to be re-segmented by means of the graph based energy minimization, the photoconsistency costs are normalized in a similar way as the triangle surface area in equation (4.9). A similar equation is used to normalize the photoconsistency costs, with the main difference to equation (4.9) being that only one standard deviation is used instead of three. This is motivated by the fact, that a lot more artefacts have distinctively larger photoconsistency costs than the valid triangles, compared to the triangle area, which is more similar between valid and badly reconstructed triangles. The expression that modifies the photoconsistency costs, prior to loading them into the graph cut framework, uses the unmodified mean and standard deviation of the photoconsisitency costs of the surface reconstructed at the privous reconstruction phase, both of them denoted as Φ_μ and Φ_σ. Similarly to (4.9), the photoconsistency costs are modified by

$$P_C \leftarrow \begin{cases} \frac{1}{\Phi_\sigma}(P_C - (\Phi_\mu + \Phi_\sigma)), & \text{if } P > \Phi_\mu + \Phi_\sigma. \\ 0, & \text{else.} \end{cases} \tag{4.16}$$

Similarly to the normalized area costs, the normalized photoconsistency costs have no historic values in the first iteration and are all set to zero.

5 Spike Removal

After the graph cut based tetrahedra carving process partitions the Delaunay triangulation to solid and empty tetrahedra, the step proposed in this chapter aid to further improve the quality of the reconstructed surface by removing remaining spiky artefact formations. Solid angles are used as spikiness measure for every vertex. Vertices, which span the reconstructed volume at a sharp solid angle are considered to be tips of spikes and the spike formations are cleared by carving adjacent tetrahedra that contribute to the sharp solid angle.

5.1 Spike Identification

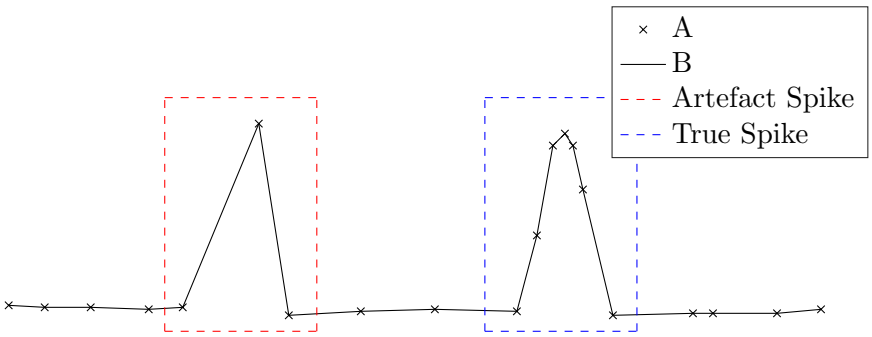


Figure 5.1: Spiky Artefact vs. Real Spike

When computing the optimal labelling for the tetrahedra with the graph cut algorithm, the volume is still not completely free of artefacts. The distinct and visible artefacts that are left in the reconstructed volume are in general very spiky. Spiky regions can of course exist in the real volume as well, however, the geometry differs. When the volume around individual vertices is observed, the solid volume reconstructed is usually spanning a large angle, even for (true) spiky regions. True spiky regions are generally reconstructed by having multiple vertices located at the sharp region, and in contrast, spiky artefacts usually are made up by a single vertex that was not reconstructed well and thus is not on the same plane as neighbouring vertices. Figure 5.1 is illustrating this in reduced dimension with a line representing the reconstructed surface. It is a sensible approach therefore to scan for spikes locally in the neighbourhood around individual vertices, considering only adjacent tetrahedra.

In the Delaunay grid, the spikiness of a vertex is determined by all adjacent tetrahedra with a solid label that are adjacent to the vertex. Figure 5.2 illustrates these by drawing the edge vectors from the vertex to the vertices of opposite faces of all solid adjacent tetrahedra.

Given that none of the adjacent tetrahedra on a vertex have share the same volume, in a 2D triangulation, the total angle spanned by all solid adjacent triangles would be an intuitive measure of the spikiness of the 2D volume. In 3D, the equivalent measure is the solid angle.

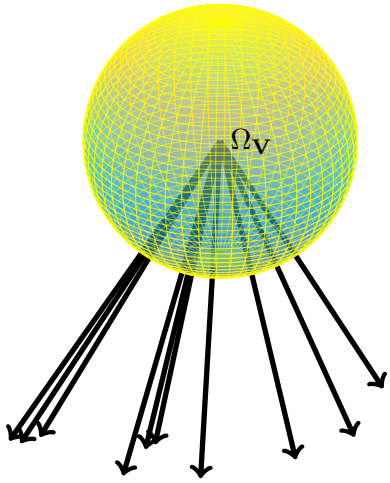


Figure 5.2: Spike and Total Solid Angle on a Vertex

A useful property that is the 3D version of the analogous 2D angle relation of angle and arc, is that the area spanned on a sphere of radius R by a solid angle Ω is simply given by ΩR^2.

To form the total solid angle $\Omega_{\mathbf{V}}$ spanned by the volume on a vertex $\mathbf{V}$, the solid angles formed with all solid labelled tetrahedra that are adjacent to $\mathbf{V}$ are added together. Out of the four faces of a tetrahedron, the thee are adjacent to $\mathbf{V}$ and span $\mathbf{V}$'s solid angle within the tetrahedron. On a sphere with sufficiently small radius R that is centred on $\mathbf{V}$, the tetrahedrons faces are intersecting with the sphere's surface forming a spherical triangle. As the name suggest, a spherical triangle is formed by pairwise intersection of 3 great arcs of a sphere and is the spherical analog of the planar triangle. Given the three vertices of a spherical triangle and the sphere's centre, the surface area Δ of the triangle is given by

$$\Delta = R^2\Omega \tag{5.1}$$

$$\Omega = A + B + C - \pi. \tag{5.2}$$

Where A, B and C denote the 2D angles between the sphere's great arcs on the spherical triangle's vertices, and Ω denotes the spherical triangle's solid angle. On figure 5.3, A, B and C are the angles of the spherical triangle on $\mathbf{T}_A$, $\mathbf{T}_B$ and $\mathbf{T}_C$ respectively. Ω is the spherical angle of $\mathbf{V}$ within the tetrahedron.

An angle between two great arcs is given by the angle between the oriented planes that contain the great arc. The orientation of the planes has to be towards the same direction with respect to the spherical triangle, i.e. either both normals are pointing towards the spherical triangle, or both are pointing away. The geometric setup is illustrated in figure 5.3, where a single tetrahedron is shown, along with the spherical triangle and the surface normals.

An easy and efficient way to ensure the triangles are oriented towards the same direction, is use the common edge in the computation of the normal for both triangles. The computation of the triangle normals ensuring same orientation is illustrated in figure 5.3 and by knowing the right hand rule for computing cross products, the implication of choice of vectors on the triangles and the resulting orientation of the triangle normals is straight forward.

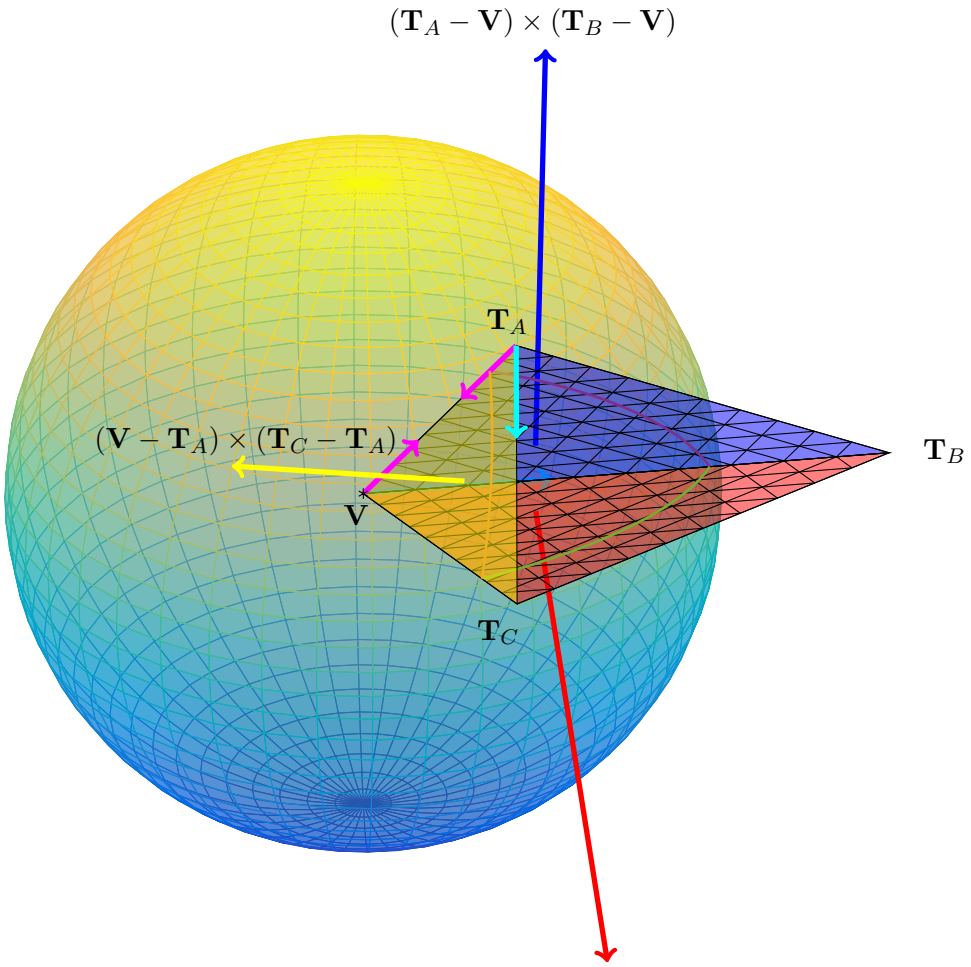


Figure 5.3: Geometry for calculating solid angles within a Tetrahedron

The angle between the planes that opens the spherical triangle (and hence between the great arcs at the intersection vertex) is π minus the angle between the oriented triangle normals, and can be calculated by means of the dot product of the triangle normals as

$$\begin{aligned} \mathbf{N}_1 &= (\mathbf{T}_A - \mathbf{V}) \times (\mathbf{T}_B - \mathbf{V}) \\ \mathbf{N}_2 &= (\mathbf{V} - \mathbf{T}_A) \times (\mathbf{T}_C - \mathbf{T}_A) \\ A &= \pi - \operatorname{acos}\left(\frac{\mathbf{N}_1 \cdot \mathbf{N}_2}{\|\mathbf{N}_1\| \, \|\mathbf{N}_2\|}\right). \end{aligned} \tag{5.3}$$

Where A is the angle of the spherical triangle at vertex $\mathbf{T}_A$. Because the individual orientation of the triangles is not important, as long as they have the same orientation, $\mathbf{T}_B$ and $\mathbf{T}_C$ can be switched and the result would still remain the same. Therefore for computing the angles B and C, in the expression (5.3), $\mathbf{T}_A$ is replaced with $\mathbf{T}_B$ or $\mathbf{T}_C$ respectively, and $\mathbf{T}_B$ and $\mathbf{T}_C$ are replaced with the tetrahedron vertices that are not used for $\mathbf{T}_A$ or $\mathbf{V}$.

The spherical angles computed with (5.3) can be used in equation (5.1) to compute the solid angle of a vertex within a tetrahedron. Lets denote $\Omega_{\mathbf{V}}^{\mathtt{T}}$ the solid angle of a vertex $\mathbf{V}$ within a tetrahedron $\mathtt{T}$ ($\Omega_{\mathbf{V}}^{\mathtt{T}}$ only defined if $\mathbf{V}$ is a vertex of $\mathtt{T}$). The total solid angle of a vertex is given by the sum of all solid angles of that vertex within its non-carved adjacent tetrahedra. Compactly, this can be written as

$$\Omega_{\mathbf{V}} = \sum_{\substack{\forall \mathtt{T}: \\ L(\mathtt{T}) = L_{in}, \\ \exists! f: \mathbf{T}_f = \mathbf{V}}} \Omega_{\mathbf{V}}^{\mathtt{T}}. \tag{5.4}$$

Since the graph cut energy minimization is implemented on tetrahedra level, and the formulation of the total solid angle as a spike measure is with respect to vertices it is not possible to implement a spikiness measure in the energy minimization term given in (4.6). Instead, the solid angles are used to clear spiky artefacts after the tetrahedra labels are computed via the graph cut. The majority of the vertices that are not adjacent to any wrongly solid labelled tetrahedra have a solid angle of about 2π that spans about half the volume. Spiky artefacts in contrast are very sharp and most of them, including the most visibly noticeable ones, and span less than an octant of the surrounding volume. A threshold of $\pi/2$, which corresponds to the solid angle of a octant, is therefore a reasonable choice and yields good results.

5.2 Recursive Deletion

All vertices that have a solid angle $\pi/2$ are classified as spikes and all adjacent solid tetrahedra are relabelled empty. When a tetrahedron is relabelled empty however, the solid angle of all its vertices is impacted and needs to be reduced accordingly. When a tetrahedron is relabelled, its solid angle contribution to other vertices has to be updated.

There is a chance therefore, that by carving a tetrahedron to empty space, the solid angle reduction on another vertex results to the solid angle dropping below the $\pi/2$ threshold, making that vertex become a spike itself. In such occurrences, the tetrahedra around that vertex have to be re-labelled and, in respect to the subsequent re-labelling process, the subsequent vertices updated. This could theoretically propagate many times, which makes a recursive formulation convenient, such as the formulation give in algorithm 6. In the pseudocode given, `RemoveSpike` is only applied to vertices that have a solid angle $0 < \Omega_{\mathbf{V}} < \pi/2$.

The smaller 0 condition makes sure that no unnecessary calls to `RemoveSpike` for vertices ($\mathbf{V}$) that have no adjacent solid tetrahedra are executed. The smaller 0 condition also implies, that `RemoveSpike` is called at most once per vertex, thus avoiding any possibility of infinite recursive loops.

Algorithm 6: Spike Deletion and Recursive Removal of Subsequent Spike Formations

```
   RemoveSpike (V)
1  forall the Solid tetrahedra T (i.e. label L(T^4) = L_in) adjacent to V do
2  |  Carve T away from the volume by labelling as empty L(T) = L_out
3  |  forall the Vertices T_i of Tetrahedron T do
4  |  |  Ω_{T_i} ← Ω_{T_i} − Ω^T_{T_i}        // Carving T away from the volume alters the
   |  |  angles on T's vertices; subtract T's contribution to the vertex
   |  |  angles
   |  end
   end
5  forall the Solid tetrahedra T adjacent to V do
6  |  forall the Vertices T_i of Tetrahedron T with 0 < Ω_{T_i} < π/2 do
7  |  |  RemoveSpike (T_i)        // If removal of a spike constructs further
   |  |  spikes, remove these recursively
   |  end
   end
```

5.3 Joint Spikes

The algorithm from the previous section that makes use of the total solid angle of vertices does not perform well on spike formations similar to what is illustrated on figure 5.4. The solid tetrahedra adjacent to a vertex $\mathbf{V}$, that contribute the the total solid angle of $\mathbf{V}$ are sufficient to exceed the spikiness threshold of $\pi/2$, even though individual groups of mutually interconnected solid tetrahedra adjacent to $\mathbf{V}$ could have a solid angle of well below $\pi/2$ (for example artefacts in the figure). Such artefacts make themselves usually visible in the output mesh in hollow spaces of the reconstructed object.

An approach to overcome this problem could be to impose the $\pi/2$ threshold on groups of interconnected adjacent tetrahedra with solid labels, but this would leave the possibility open to allow the tips of two or more spikes (with solid angles greater than an octant) to share the same vertex. Intuitively, such a scene seems unrealistic to exist in a real scene, because the tips of two close real spikes would be more likely reconstructed by separate vertices.

A group of mutually interconnected solid tetrahedra adjacent to a vertex $\mathbf{V}$ refers to a set of L_{in} labelled tetrahedra, where $\mathbf{V}$ is one of the vertices in each tetrahedron in the set, and where every tetrahedron in the set can reach all other tetrahedra in the set merely through the neighbouring connections within the set.

A more sensible and intuitive approach to reject joint spikes, is to allow at most one solid volume, formed only by adjacent solid tetrahedra to $\mathbf{V}$, to be spanned at any single vertex. An easily implemented measure suitable for the purpose is to relabel all tetrahedra in tetrahedral groups at vertex $\mathbf{V}$, which contribute to less than half the amount of the total count of solid

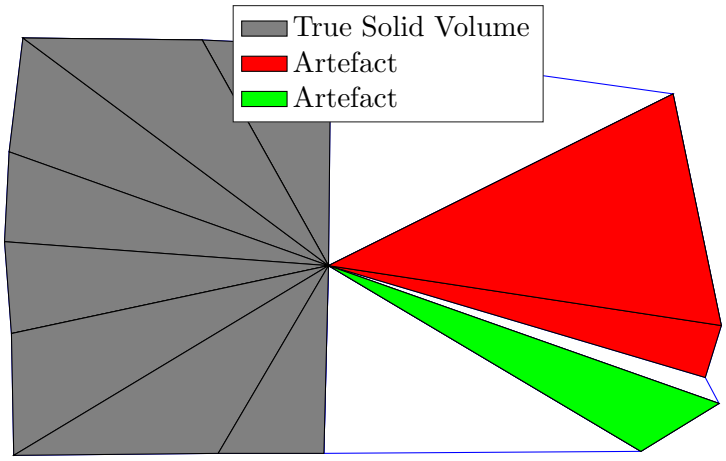


Figure 5.4: JointSpikes

tetrahedra. From the solid groups spanned on $\mathbf{V}$, either one or none are left.

Algorithm 7 shows the corresponding pseudocode that removes joint spikes, based on the criterion that removes groups with contribution of less than half the total solid tetrahedra count on a vertex. The counting and relabelling of the size of an adjacent solid group can be conveniently implemented recursively (this detail is not shown in the pseudocode). `RemoveJointSpikes`($\mathbf{V}$) removes spike joints on a specified vertex $\mathbf{V}$, and, in code, is called on all vertices that passed the spike clearing and have a total solid angle larger than $\pi/2$.

Similarly to removing standalone spikes, by relabelling groups of solid spikes to empty, subsequent spikes could be formed. Hence, when tetrahedra are relabelled empty during a joint spike clearing, their vertices are checked and `RemoveSpike` from algorithm 6 is used if the solid angle of any vertex drops below $\pi/2$.

Algorithm 7: Removal of Joint Spike Formations

```
   RemoveJointSpikes (V)
1  forall the Groups of solid tetrahedra adjcent to V do
2  |  if Size of the group less than half the total amount of solid tetrahedra on V then
3  |  |  RemoveGroup (G = set of tetrahedra in group)
   |  end
   end

4  RemoveGroup (G)
5  forall the Tetrahedra T in Group G do
6  |  Carve T and update solid angles of T's vertices
7  |  forall the Vertices V of T, whose solid angle dropped below π/2 do
8  |  |  RemoveSpike (V)   // remove subsequent spike formed from carving T
   |  end
   end
```

6 Complete Reconstruction Pipeline

When the algorithms have processed the new data at frame iteration, the surface that lies between differently labelled tetrahedra is constructed. The current implementation implements only a simple exhaustive search in the set of all empty labelled tetrahedra, looking faces that are adjacent to solid labelled neighbours. When such face is found, the triangle is saved as part of the surface, with the triangle normal pointing to the direction of the empty tetrahedron. During a surface extraction, the statistical mean and variance of the area and photoconsistency costs of the triangles are computed, which will be used for normalizing the graph cut costs in the next frame's iteration.

The triangles of the surface are textured with a simple procedure, based on the angle between the triangle normal and the vector from the triangle centroid to the cameras. Since these are directed in opposite ways, an angle of π would be the most geometrically normal view possible. Thus, the frame that maximizes the angle is chosen for the texture.

The following pseudocode put all algorithms individually presented in all above sections in wide context. Some of the algorithms are executed online and wrapped within other algorithms for efficiency reasons. A good example is the calculation of the photoconsistency costs. Photoconsistency for a face is initialized and loaded with the latest visibilities of its vertices during the update of tetrahedral links with the set of new tetrahedra during an insertion; enclosing faces are reused however so only truly new faces are initiated, while online updates for existing faces are executed after `ProcessRay` functions terminate. Computed costs and probabilities are reused in code whenever possible, and the same values (for edge costs, probabilities, angles) are never computed more than once. Vertex coordinates are stored only once in memory with pointers referring to coordinate blocks. Sets of such pointers are used to define tetrahedra, triangles and edges. Costs are also stored only once in memory, and face costs are attached to triangle faces, which are pointed to from two tetrahedra.

In the pseudocode specifying the algorithm and surface extraction is executed at every frame. Although costs and angles are reused as much as possible, the graph cut and surface extraction steps cannot be reused. In the spike removal algorithm, only the solid angles can be computed online and the search through all vertices for spikes (thresholding solid angles) has to be processed completely at every frame. If needed, these steps could be skipped at frames where a large number of vertices are fed into the algorithm to not have to discard or queue up pointcloud data. However, since costs from a reconstruction are impacting computation of new costs it would be best to execute volume and surface extraction as often as possible.

Algorithm 8: Complete Pipeline

```
1  Initialize Delaunay triangulation
2  Find Tetrahedron that contains the first camera
3  forall the Frames do
4      Load camera parameters, including camera centre C and pose
5      Load image from file
6      Convert to L*a*b* space
7      Blur image
8      forall the Vertices V that are visible from camera C do
9          if V already in triangulation then
10             Update V's visibility
11             Update estimation of V's 3D Gaussian distribution based on this new
               visibility
12             Update probabilities of tetrahedra impacted by V's distribution
13             Trace ray : ProcessRay(V, C, false)
14         else
15             C_aux ← get auxiliary camera reconstructing V: C_aux ≠ C
16             D ← find tetrahedra in collision set with V
17             F ← find enclosing faces of D
18             R_O ← find rays that originate in collision volume D
19             N ←Create new tetrahedra that replace tetrahedra in volume D
20             Update neighbourhood links of tetrahedra by using UpdateGridLinks(F)
21             Retrace all rays originating in volume D : ∀(V, C) ∈ R_O run
               ProcessRay(V, C, true)
22             Retrace rays inbound intersecting the re-triangulated volume through faces
               F: RetraceFromFaces (N)
23             Trace ray : ProcessRay(V, C, false)
24             Trace ray : ProcessRay(V, C_aux, false)
           end
       end
25     Define graph capacities for preparation for the graph cut
26     Find optimum labels by obtaining the minimum cut via a max flow computation
27     Compute total solid angles of all vertices; reuse previous computations is available
28     Use RemoveSpike (V) on all sharp vertices to remove all spikes
29     Use RemoveJointSpikes (V) on all vertices to remove all joint spikes
30     Reconstruct the surface model and remember mean and variance of face costs
31     Texture the surface based on most normal views of triangles
32     Optional: Print surface (to file)
   end
```

7 Experiments

This section contains some specific examples that show reconstruction performance. The experiments carried out should illustrate the reconstruction quality of the algorithms and the fitness for real time application. Contributions of the different algorithms in the reconstruction pipeline is qualitatively determined, by visual inspection of the produced surface models. The produced models are also directly compared with models produced by ProFORMA.

Since maintaining high reconstruction speed in the reconstruction pipeline, timing experiments have been carried out that show speed of individual algorithms. The speed of Delaunay triangulation is compared with CGAL [7] and VTK [10], two publicly available geometry libraries that offer Delaunay triangulation capabilities.

The majority of experiments uses the same datasets in order to be able to relate better between results from different sections in this chapter. The last section however shows reconstruction results for three more datasets.

7.1 Environment

The algorithms in their latest version have been compiled for single core use. The timing values were collected on one of the cores of an Intel i7-4702MQ CPU, at a clock speed of up to 3.2 GHz with Turbo Boost, and L1, L2, L3 cache sizes of 256KB, 1MB and 6MB respectively. There are 8GB of 1600MHz RAM available. The application is compiled in x64 and run on a Windows 10 Pro N OS.

7.2 Simulation Setup

The purpose of the algorithms described in this thesis is the implementation on a real time system on a mobile device. The algorithms are developped on a desktop OS however, and since the pointcloud for testing has been computed offline by VisualSFM [11] [12], the algorithms have to operate on the SfM data in a sensible order that simulates the pointcloud information that is computed during the online SfM. In an online SfM, the pointcloud and visibility knowledge are only known at most up to the latest frame and vertices are reconstructed with at least two cameras. This has been simulated by converting the n-view file outputted by VisualSFM to a set of files, each for the data intake at every frame iteration. Every such frame-file provides information of the camera pose and filename of the image used for reconstruction, along with new visibilities of old vertices from the new camera, and new vertices that are reconstructed in that iteration. Specifically, the information is provided in the following order in each file:

Camera centre; image filename; camera pose matrix `P` as a row vector (first 4 elements are `P`'s row 1, then row 2 and 3); focal length; rotation matrix; list of vertices.

The list of vertices can contain reference to both new and old vertices. All information is given in either a 10 element row vector fore new vertices, or a 4 element row vector for old

vertices that are visible in the new frame. The first element in both these vectors is a key to denote if the vertex is new or old, being 0 and 1 respectively. Vectors of new vertices are in the following format:

Key 0; vertex id; XYZ coordinates; xy coordinates in current camera's image; id of auxiliary camera; xy coordinates in auxiliary camera's image.

Old vertices have the format:

Key 0; vertex id; xy coordinates in current camera's image.

Simulating vertex deletions is not supported by the current version.

7.3 Experimental Datasets

The experimental reconstruction results shown in the following chapters will be all based on either one of two datasets shown on figures 7.1, 7.2 and 7.3. Both datasets are images from video sequences scanning around a model of an ancient temple. The model that is video recorded in the first dataset, with the pointcloud shown in figure 7.1, is of a miniature model of a temple in Agrigento in Sicily and will be referred to as the Agrigento dataset.

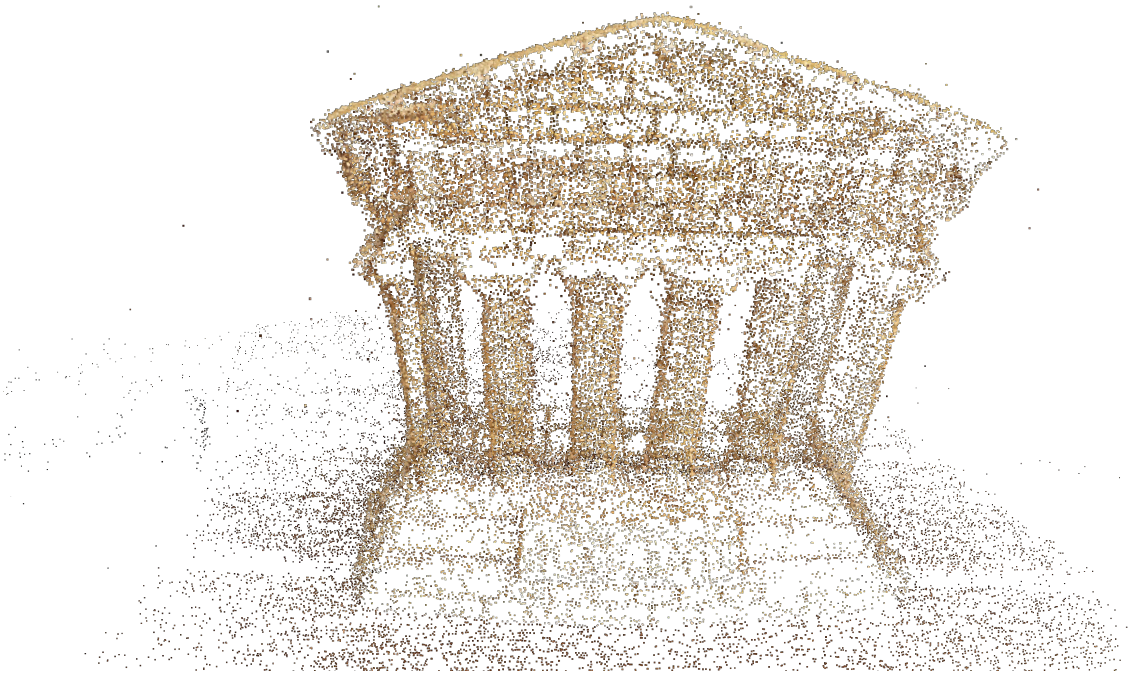


Figure 7.1: Pointcloud of Agrigento Dataset

The Temple Ring dataset is available on the internet, published alongside [13] and has been recorded in a controlled environment, where the background of the object is black and the scene is relatively uniformly illuminated. The second dataset is of a similar object and has been personally recorded and reconstructed. The pointcloud for the two datasets is obtained with VisualSFM, a SfM application written by Changchang Wu.

There are multiple reasons why these two particular sets have been chosen. having a pointcloud that one would expect to be able to deliver qualitative results is an important factor when it comes to testing the accuracy of the algorithms. This does not mean that the

Figure 7.2: Reconstructed Surface for Agrigento Dataset

(a) SfM Pointcloud **(b)** Reconstructed Surface

Figure 7.3: Temple Ring Dataset

vertices in the cloud have to be all accurate, it rather means that there should be sufficient vertices such that a human that has seen the real scene can recognise the scene based on the rendered pointcloud. There may be noise in the vertex coordinates and there may be outliers, but the main object reconstructed should be well viewed in the video sequence. These dataset provide a well rounded view of the object in the centre of the scene. In these two datasets, the objects scanned are very similar, which allows for direct comparison of the reconstructions of both scenes. The Agrigento set is relatively large with 74733 vertices, while the Temple Ring set has 11572 vertices. From these however, only the vertices which have at least 3 views are used for the siulation, as vertices reconstructed from only 2 views tend to involve a great deal of noise and harm the reconstruction quality. Therefore, the portion of the pointcloud used from the datasets is 52188 vertices for the Agrigento set and 6157 vertices for the Temple Ring set. Vertices are reconstructed 3D points with SfM and cameras do not count as vertices. The Temple Ring dataset consists of 47 frames at a 640×480 resolution and the Agrigento dataset consists of 183 frames at a 1920×1080 resolution.

These two datasets have been the most strongly utilised sets during the development and testing of the algorithms, throughout the thesis.

7.4 Task Timing

All statistics and timing results displayed in this section are based on the reconstruction of the Agrigento dataset. In the subsection 7.4.1, figure 7.8 also displays the algorithm timing for the smaller, Temple Ring dataset.

The algorithms process vertices and vertex visibilities to calculate the tetrahedral grid and reconstruct a surface between solid and empty tetrahedra. At each iteration, as the online SfM reconstructs more vertices and the pointcloud grows, the grid and the surface grow too. Figure 7.17 shows the amount of vertices, tetrahedra, triangles and rays in the reconstruction process against the number of frames processed. Although this is subject to a certain degree of randomness, the amounts increase relatively linearly. It can be aslo seen that the size of the Delaunay triangulation has a very linear relationship to the amount of vertices in the triangulation, by having about 6.3 tetrahedra per vertex.

By observing the ratio to the number of vertices specifically, in figure 7.5, it can be seen that after a certain amount of frames, the size of the reconstruction increases linearly to the amount of vertices.

The incremental reconstruction pipeline proposed in this thesis contains many algorithms, and all have different complexities. All algorithms try to reuse as much of the computed data as possible and perform online updates, except the graph cut and the surface computation, which run from scratch at every iteration. The surface extraction at the current implementation is implementing an exhaustive search and thus is very inefficient, with a squared complexity. Figure 7.6 shows the total time spent on each of the individual algorithms in the reconstruction pipeline, sampled on every frame.

Although the important factor of a real time system is the framerate it can operate, the algorithms are processing geometric objects such as vertices, rays and tetrahedra. Their complexity is more tightly related to these, rather than the actual framerate. Since there is a more or less linear relationship between number of vertices and number of rays, tetrahedra and triangles in the reconstruction it frames other than at the very beginning (figure 7.5), many of the algorithms can be expected to have a complexity that is determined w.r.t. the

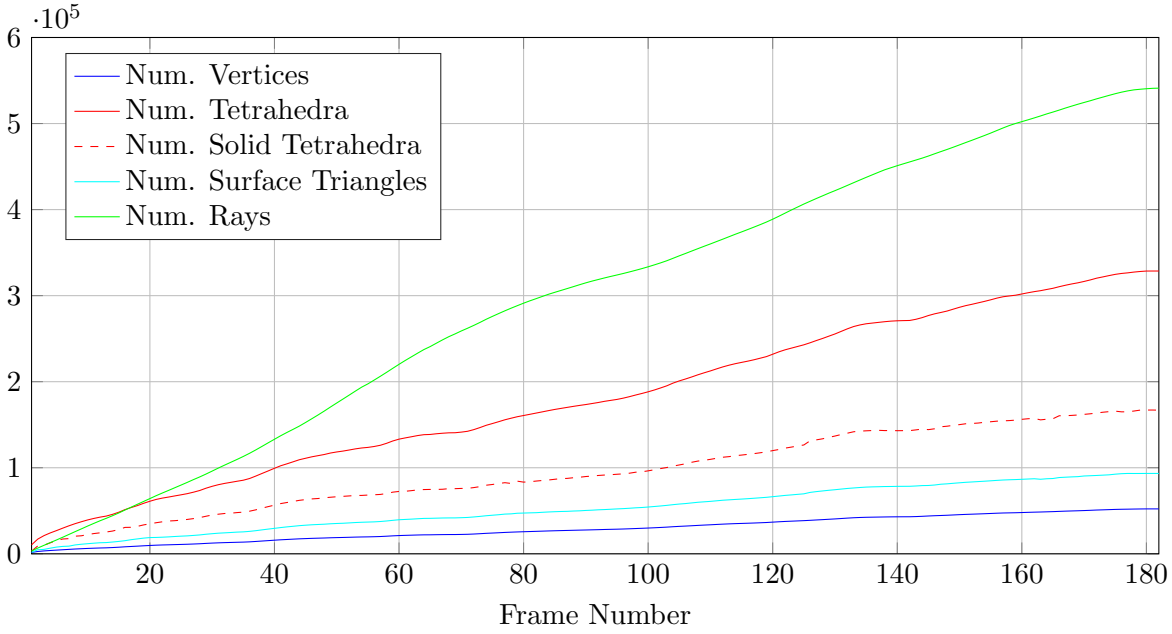


Figure 7.4: Reconstruction size of Agrigento Set

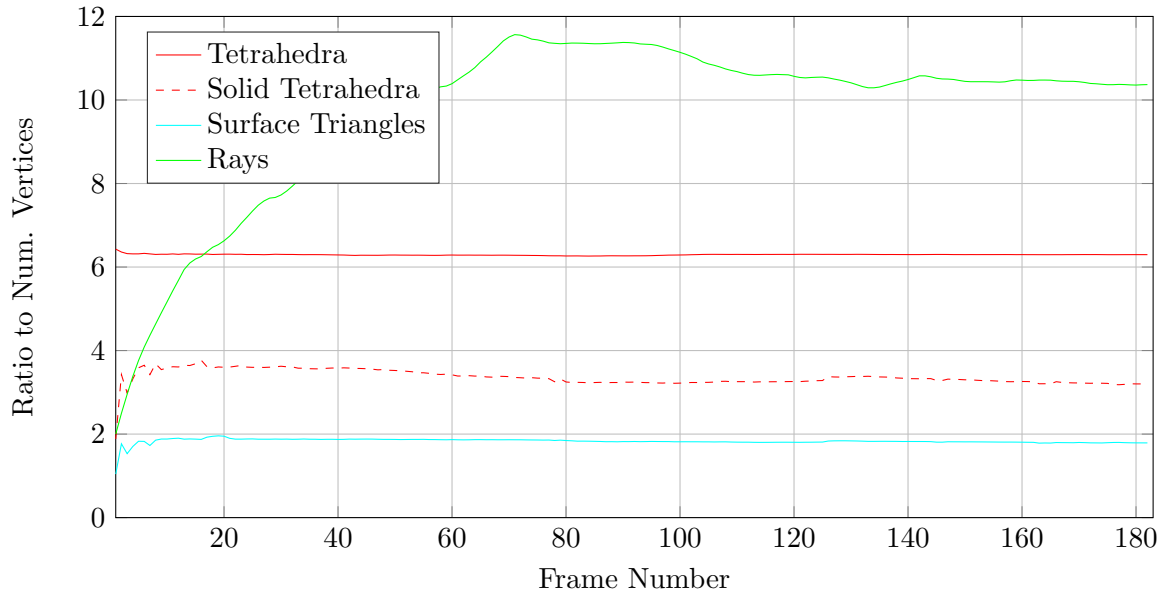


Figure 7.5: Per Vertex Ratios

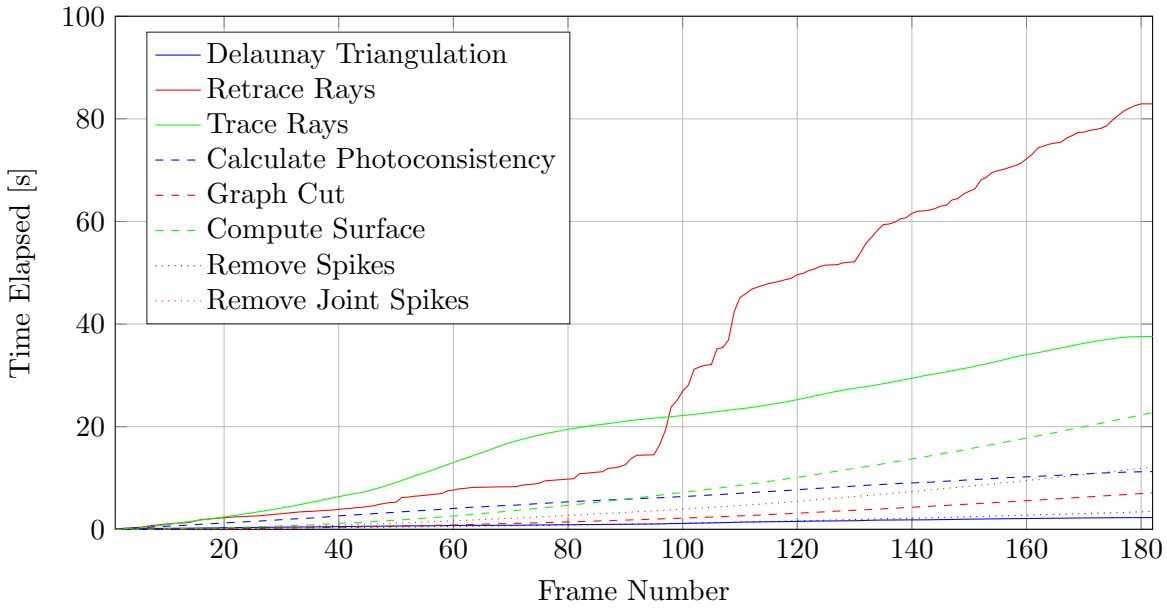


Figure 7.6: Algorithm Timing

number of vertices rather than the number of frames. Figure 7.7 is displaying the total time taken by individual algorithms w.r.t. the number of vertices.

Multiple core implementation could be expected to improve the results by a relatively large amount, given that ray traces and other operations can be implemented independently. At some point during development, a multicore implementation was in use, which was later abandoned however.

7.4.1 Task Timing of Temple Ring Dataset

The algorithm timings for the Temple Ring dataset are also shown on the figures 7.8. The overall framerate of this dataset is considerably larger than the Agrigento dataset, which is linked to the amount of vertices in the pointcloud. Due to the large scale difference between 7.8 and 7.7 it cannot be seen, but when zooming into the region of up to 6000 vertices, it can be seen that the timing curves for the various algorithms is very similar, hinting that the reconstruction speed is directly dependent on the number of vertices, or tetrahedra, or rays, since these all have mutually linear ratios (figure 7.5).

7.5 Comparison with ProFORMA

Since the most relevant related work to this these is the ProFORMA project [3], reconstruction results are directly compared between the reconstruction algorithms proosed by this thesis, and the algorithms proposed by the authors of ProFORMA. The actual software of ProFORMA is not publicly available, however, during the development timeline in this thesis, an early version of the reconstruction pipeline employed the algorithms proposed in the ProFORMA paper. The only difference between reconstruction techniques in that early version and ProFORMA is that texturing was not supported.

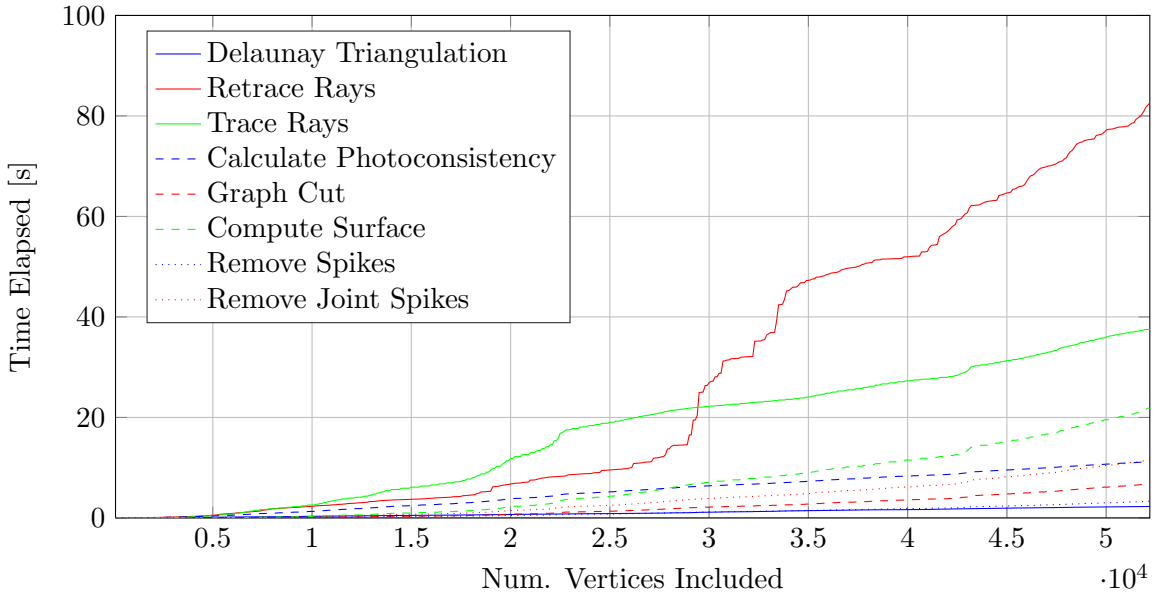


Figure 7.7: Algorithm Timing w.r.t Vertices Processed

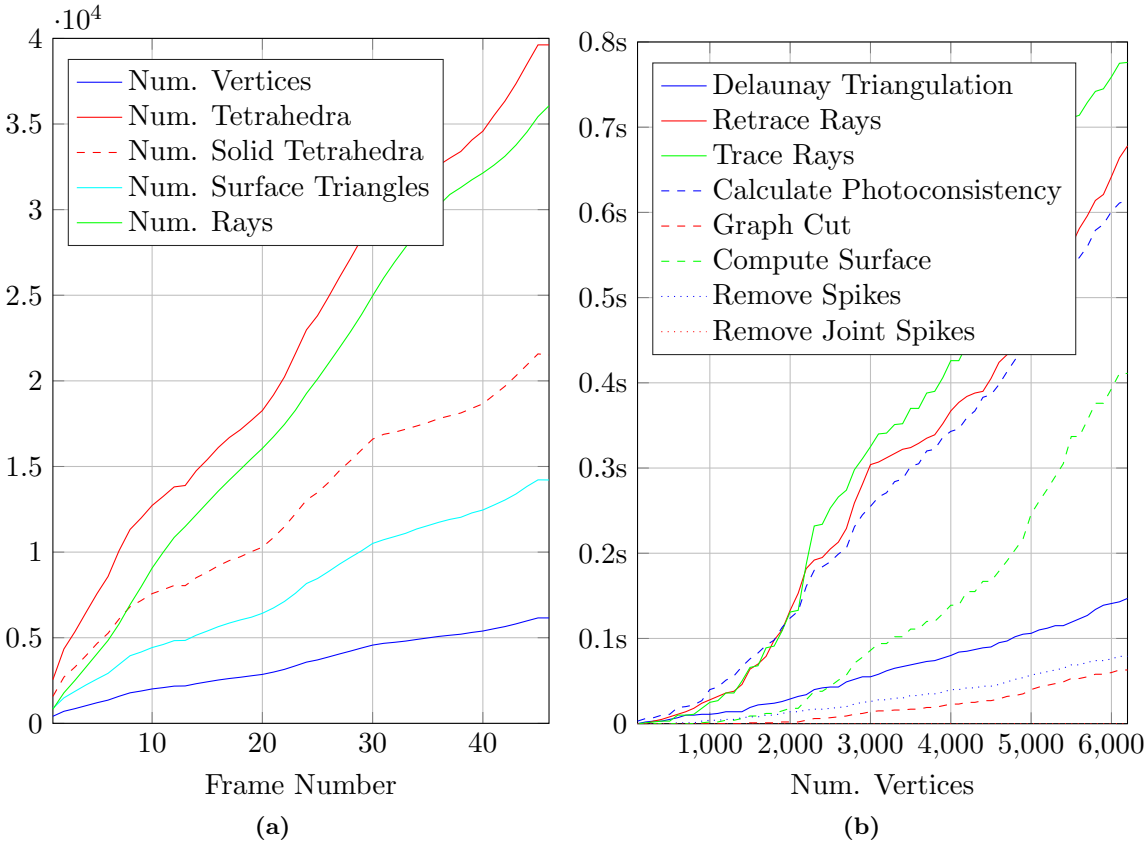
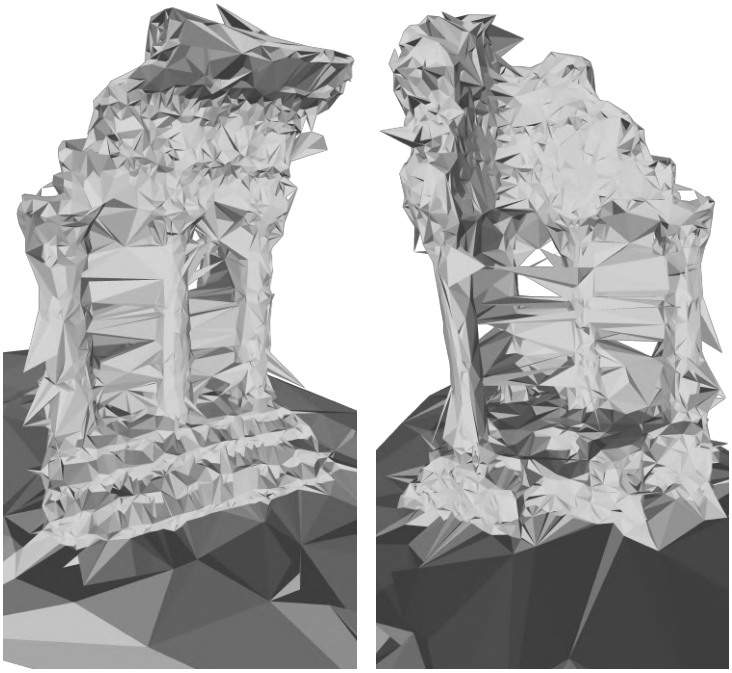


Figure 7.8: Timing Statistics for Temple Ring Dataset



(a) Front View **(b)** Back View

Figure 7.9: Temple Ring Reconstruction with ProFORMA

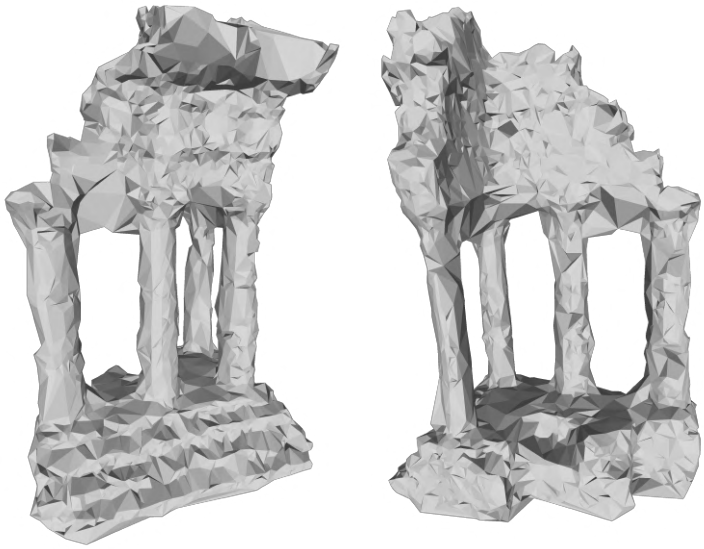
The figures 7.9 illustrate the reconstructed surface using the ProFORMA approach on the Temple Ring dataset. By comparing figures 7.9 with the textureless reconstructions in figures 7.10 using the proposed pipeline in this thesis shows a large difference in quality. The proposed techniques result to a clearer and more accurate reconstruction, with less artefacts.

7.6 Quality Evolution

This section illustrates how each of the steps in the reconstruction pipeline affect the quality of the surface. The enhancements are always shown with respect to the previous enhancement step. Enhancements are shown on both, the Agrigento and the Temple Ring datasets, in order to illustrate the applicability to differently sized triangulations.

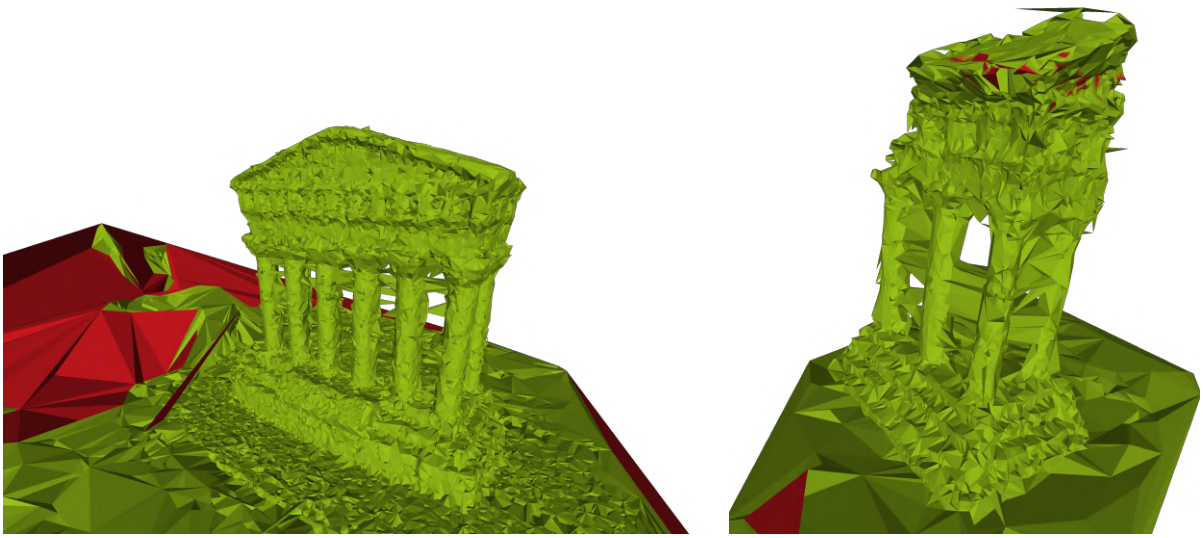
The first comparison is between the simple carving process, where each tetrahedron is intersected by a ray gets carved out, and the tetrahedra labelling based on minimizing the probabilities with a graph cut that takes into account the area costs of faces between tetrahedra. Figure 7.11 shows which tetrahedra that were reconstructed with the simple carving procedure are removed from the object's volume when the graph cut labels are used. Tetrahedra that cease to exist are having red colour. It can be seen here that on the object at the centre of the scene the difference is not too large. On the surrounding area however, many large reconstructed surface artefacts have been removed with the graph cut implementation.

The second large improvement is the introduction of the photoconsistency costs of the faces between tetrahedra, which are optimized within the graph cut minimization framework.



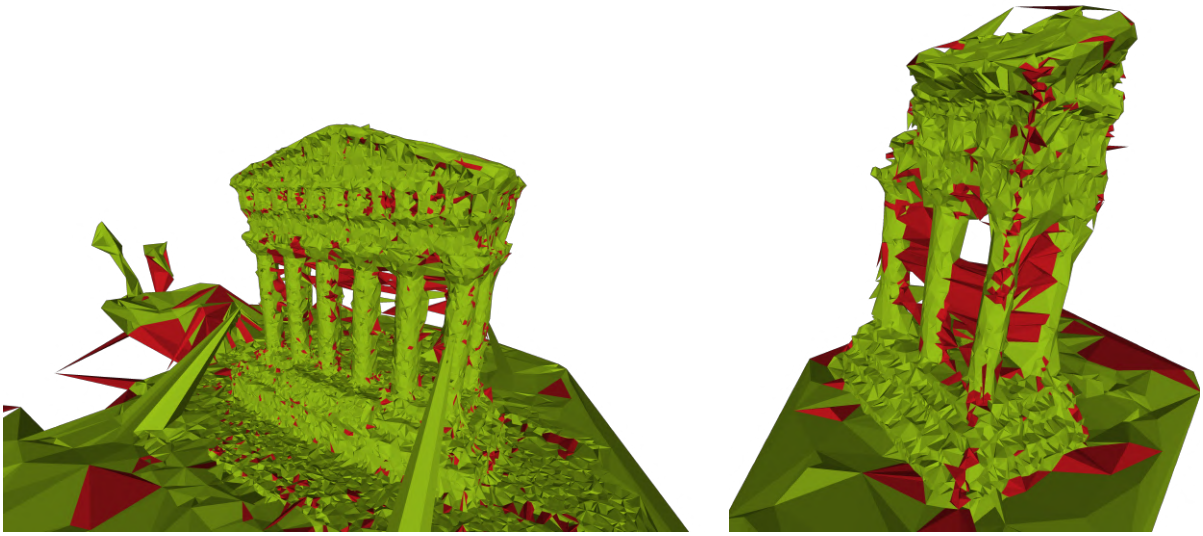
(a) Front View **(b)** Back View

Figure 7.10: Textureless Temple Ring Reconstruction



(a) Agrigento Set **(b)** Temple Ring Set

Figure 7.11: Graph Cut with Area Costs enhances Simple Carving



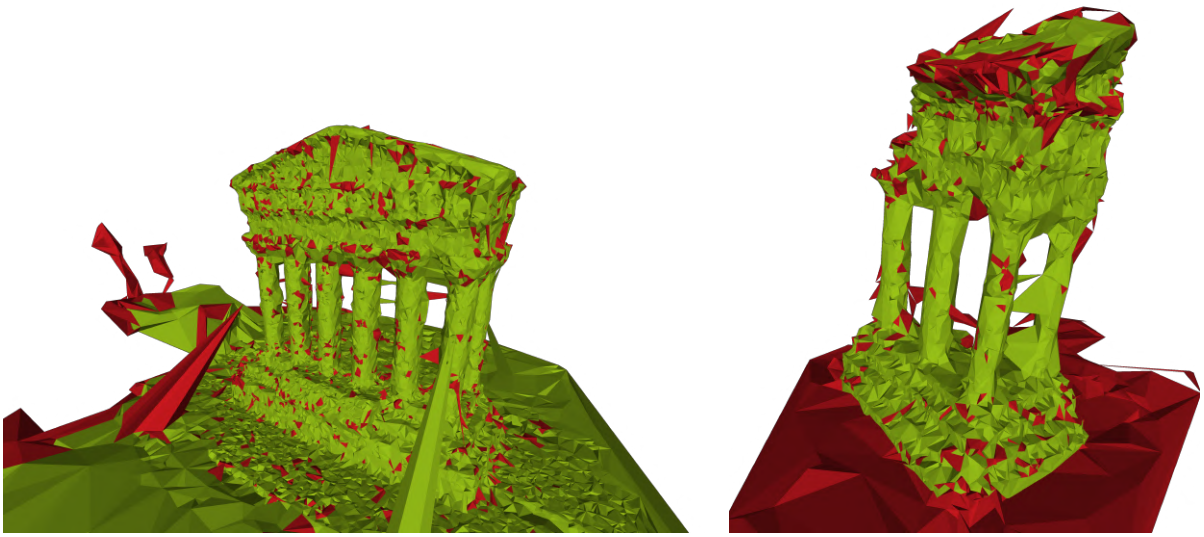
(a) Agrigento Set **(b)** Temple Ring Set

Figure 7.12: Photoconsistency Enhancements

Figure 7.12 shows the improvements with the same colour coding again. The large benefit of the photoconsistency measure is that badly reconstructed triangles in concave and hollow regions of the grid can be removed, given sufficient colour diversity in the views of such triangles. In these datasets in particular, the hollow regions between the columns of the temple present a great target for the manifestation of artefact triangles between adjacent columns. unless a very large number of vertices and vertex views is reconstructed by SfM, it is almost guaranteed that reconstructions of such objects like the temples leads to this kind of artefact formation. Especially in the Temple Ring dataset which has a fewer vertices and thus fewer rays, the triangles between the columns remain either un-intersected or poorly intersected. Introducing the photoconsistency costs in the graph cut minimization can very effectively remove these triangles, as can be seen in figure 7.12, as they are marked red. Applying photoconsistency costs however seems to also produce new artefacts as can be seen on the spike appearance on the reconstruction of the Agrigento set from figure 7.11 to 7.12. In general though, more artefacts are removed than created by introducing photoconsistency costs and the overall reconstruction quality increases.

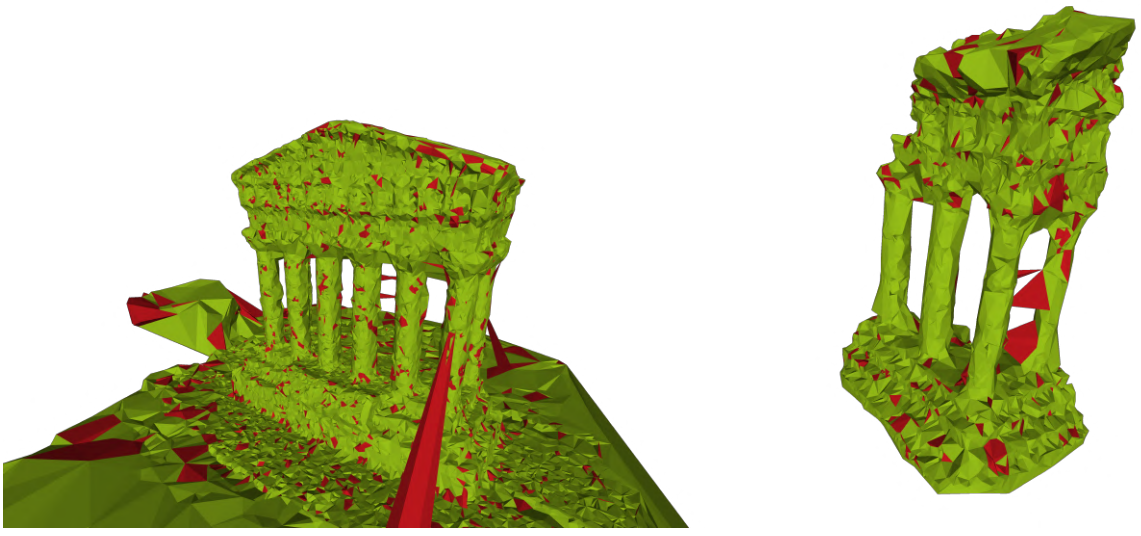
After the graph cut labels the tetrahedra, there are still artefacts left in the reconstruction. As it can also be seen on the previous two figures, the most visible of the remaining artefacts are spikes. Spike removal is executed after the graph cut, which removes the majority of spiky artefacts, as can be seen on figure 7.13.

After de-spiking, some spikes seem to still remain. These have such a geometry, that a spike is attached to a vertex reconstructing the surface, and thus the solid angle on that vertex (which is the spikiness measure) does not trigger a spike removal. These occurrences are referred to as joint spikes and their removal is separately handled. Figure 7.14 shows how the joint-spike removal algorithm clears the reconstruction from these type of artefacts.



(a) Agrigento Set **(b)** Temple Ring Set

Figure 7.13: Spike Removal



(a) Agrigento Set **(b)** Temple Ring Set

Figure 7.14: Joint Spike Removal

7.7 Speed of Delaunay Triangulation

In the final version of the software, an own implementation of the 3D Delaunay triangulation has been used. Initially, The Visualization Toolkit (VTK) by Kitware [10] was used to produce the triangulation, but at some point the algorithm with the tetrahedron locator that uses the SfM visibility has been implemented.

The main motivation here was speed, as the VTK triangulation was very slow and the methods that are used to obtain the neighbouring tetrahedra and faces of a tetrahedron are comparably slow. A very popular choice of a Delaunay triangulation library is CGAL [7], which is a very efficient implementation of many geometric algorithms. The issue with CGAL is, that it does not compile for mobile OSs, which is why VTK was initially considered.

Figure 7.15 illustrates how much slower VTK's Delaunay triangulation is compared to CGAL and the own implementation in this thesis. The data i the figure is based on the reconstruciton of the Agrigento dataset. The current implementation seems to even be slightly quicker than CGAL.

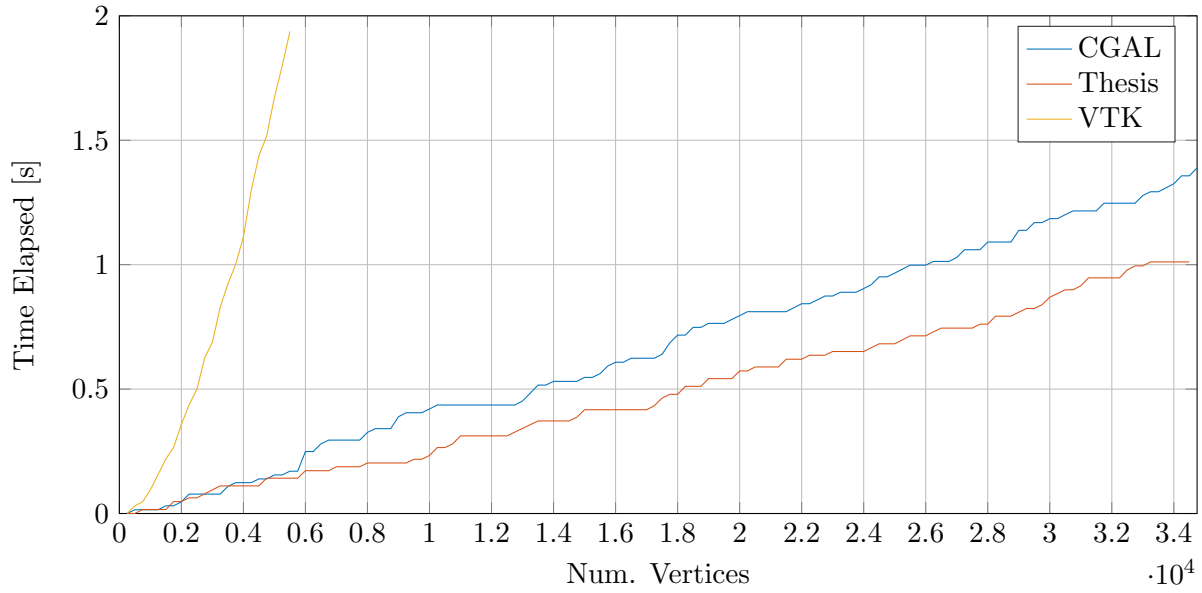
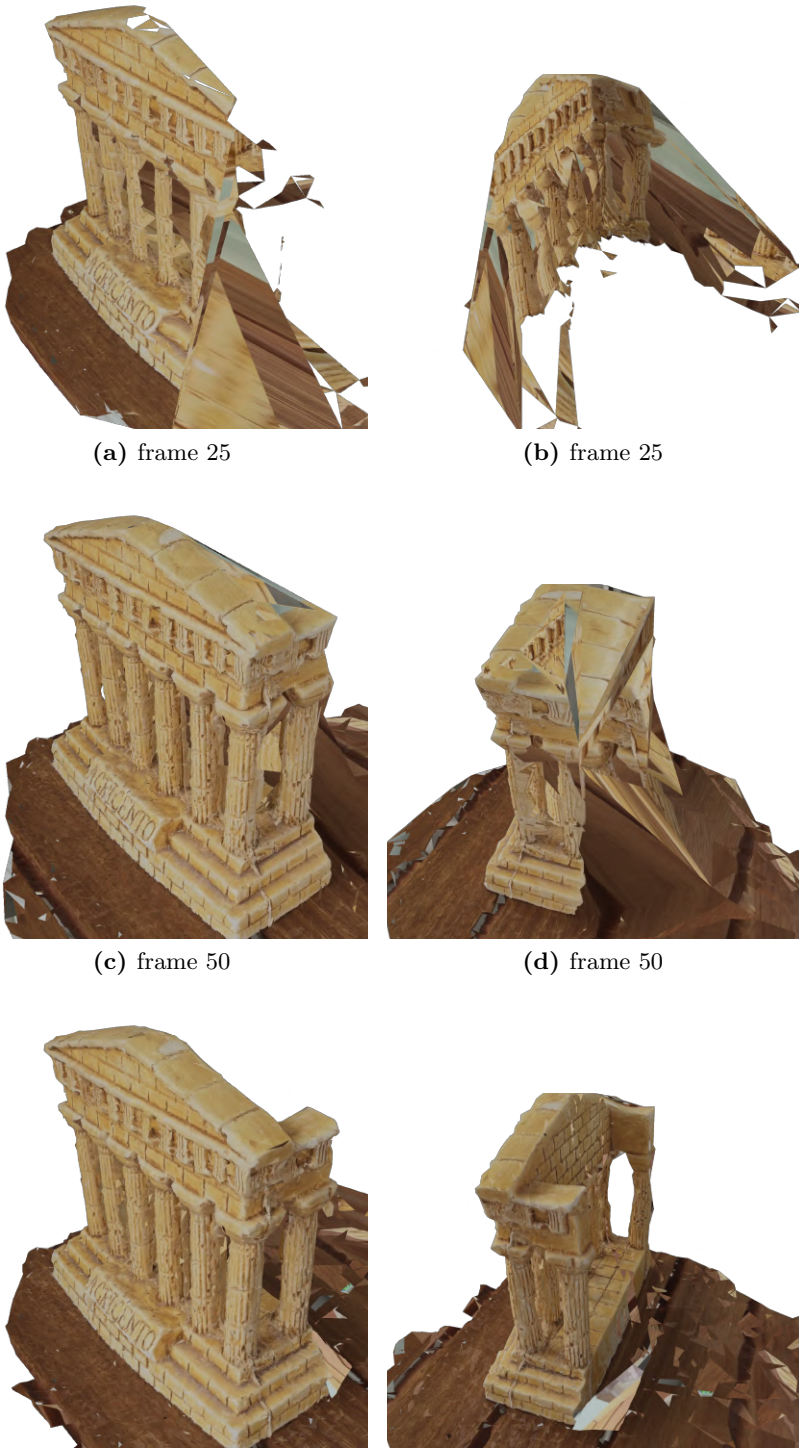


Figure 7.15: Speed Comparison for the Delaunay Triangulation

7.8 Incremental Surface Reconstruction

The reconstruction pipeline is incremental, and after every frame an intermediate surface can be reconstructed and rendered (the current implementation does not render, but rather has the option of saving to a file). The user of the hypothetical end application, that runs on a mobile device and uses SLAM to compute the pointcloud, will be able to see the intermediate reconstructions in real time.

Figures 7.16 display the intermediate surface reconstructed from the Agrigento dataset at frames 25, 50 and 75. At frames 25 and 50 the back side of the temple has not been explored yet. Between frames 25 and 50 one can see the gradual construction of the front side of the temple and between frames 50 and 75 the gradual reconstruction of the back side.



(a) frame 25 **(b)** frame 25

(c) frame 50 **(d)** frame 50

(e) frame 75 **(f)** frame 75

Figure 7.16: Online Surface Rendering

7.9 Varying Sequence Framerate

The Agrigento dataset has been extracted from an original video sequence which was recorded at a framerate of 60 and at a resolution of 1920 × 1080. The full image sequence of the Agrigento dataset contains every 10^{th} image of the video. If theoretically the online SfM would be able to reconstruct the same amount of vertices as VisualSFM can reconstruct for the images, then the real time reconstruction pipeline would not be fast enough to keep up at 6 Fps. Sticking for now to the pountcloud resolutions delivered by VisualSFM, reconstructing from an image sequence with lesser framerate results in general to a sparser pointcloud with less points, which reduces the total timing of the reconstruction pipeline. For example x amount of seconds from a 6 Fps sequence takes longer to reconstruct that the same amount of seconds from a 4 Fps sequence. In figure 7.17 the pipeline reconstruction duration for various video framerates is shown.

For smaller video framerates, naturally, the quality of the reconstructed surface reduces. Figure 7.18 shows the reconstruction from a 2 Fps video, which is not as qualitative as the reconstruction from the 4 Fps reconstruction shown on figure 7.2. Too large framerates however lower the quality of the reconstructed surface (for example the 6 Fps Agrigento sequence has more artefacts than the 4 Fps). This could be due to a larger number of outliers and badly reconstructed vertices. Figure 7.19 shows the surfaces reconstructed from datasets with framerates of 2, 4 and 6 Fps from a similar angle, that emphasizes the most obvious region proving the reconstruction from the 4 Fps Agrigento set produces better results than reconstruction from the 6 Fps dataset. Note that all these datasets with different framerates have frames with same resolutions of 1920 × 1080.

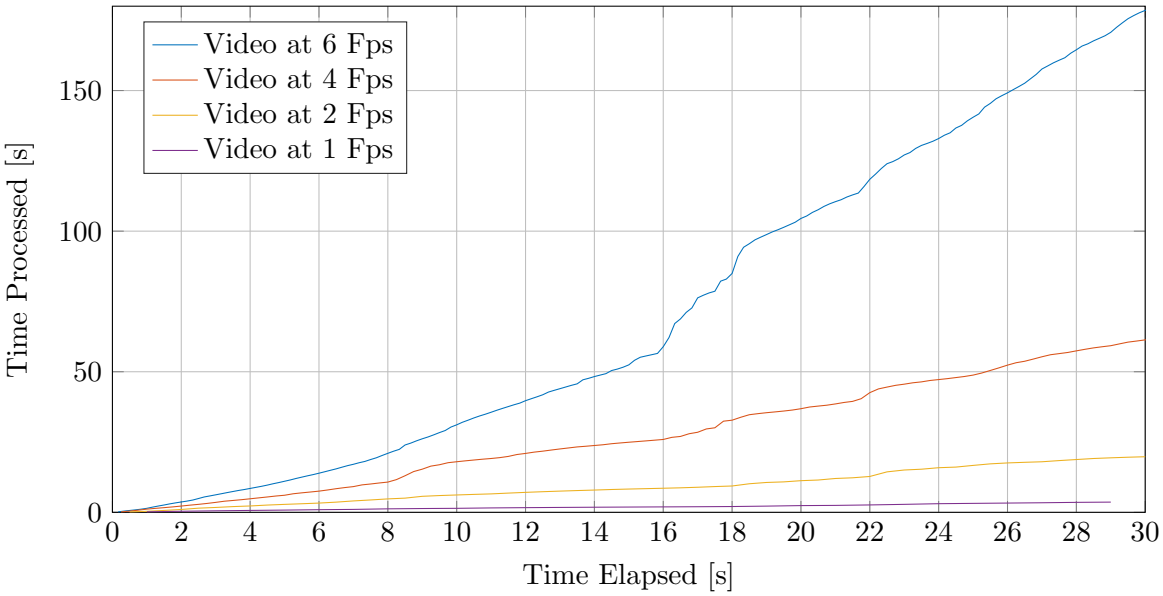


Figure 7.17: Reconstruction size of Agrigento Set

Figure 7.18: Agrigento Reconstruction at 2 Fps 1080p Video Sequence

(a) 2 Fps **(b)** 4 Fps **(c)** 6 Fps

Figure 7.19: Reconstruction at Different Video Framerates

7.10 Reduced Pointcloud

Online SfM does not in general reconstruct as many vertices as an offline implementation such as VisualSFM. Although it can be seen that in some caases smaller pointcloud sizes create better surfaces, the number of vertices is by quite a factor smaller in online SfM. This means that the quality of the reconstruction can be expected to be limited by the pointcloud size.

In this section, the pointcloud size of the Agrigento dataset is sampled to 1/3, 1/5 and 1/15 of the original size of the Agrigento dataset reconstructed with the 4 Fps video. Figure 7.21 shows in above order the textureless reconstructions using the reduced pointcloud sizes. As it can be expected, the reduced pointclouds results to a quality loss of the reconstructed surface.

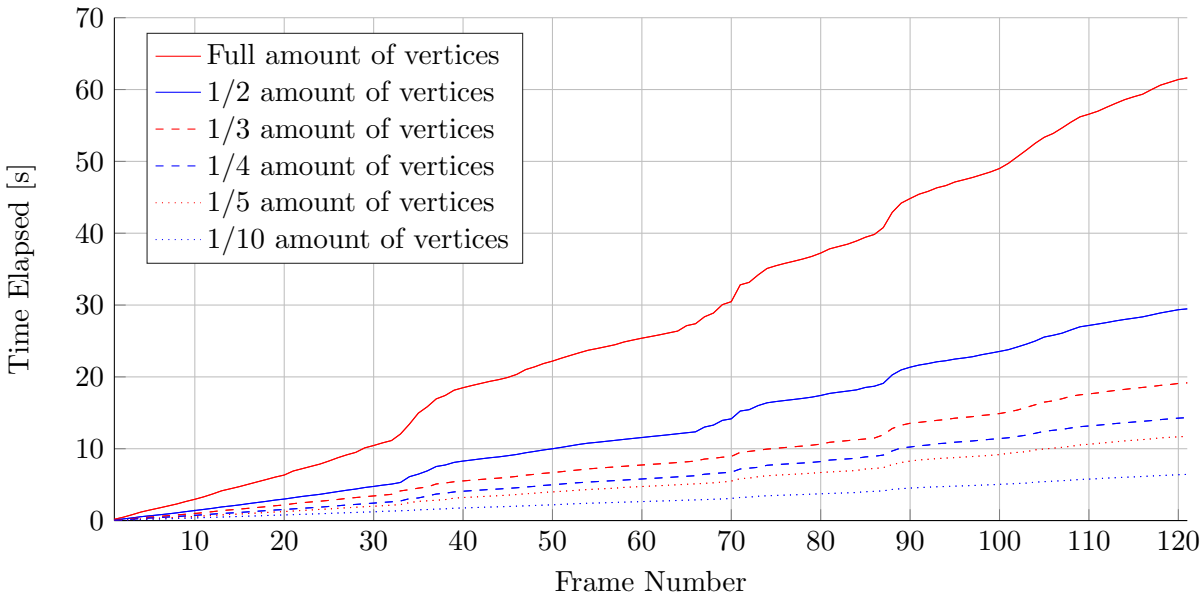


Figure 7.20: Reducing Pointcloud Size of Agrigento Dataset

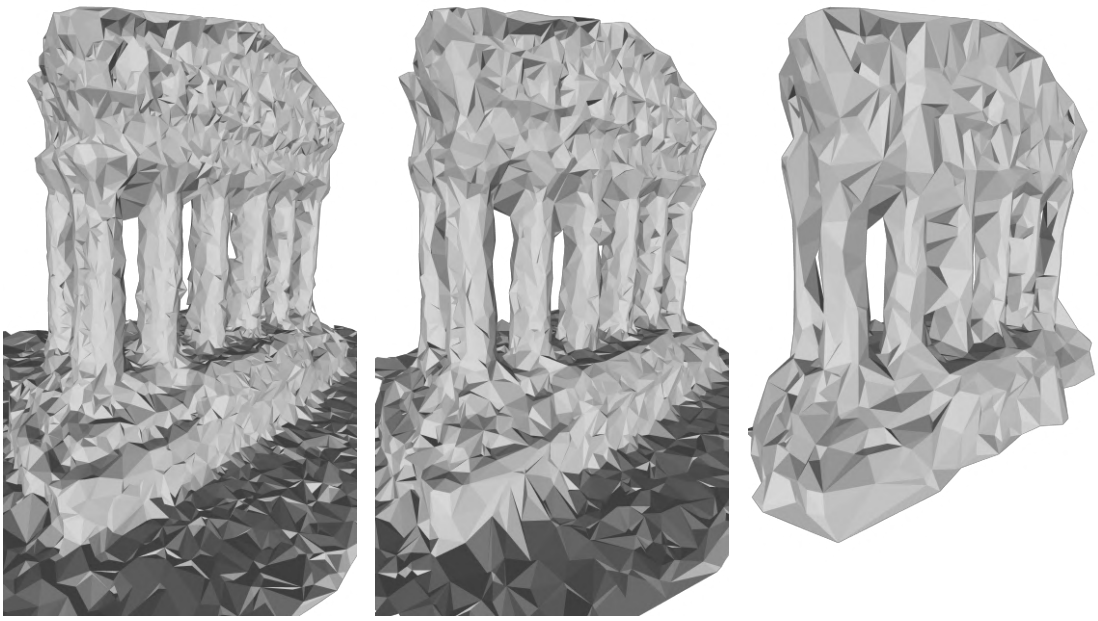
Simultaneously however, since the algorithms' complexity depends on the pointcloud sizes, the reconstruction speed increases. Figure 7.20 shows the total reconstruction time per frame for different pointcloud sizes.

7.11 Varying Resolution

An important factor that needs to be considered in both quality and speed of the reconstruction is the resolution of the video sequence. The higher the resolution, the more vertices can be reconstructed from the SfM, which generally is desirable but impacts reconstruction speed negatively. Processing lower resolution frames quicker enables on the other hand support for reconstructing at higher video framerate, or at least one would think so. From the experiments with various framerates in chapter 7.9 it could be noted however, that too high framerates produces more outliers in the pointcloud (4 Fps seemed to be doing best).

The Agrigento dataset is natively recorded with a 1920×1080 resolution. two further reconstructions have been computed with pointclouds generated from image sequences at

(a) 9937 Vertices + Texture **(b)** 5962 Vertices + Texture **(c)** 1988 Vertices + Texture



(d) 9937 Vertices **(e)** 5962 Vertices **(f)** 1988 Vertices

Figure 7.21: Reduced Pointcloud Reconstructions

1280×720 and 640×360. The pointcloud generated with VisualSFM with the datasets at a reduced resolution of 720p however, is in fact larger than the pointcloud generated using the full resolution images. The 720p dataset produces a total of 139074 vertices, from which 82827 vertices have at least 3 views; compared to the 1080p dataset, which produces in VisualSFM 74733 total vertices and 52188 vertices with minimum 3 views.

The larger amount of vertices leads naturally to a slower reconstruction framerate, as can be seen on figure 7.22, that shows timings for the different resolutions. The 360p resolution dataset on the other hand produces 42970 vertices in VisualSFM, of which 28857 have more that 2 views.

Figures 7.23 show the reconstructed surfaces for when a lower resolution Agrigento dataset has been used in VisualSFM. From a qualitative inspection of the reconstructions it seems that the 720p dataset not only reconstructs more vertices in VisualSFM, but also more qualitative vertices, as the number of obvious artefacts is reduced. The overall surface however is slightly rougher for the 720p set, most likely due to its pointcoud size and the variations that naturally are present in vertex reconstructions. The reconstruction from the 360p dataset is on the other hand worse, and one of the back columns of the temple model has been carved out. In congunction with the other experiments, this suggests that at least for use together with VisualSFM, it is better to drop the frame rate than to drop the resolution to as low as 360.

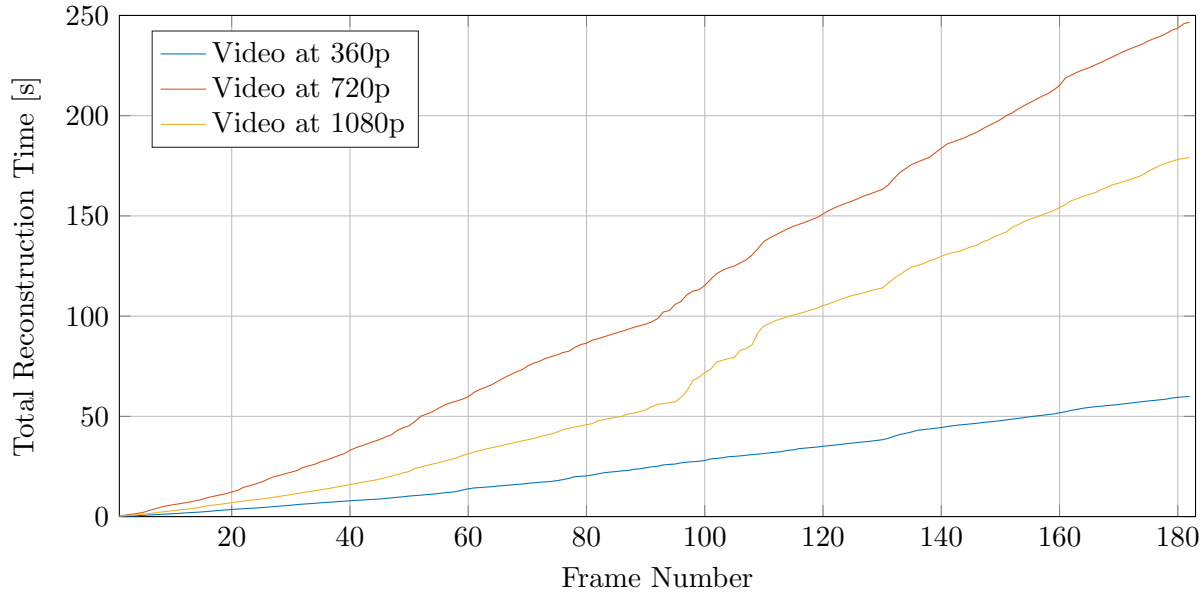


Figure 7.22: Reconstruction at Different Video Resolutions

7.12 More Reconstructions

So far, reconstructions shown were either from the Agrigento or the Temple Ring dataset. This section shows a collection of other reconstructions that illustrate the quality of the algorithms. Timing and other statistics are not shown here, as this section's purpose is to merely illustrate algorithm applicability at different scenes.

The Citywall dataset shown in figure 7.24 is reconstructed with only 200 of the 562 frames,

(a) Reconstruciton from 360p Agrigento Set

(b) Reconstruciton from 720p Agrigento Set

Figure 7.23: Reconstructions at Different Video Resolutions

as at the point of around 200 frames, the set of images for the surface's texture becomes too large and is not supported by the used renderer, Meshlab [14]. During reconstruction of the full 562 frame long Citywall dataset, more memory than available in RAM is needed, at which point hard drive is utilised and the algorithm becomes very slow. The "Der Hass" and "Citywall" datasets are available online from the same publisher of [15].

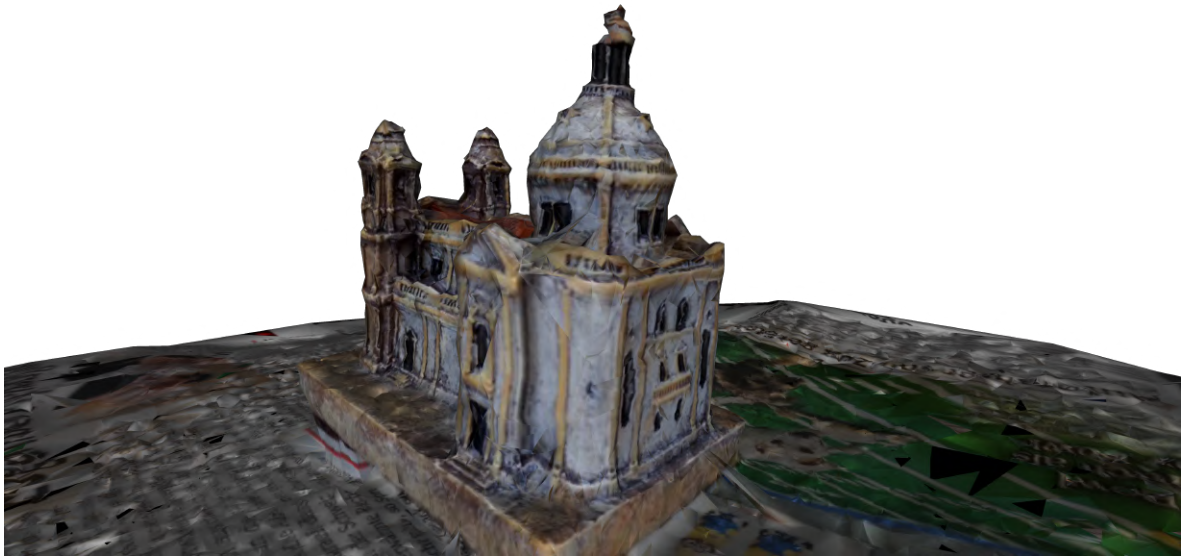


Figure 7.24: 37 Frame Miniature Church Model

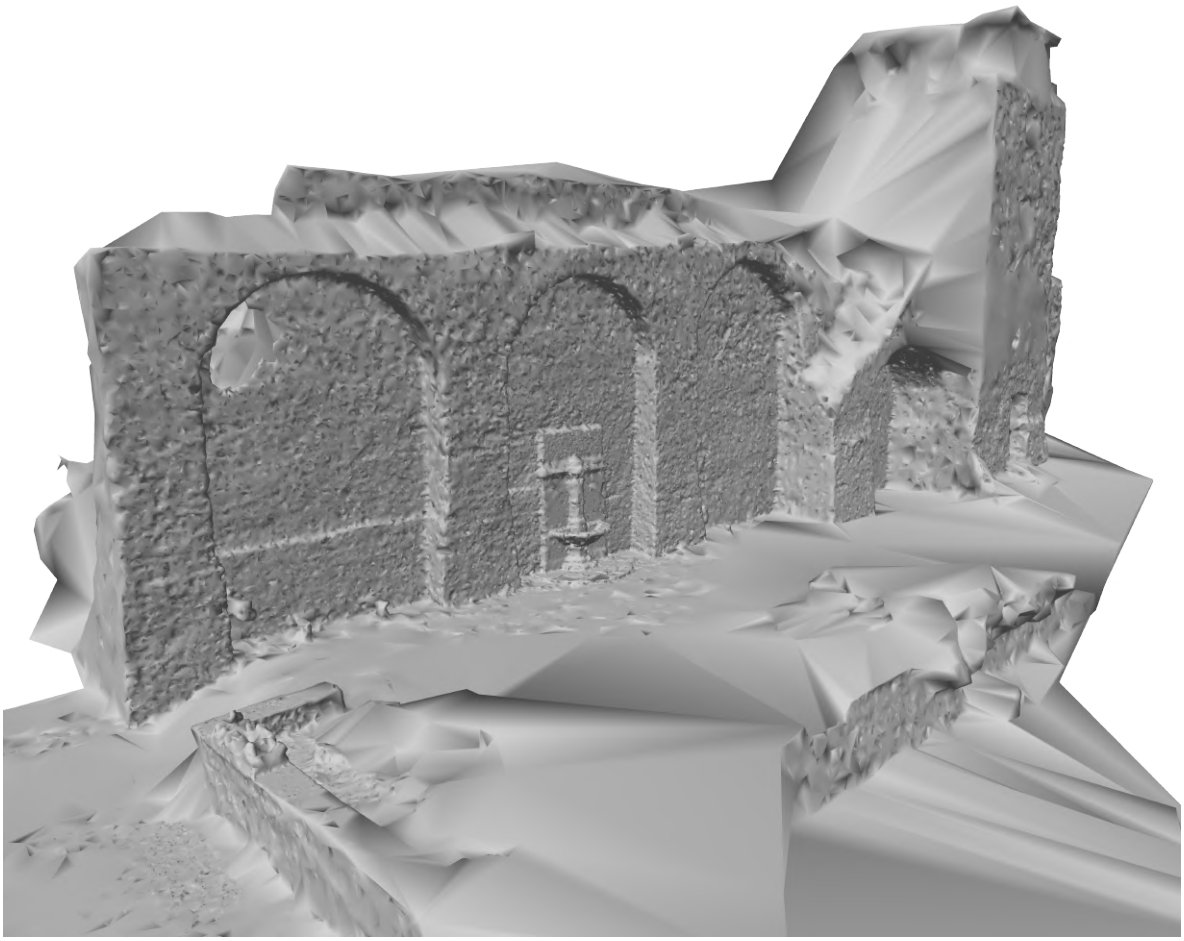
7.13 Further Work

In order to evaluate the efficiency and performance of the algorithm, the code was tested on a mobile phone. This part of the work was done right after the official end of this master thesis. In the examples on figures 7.28 to 7.31, ORB-SLAM [16] was used as the tracking and mapping module, and based on the map update events, the reconstruction was formed incrementally. Note that in these figures, some mesh triangles can be missing. The reconstruction itself is in fact watertight, however, triangles where no texture is found are being deleted in the texturing step, which accounts for the holes seen in figures 7.28 to 7.31.

Figure 7.25: 79 Frame "Der Hass" Dataset

Figure 7.26: 149 Frame Model of Two Hats

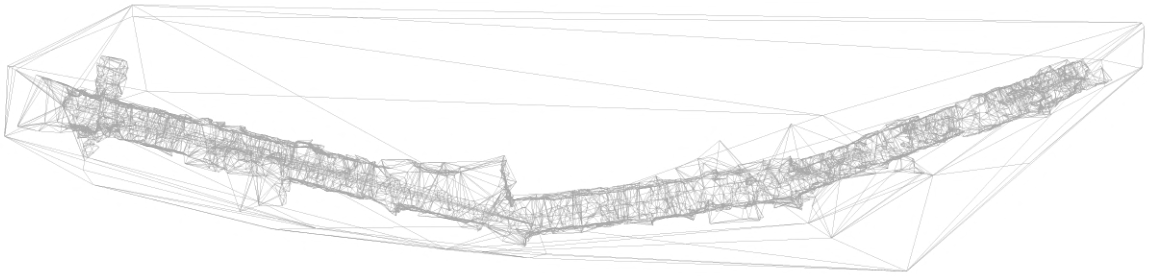
(a) Textured Mesh Reconstructed with Subset of 200 Frames



(b) Full Reconstruction without Texture

Figure 7.27: 562 Frame "Citywall" Dataset

(a) SLAM Map



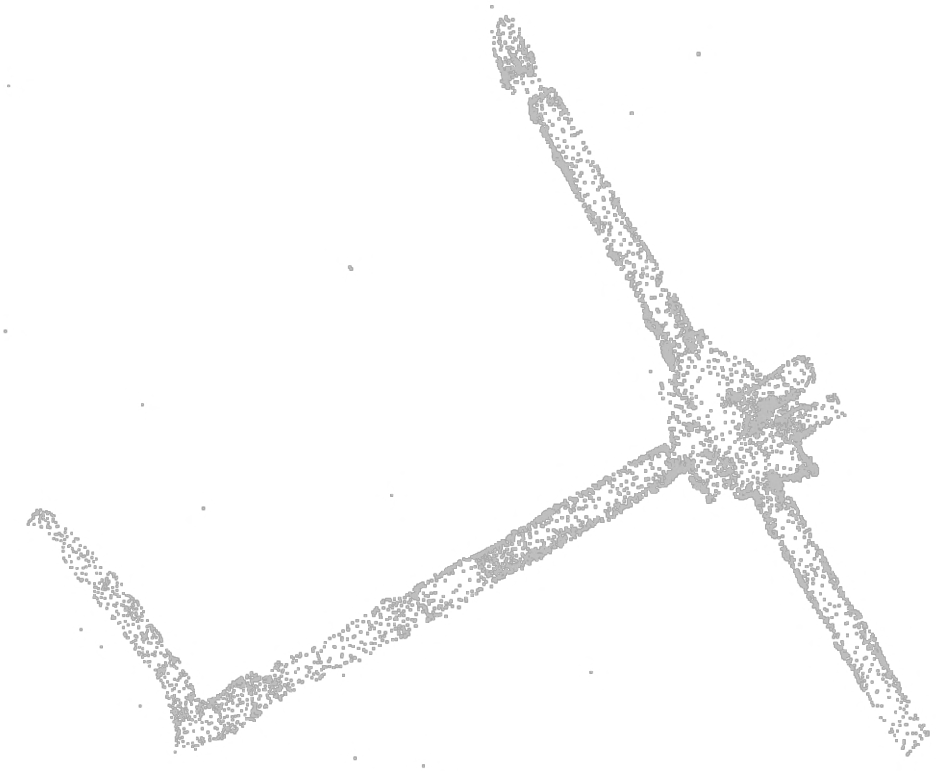
(b) Reconstructed Mesh

Figure 7.28: A 650 Meter Walk on Bahnhofstrasse in Zurich

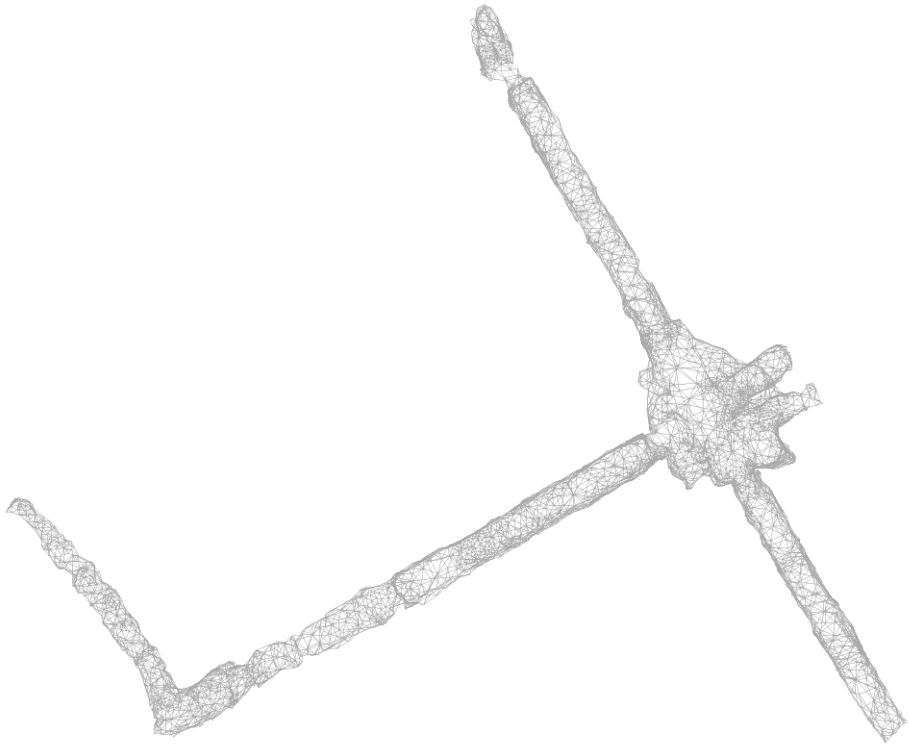
(a) Reconstructed Mesh

(b) Reconstructed Mesh with Texture

Figure 7.29: Niederdorf in Zurich

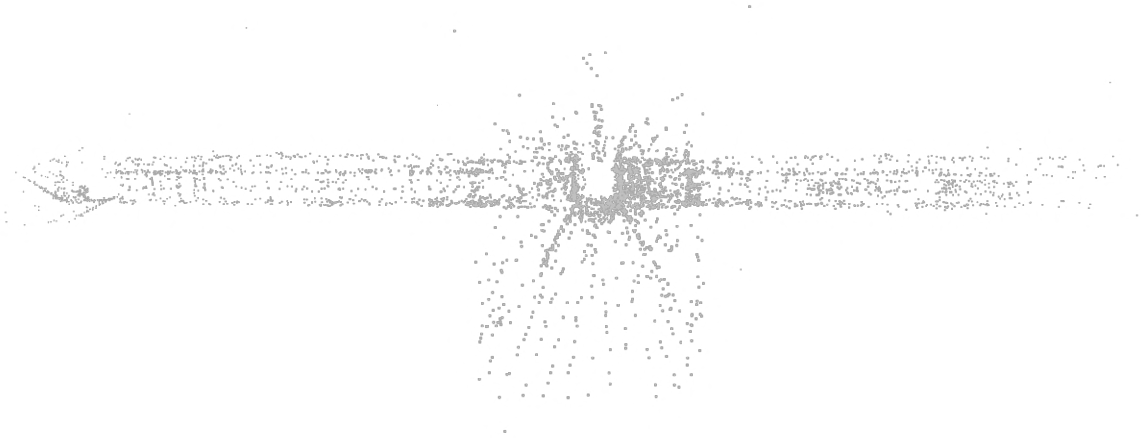


(a) SLAM Map

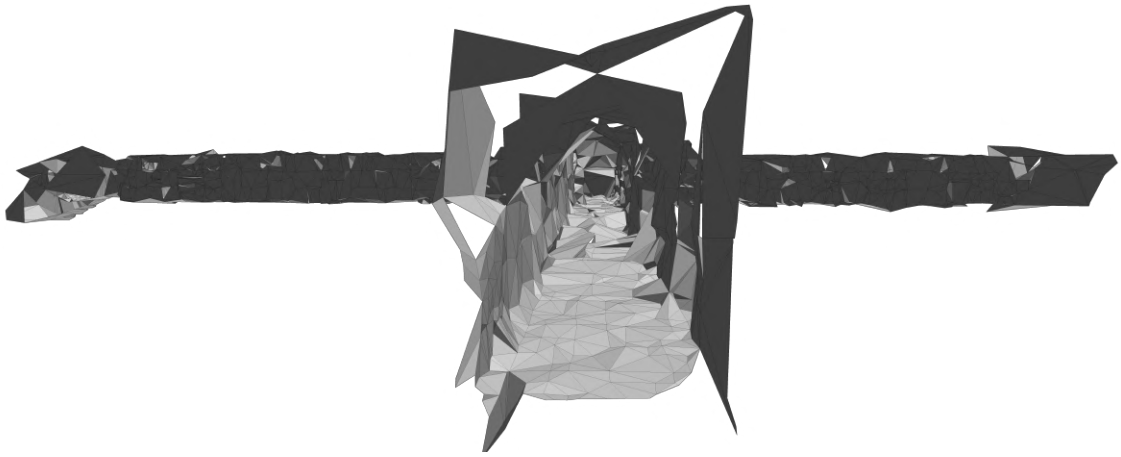


(b) Reconstructed Mesh

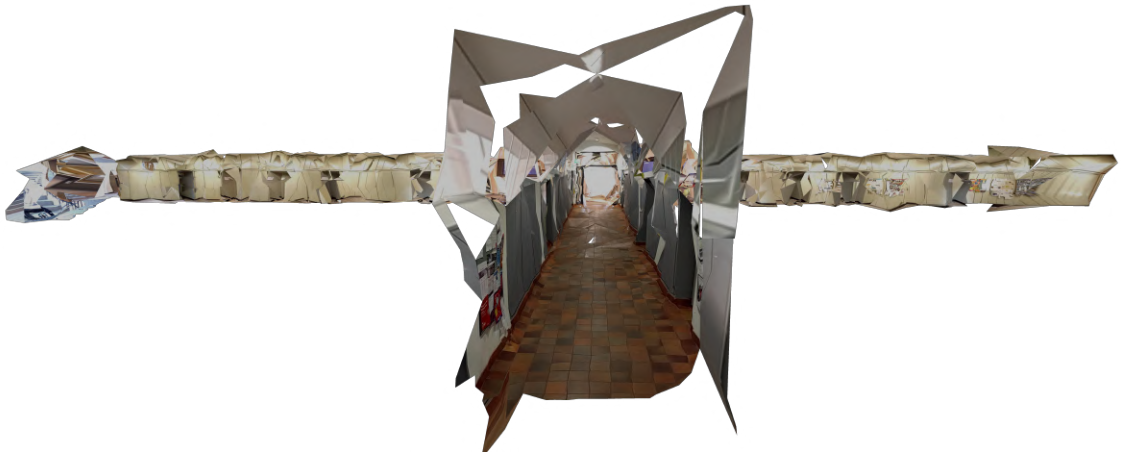
Figure 7.30: Top View on Corridors of CAB Building of ETH Zurich



(a) SLAM Map



(b) Reconstructed Mesh



(c) Reconstructed Mesh with Texture

Figure 7.31: CAB Building of ETH Zurich

8 Conclusion

Concluding, the work developed in this thesis proposes a set of algorithms in a real-time reconstruction pipeline. Briefly captured, the proposed methods describe a probabilistic tetrahedra carving process based on tracing rays between cameras and visible tetrahedra from these cameras in an online fashion. ProFORMA, which also employs a probabilistic carving technique, does not utilise the reprojection errors of vertices, as is the case in the proposed formulation. This enables the tetrahedra probabilities to be computed in a more sensible manner and with better accuracy than proposed by ProFORMA [3].

Further, improvements include the formulation of the tetrahedra carving process as an energy minimization problem, for which a graph cut algorithm is used to efficiently solve. This is an improvement compared to assigning tetrahedra labels based on a threshold probability, as is the case with ProFORMA. Photoconsistency and surface area costs are included in the energy functional, which contribute to more robust and accurate reconstruction. Contrary to related works [4] that also implement similar cost functional, the proposed implementation is designed for an incremental reconstruction pipeline. This allows for automatically compute weightings of individual costs based on statistic values form previous frames, instead of having to manually and statically define them, as is the case in [4]. The pipeline in [4] does also not include a probabilistic visibility based carving formulation.

The spike removal algorithm is also included as a new component in the proposed pipeline, and in general seems to perform well in clearing remaining artefacts in the volume reconstructed by the graph cut.

The last chapter which shows reconstruction results on real data, shows that the algorithm is capable of reconstructing fully textured surface models from video sequences of various scenes with varying physical size and content. The reconstructed surface is computed at every frame and can be rendered live on-screen when deployed on a smartphone.

In the greater context, this enables users to very quickly and spontaneously reconstruct a variety of scenes. Because of the live rendering on the phone's screen, there is little risk of insufficient reconstruction due to insufficient exploration. Users can use the feedback from the live rendered reconstruction to decide on the spot if certain regions require more detail to be captured.

Bibliography

[1] P. M. Kolev K. Tanskanen P., Speciale P., "Turning mobile phones into 3d scanners," *IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pages 3946-3953*, 2014.

[2] K. K. M. L. C. F. S. O. P. M. Tanskanen, P., "Live metric 3d reconstruction on mobile phones," *IEEE International Conference on Computer Vision (ICCV), pages 65-72*, 2013.

[3] D. T. Pan Q., Reitmayr G., "Proforma: Probabilistic feature-based on-line rapid model acquisition," *Proceedings of the British Machine Vision Conference (BMVC), pages 112.1-112.11*, 2009.

[4] K. R. Labatut P, Pons J.-P., "Efficient multi-view reconstruction of large-scale scenes using interest points, delaunay triangulation and graph cuts," *IEEE International Conference on Computer Vision (ICCV), pages 1-8*, 2007.

[5] K. V. Boykov Y., "An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision," *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI), pages 1124-1137*, 2004.

[6] B. G., *Dr. Dobb's Journal of Software Tools.*

[7] The CGAL Project, *CGAL User and Reference Manual*, 4.6.2 ed. CGAL Editorial Board, 2015. [Online]. Available: http://doc.cgal.org/4.6.2/Manual/packages.html

[8] P. Maur, "Delaunay Triangulation in 3D, State of the Art Concept of Doctoral Thesis," Ph.D. dissertation, University of West Bohemia in Pilsen, 2002.

[9] Yuanxin Liu , Jack Snoeyink, *Combinatorial and Computational Geometry, pages 439-458, A Comparison of Five Implementations of 3D Delaunay Tessellation.* MSRI Publications, 2005.

[10] W. S. et al., "The visualization toolkit," *3rd Edition. Kitware, Inc.*, 2003.

[11] Changchang Wu, *VisualSFM: A Visual Structure from Motion System*, 2011. [Online]. Available: http://ccwu.me/vsfm/

[12] C. B. S. S. Changchang Wu, Agarwal S., "Multicore bundle adjustment," *IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pages 3057-3064*, 2011.

[13] D. J. S. D. S. R. Seitz S.M., Curless B., "A comparison and evaluation of multi-view stereo reconstruction algorithms," *IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pages 519-528*, 2006.

[14] V. C. L. I. CNR, *MeshLab.* [Online]. Available: http://meshlab.sourceforge.net/

[15] M. G. Simon Fuhrmann, Fabian Langguth, "MVE - A Multi-View Reconstruction Environment," in *Proceedings of the Eurographics Workshop on Graphics and Cultural Heritage (GCH)*, 2014.

[16] M. J. M. M. Mur-Artal, Raúl and J. D. Tardós, "ORB-SLAM: a versatile and accurate monocular SLAM system," *IEEE Transactions on Robotics*, vol. 31, no. 5, pp. 1147–1163, 2015.