NTNU
Det skapende universitet

Completing a model based on laser scan generated point cloud data

Ann-Silje Kirkvik

Master i datateknikk
Oppgaven levert: Mai 2008
Hovedveileder: Torbjørn Hallgren, IDI

Norges teknisk-naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

Oppgavetekst

A model based on laser scan generated point cloud data has holes mostly because of occlusions. Find and implement methods to fill such holes to complete the model in way that as much as possible maintain a correct and visually pleasing result. Use an existing model of the Nidaros Cathedral as a test case.

Oppgaven gitt: 14. november 2007
Hovedveileder: Torbjørn Hallgren, IDI

Abstract

This paper is a master thesis for the Department of Computer and Information Science at the Norwegian University of Science and Technology, spring 2008. It is a study of hole filling in three dimensional surface models obtained from scanned real world objects.

The goal of this project is to find solutions that are capable of filling an incomplete model in a plausible and visually pleasing manner. To reach this goal both theoretical studies and practical testing were performed.

This paper presents a theoretical foundation, needed to gain a greater understanding of the problem, and the results from the testing phase. This knowledge and experience is then used to present a possible solution to the hole filling problem.

The conclusions of this project is that automatic procedures, that are thoroughly documented in the literature, fails to perform in a satisfactory manner when the data set becomes too complicated. The Nidaros Cathedral is such a difficult data set, and will require a customized and user guided solution to met the goals of this project.

Preface

This master thesis is the result of the diploma work of Ann-Silje Kirkvik for the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU) during autumn of 2007 and spring of 2008.

The assignment was to find and test hole filling techniques for scanned three dimensional models. The test case for the project is a model of the Nidaros Cathedral in Trondheim, Norway. This model has been the subject of previous projects and a master thesis at IDI. The supervisor of this project was Torbjørn Hallgren and the additional information on the Nidaros project was supplied by Kristin Bjørlykke at The Restoration Workshop of Nidaros Cathedral. A special thanks to all those involved in the project.

Trondheim, May 28, 2008

Ann-Silje Kirkvik

Contents

List of Figures

Chapter 1

Introduction

This project addresses one of the problems that occur during surface reconstruction of 3D scanned model data, namely hole filling. The test case used for these studies is a scanned version of the Nidaros Cathedral. This chapter presents the problem that give rise to this project and give an overview of the remaining document.

1.1 Motivation

A popular approach to modeling complex real world objects is to perform a three dimensional scan of the object and use this information to derive a virtual model. This, however, is not a trivial process. One of the problems that arise are due to missing data in certain regions of the model. It can be multiple reasons why data are missing.

The object may be occluded by other objects or even self occlude. Due to the practical implications of scanning a model, only a limited number of scans are usually performed. These scans are in almost all cases insufficient to cover the entire model and holes will arise. The size of these holes will vary greatly based on the number of scans and complexity of the model surface. The surface may also have reflectance properties that make it difficult to record data in certain areas. Usually such holes are undesirable and needs to be filled.

1.2 Problem Statement

The purpose of this paper is to investigate possible methods to fill the holes that are present in scanned 3D models. A lot of research has been put into this field and several algorithms have been proposed. The information in the input data and the properties of the final result varies greatly. To analyze these methods experiments need to be performed in the form of implementations of existing methods. Several criteria for a hole filling system will be defined and a proposed hole filling system will be evaluated in relation to these criteria. It is therefore also a part of this project to define these criteria.

1.3 Scope

This project is about hole filing techniques for data models obtained from 3D scans of real world objects. Investigating methods that can accomplish this is both a theoretical and practical task. Theoretic background is needed to get a greater understanding of the problem and possible solutions. Practical testing is needed to confirm or reject the usefulness of existing theories on real model data. It is not within the scope of this project to test every available method, but rather aim for a understanding of how different classes of methods perform in a given situation.

The test case is a difficult data set and by identifying how some algorithms fail to perform new and better methods can be developed. By defining a set of criteria for the hole filling solution, it becomes easier to evaluate different approaches. These criteria is defined with the intension of having different weights for different test cases. The project aims to fill as many holes as possible, but with the limited time available it is only realistic to partially meet this goal and rather show how remaining holes can be filled.

1.4 Overview

This section gives an overview of this document. Chapter 1 is this chapter. It described the problem this project is aiming to solve, the motivation for creating such a project and the limits of the project. Chapter 2 briefly mentions some similar projects and gives a summary of the existing literature on the subject of hole filling. Chapter 3 defines a set of criteria for a solution that

is aiming to solve the hole filling problem in scanned 3D models. Chapter 4 describes the testing phase. It mentions the environment in which the testing was done, the implementation step and the results that were produced. Chapter 5 outlines a scheme for creating a interactive hole filling tool and evaluates how this tool meets the criteria from chapter 3. Chapter 6 talks about possible future work on this project and the conclusions that were drawn.

Chapter 2

Background

This chapter is intended to give an overview of pervious work and research on the subject of hole filling in three dimensional computer models. It should provide some theoretical background, which is needed to evaluate and compare different approaches. First, a few similar projects are presented, followed by a categorical summary of the methods found in the literature. The issues addressed are input and data representation, algorithms and user interaction.

2.1 Projects

The test case that will be used throughout this paper is a scanned 3D model of the Nidaros Cathedral in Trondheim, Norway. Previous work on this data set include gathering of data and visualization of the cathedral. More details on this project can be found in [25]. Hole filling is only briefly addressed in that project and leaves a perfect opportunity to investigate the model further in this paper. [7] describes some of the goals for scanning the Nidaros Cathedral. These include visual presentations, reconstruction work, research and information storage. Refer to the report for further details.

The Nidaros cathedral is only one of many potential test cases. In the literature models from The Digital Michelangelo Project [26] and The Stanford 3D Repository [34] are frequently used. The "Stanford Bunny" is particularly popular. The Stanford 3D Repository is a collection of scanned models available for download. The Digital Michelangelo Project scanned multiple sculptures and other work by Michelangelo in 1998-1999. These scans has been the source of several papers on model reconstruction and thereby also

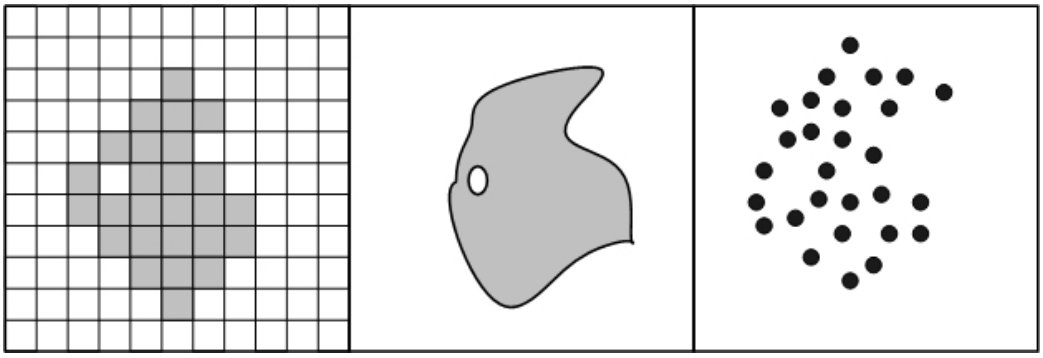

Figure 2.1: Volumetric, surface and point cloud representations

hole filling.

2.2 Methods

There are several ways to fill the holes in a scanned 3D model. These methods vary in how the input data is represented, the amount and type of information needed, the level of user interaction supported and finally how the holes are filled. Below follows an attempt to classify these methods and present the different mechanisms that should eventually lead to hole filling.

2.2.1 Input and Data Representation

Most hole filling algorithms start out with a partial mesh representing the surface. However, many algorithms work with a different data representation like an implicit surface representation or a volumetric representation. Some algorithms transform the partial input mesh into the desired representation while others construct the data structure as a part of the conversion from range scans to 3D model. The former approach has the advantage that the scanning and conversion to 3D model is independent of the hole filling method. The latter approach ties these processes together and optimizes performance by leaving out intermediate steps of data conversion from one representation to another.

The most common data representations found in the literature were volumetric data, surface data or point cloud data. Surface data can be described by a function or by a polygonization of data points. The latter case is of

most interest to hole filling, because functional data usually already represents hole free surfaces. Figure 2.1 illustrate a two dimensional version of the different types of data representations. Volumetric and triangulated data are explained in more detail below.

Volumetric data

One way of representing a surface is by partition the space into two distinct volumes, namely the volume inside the surface and the volume outside the surface. The surface itself is then defined as the boundary between the inside and outside volume. This definition has numerous advantages. The final surface is guaranteed to be closed, watertight, non-self-intersecting and 2-manifold. Many applications require input meshes to have these qualities to function properly.

There are several ways to partition the space and an octree approach is commonly used. The space is only subdivided in the areas of interest, saving both memory and computation time. A regularly spaced grid is also used in some cases, like in [14]. Another typical example is data from medical imaging, such at MRI or CT scans, which is already voxelized on a uniform grid. Other partitioning schemes are also possible, like BSP-trees, but are usually more involved and complicated and will not be given any more attention in this paper.

Holes are typically represented by some value that characterizes that portion of the space as unknown. In [14] the surface is approximated using a signed distance function. Inside and outside are represented by values in the range of [-1,1] (positive outside, negative inside) where 0 represents the surface itself. A confidence value in the range of [0,1] is added to each point. The confidence typically decreases near holes and boundaries and are set to 0 in the regions not observed by the scanner. As explained in [13] the confidence values are derived from the viewing direction of the scanner and boundary information. This is intended to reflect the uncertainty in data recorded at grazing angles and the fact that noise is usually more common at the boundaries of a surface. This method therefore relies on confidence data being available.

In [32] the original surface is embedded in an octree. The cubes in the octree are classified either as **blank**, **I\O** or **hole**, and octree subdivision is carried out until the entire volume can be labeled according to this scheme. As a consequence the octree is only expanded in areas affected by the holes.

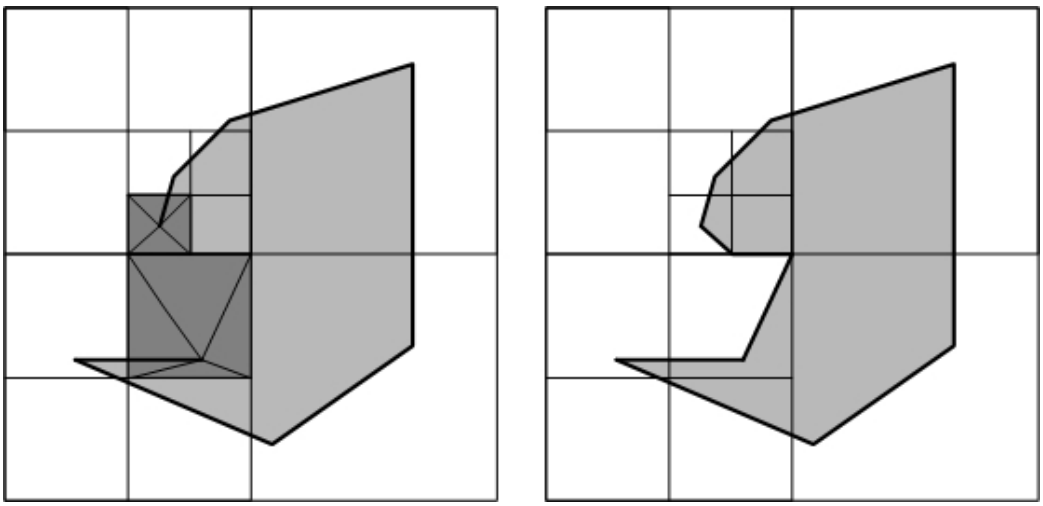

Figure 2.2: Atomic Volumes for Mesh Completion

An example is shown in the left half of figure 2.2. A white cube represents a blank octree cell, a dark gray cell represents the hole cubes and the cells that are partially white and light gray are the I\O cubes. The filled surface is shown to the right.

A graph is then created based on these volumes. A node in the graph represents an atomic volume. A blank cube is already atomic and represented by one node. An I\O cube consists of two atomic volumes separated by a completely defined surface. The inside and outside volumes in this cube are then represented by one node each. A hole cube is tetrahedralized into several atomic volumes by creating edges from the boundary vertex to the cube vertices. Each tetrahedron is represented by a node. Graph edges are created between neighboring atomic volumes and have a weight representing the cost of adding a face between these volumes. To partition the space as described here the original triangle mesh must have consistently oriented normals.

A partial triangle mesh with consistently oriented normals may not always be available. A method to convert a polygon soup into a volumetric representation may be desirable. [23] presents such a method. Here, polygons are tested for intersection with the octree grid. The edges in the octree that intersects with the polygons are marked as intersection edges. If it is desirable the points of intersection and triangle normals can be stored too.

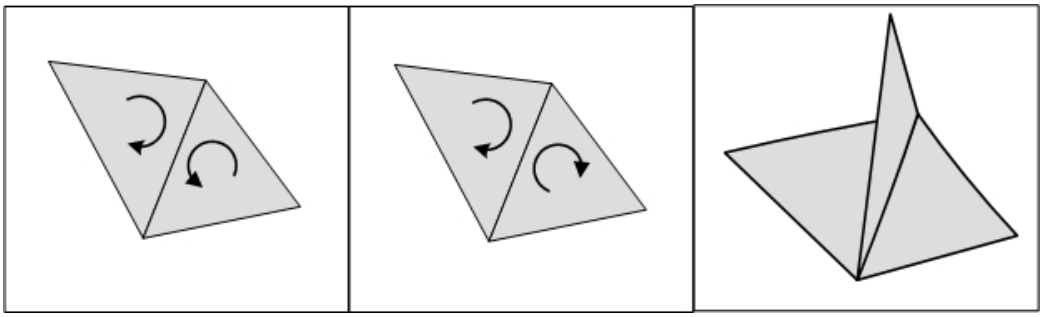

Figure 2.3: Edge configurations

Intersection edges are then given a sign to each of its endpoints such that the edge represents a sign change. A consistent sign configuration will separate the inside volume from the outside volume. However, as long as the model contains holes such a configuration does not exists and holes must be patched.

Triangulated data

Some algorithms also work directly on the original triangle mesh. Other polygon meshes are also possible but triangle meshes are by far the most common. A triangle mesh is usually required to be consistently oriented, manifold and/or connected. The hole filling process aims to make this triangle mesh watertight, closed and/or connected.

The convention is to orient the triangle edges in a counter clockwise order, even though clockwise ordering will give the same results as long as the mesh is consistently oriented. An edge can be shared by several triangles. If it is only adjacent to one triangle, it is designated as a boundary edge. If it is shared by two triangles, it is an inner edge of the mesh. For a mesh to be consistently oriented the inner edges has to be defined in opposite directions for each of the triangles it belongs to. If an edge is shared by more than two edges the mesh is no longer 2-manifold. Figure 2.3 shows an inner edge with these different configurations. To the left are two triangles that have different orientations, resulting in a conflict at their shared edge. The middle figure shows the correct configuration. To the right are an edge shared by three triangles and is therefore not 2-manifold.

A mesh may or may not be connected. If a component of the mesh is separated from the rest of the mesh it might be considered noise and is

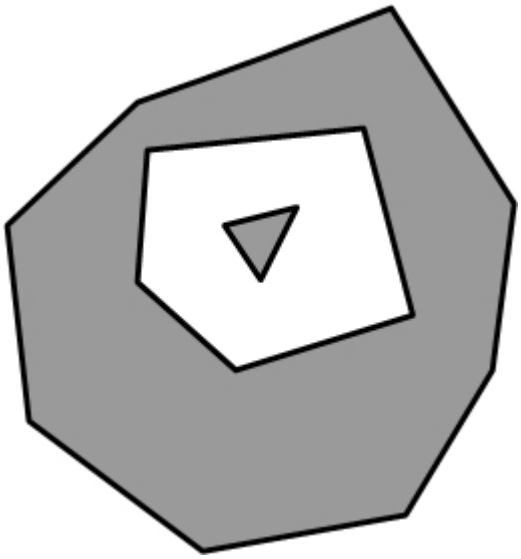

Figure 2.4: Hole with an island

suppose to be filtered out. These components may also be an important part of the mesh. If so, the gap that separates it from the rest of the mesh should be filled. Typical separated components are islands in hole areas.

A hole in a triangle mesh is identified as a loop of connected boundary edges. However, not all such loops are holes. An island is enclosed by such a loop of boundary edges. If the triangle mesh is not closed the boundary of the mesh will also fit this classification. To separate between these two cases and identify which of the boundary loops should be filled, user interaction is usually required. Figure 2.4 displays a hole with a disconnected island inside. The is a total of three boundary loops, but only one that encloses a hole. The other two encloses the geometry of the sample.

Hole filling algorithms that work on this kind of data set usually aim to triangulate the hole. The methods are normally local and use information from the triangle mesh near the holes, often referred to as the vicinity of the hole, to estimate the new triangles.

2.2.2 Hole filling techniques

Once a hole has been identified there are several ways to fill the hole. Different models and different types of holes usually require different approaches to

get a satisfactory solution. The interpretation of satisfactory is also highly subjective and case sensitive. Below follows a short description of some of the methods that occur in the literature. A thorough presentation is outside the scope of this project. For details refer to the respective articles.

Radial Basis Functions

Radial Basis Functions (RBF) are all functions that satisfies $\phi(x, c) = \phi(\|x - c\|)$, where c is referred to as a center. For the purpose of approximating functions with RBF variations of the formula in equation 2.1 are normally used. For more information refer to [17].

$$y(x) = \sum_{i=0}^{N} \omega_i \phi(\|x - c\|) \tag{2.1}$$

In [9] RBF are used to model an entire 3D model. This is computational expensive and previously not considered a viable option. With approximation techniques, center reduction and fast evaluation methods this had been made computationally feasible even for larger models. As a bi product of this modeling technique holes or under sampled regions in the data are smoothly filled by the radial basis function. This method is also promising for remeshing, compression and smoothing algorithms.

[8] used Local RBF to fill holes in a triangulated mesh. A group of points surrounding the hole, also referred to as centres, are selected to calculate a function that approximates the mesh around the hole. This function is also extended to include the hole itself. By evaluating this function in the hole region new points can be added to the mesh. Finding this function is not a trivial task and the computation time is highly dependant on the number of points used to describe the region around the hole. An iterative approach is selected that adds new points to this set until the function satisfies the desired degree of precision. Points in the vicinity of the hole which are not used to calculate the function itself, are used as control points to determine the level of accuracy.

RBF are smooth interpolation functions that represent a surface implicitly. For the purpose of visualization and further processing by different applications a polygon model is often needed. A subsequent surface extraction algorithm must therefore be applied to generate the polygon model. Methods such as Marching Cubes [12] serves this purpose. Also, since it

is a smooth interpolator sharp features are poorly reconstructed with this method. [10] tries to eliminate this problem by applying a sharpness filter on the polygonized filled hole. Polygon normals are analyzed and readjusted to enhance sharp edges that have been smoothed by the radial basis function.

Moving Least Square

Moving least square is a method for approximating a smooth function to a set of data points by something similar to a minimum least square procedure. But instead of a global solution the function is related to a chosen point which causes the function to *move* as this point is changed. For the purpose of surface reconstruction points are usually projected onto a local reference frame specific for the problem domain, also referred to as the moving least squares projection procedure. Equation 2.2 shows the error of fitting the function s through the function values f_i of each point p_i, and where p is the new resampled point, the point that moves. The goal is to minimize this error by finding an appropriate function s for each point p.

$$E_p(s) = \sum_{i=1}^{N} \omega_i(p)(s(p_i) - f_i)^2 \tag{2.2}$$

In [41] the entire vicinity of a hole is transformed into a local coordinate system defined by the points in the vicinity. By projecting the vicinity points and the mesh down to a two dimensional plane in this coordinate system the hole filling process has been simplified by the reduction of one dimension. Points are added in a grid like structure in this plane and the moving least square function is used to determine the coordinates of these new points in the local three dimensional system. Finally the points must be transformed back into the global coordinate system. For this procedure to work the holes must have a fairly simple disk shaped topology that can be projected onto a plane without loss of information.

[39] utilizes the moving least square projection procedure too, but is not limited to topological simple holes. The method consists of alternating steps of adding vertices and edges. Before any edge or vertex is added a local reference frame of the neighborhood the current edges is calculated. New edges are added as long as they do not intersect the current geometry and similarly are new points added as long as their new triangle does not violate existing geometry. Figure 2.5 shows an edge and a triangle, respectively,

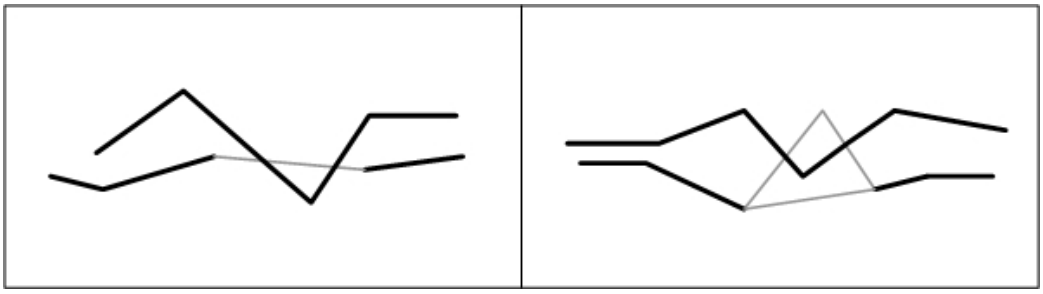

Figure 2.5: Edge and triangle causing self-intersection problems

that will be discarded by this method to avoid self-intersection. Moving least square approximations determine the final position of the new points on the surface.

Inpainting

There is a class of methods that try to fill holes by copying information from similar regions on the object surface. The term inpainting is normally associated with 2D image restoration, but it originates from restoration of traditional artworks such as paintings. The problem is basically the same - filling in the missing pieces - so the term is also used for 3D surface completion, even though the methods are a bit more complicated.

Objects often have some kind of surface structure or a hole might (partially) cover a distinguishing feature of the object. A smooth filling of the hole would not look plausible and even disturbing. The general idea of the inpainting techniques is to analyze the model for structures or features that might plausibly fill the hole if copied into the hole region. There are, however, some difficulties that must be addressed to employ such a method.

One of the problems is how to identify similar regions. It is not obvious what the term *region* means or how to compare two regions to determine their similarity. These methods also work with data that are not triangulated, leaving the identification of holes more complicated. Simply copying a similar region into a hole area is neither satisfactory. Usually, a series of translation, rotation, mirroring, and slight deformation operations are required. If the deformations are too great the similarity value will become invalid and the operation becomes meaningless.

In [36] the model is embedded in a octree data structure. Cells are la-

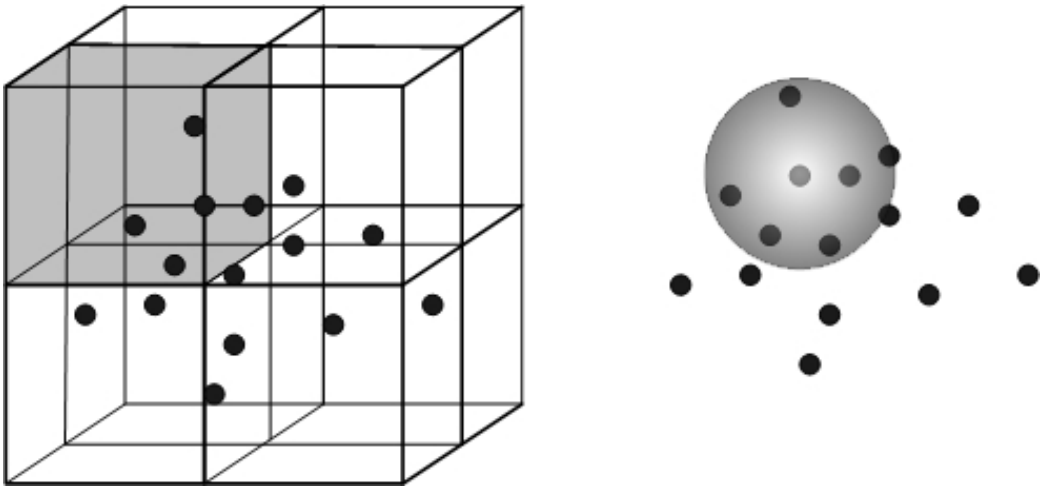

Figure 2.6: Regions

beled either *void* (empty), *valid* or *invalid*. Each cell contains a local surface approximation. The goal is to get rid of all the invalid cells. A cell is labeled invalid if the local surface approximation error exceeds a limit or the cell contains too few data points. In this case a *region* is a cell in an octree, as shown to the left in figure 2.6. For an invalid region the algorithm searches all equally sized valid octree cells for the most similar cell. The similarity measure or shape descriptor of a region is a bit involved. For each cell a vector is constructed that contains values representing the different properties of the cell. Comparing these vectors determine the cells similarity. Once a similar cell is found it is pasted into the invalid cell and fitted through a series of geometric rigid and non-rigid transformation. Changes in a cell are propagated to the corresponding cells in the octree and the process continues until all cells are valid and the surface is complete.

[5] and [4] also describes such a method. Here, a region is defined as all points within a certain radius of a sample point, shown to the right in figure 2.6. This point set is then resampled on a regular grid in a geodesic polar coordinate system. A vector containing the height values of the these resampled points are used as a shape descriptor for the region. The mean square error between two such descriptors determines their similarity. The hole filling algorithm is hierarchical. The data set is smoothed or subsampled into several data sets with a decreasing level of detailed information. Hole filling is performed on the coarsest level first. The filled holes on the coarser

levels are used as guidance surfaces for the finer level. This hierarchical approach allows the algorithm to detect similarities on different scales. Coarse structures are identified on lower levels and the finer details are derived later in the progress. Since similar regions might be found at completely different areas of the model for different scales the algorithm is quite versatile.

Both these methods use automatic detection of similar regions. In some cases, however, this approach fails. The algorithms have no way of understanding the semantics of a model and may incorrectly copy meaningless regions into a hole because it fulfills the similarity requirement. An alternative is to let the user indicate which regions to copy from. The model itself may also contain an insufficient number of sample regions to be able to locate good similar candidate regions. A normal practice is to mirror and rotate the sample regions to increase the number of samples. An additional technique is to scan several similar objects to create a more extensive database. Notice how the method in [5] and [4] allow regions to overlap while the method in [36] does not.

There is also a slightly different method that potentially can fill these holes in a meaningful way. It is similar to the inpainting techniques and will therefore be outlined in this section. A description of the method can be found in [42]. Instead of using the model geometry to fill holes, photographic images are used to estimate the missing geometry. The images are recorded together with the scanning process, which makes the subsequent alignment of photographs and scans a lot easier. The model is projected onto each image and a mapping from triangle normals to image intensities is established. A selection of these mapped values constitute a training set. This training set is then used to estimate normals where geometry is missing in the model. Finally all the normals in the hole region are integrated to calculate the missing surface.

Geometric Filling

Some methods aim to triangulate the hole polygon directly. However, as stated in [27], finding a non-self intersecting 2-manifold connected triangulation of an arbitrary 3D polygon is a NP-complete problem. See [11] for more consequences of this fact. A slightly less strict solution is therefore normally acceptable. In lack of a better term these methods are categorized as geometric hole filling procedures.

The method described in [27] is probably one of the most frequently

mentioned methods of this kind in the literature. The algorithm works on connected, oriented and 2-manifold meshes. As a consequence it does not consider any islands in the holes. The filled hole may also cause the surface to self-intersect. Despite of these problems it is still a popular method. Once a hole boundary is detected a minimum weight triangulation is calculated. The are several options regarding the weightening criteria. Minimum area triangulations or minimum maximal dihedral angle triangulations are the ones mentioned in the article. After the triangulation is determined it is refined to fit the edge length and sampling density of the surrounding mesh, yielding a delaunay-like triangulation. Finally, the mesh is smoothed. One important note is the time complexity of the triangulation step. Its time complexity is $O(n^3)$, where n is he number of vertices on the hole boundary. This efficiently limits the size of the holes this method is useful on.

The method proposed in [28] is better suited to deal with larger holes. Also here, is a connected, 2-manifold and oriented mesh assumed. If these criteria are not met the algorithm can perform a preprocessing step to assure these qualities. The same preprocessing procedure can be utilized by the method mentioned above. Based on the hole boundary and the connectivity information in the mesh a list data structure is made for each hole. Each entry contains a boundary vertex and pointers to its successor and predecessor vertices on the hole boundary. Figure 2.7 illustrate the concepts used in this algorithm.

Starting with a random vertex, p, a plane is defined by the cross product n of the two edges to its successor and predecessor. The list is then traversed in both direction until a point is discovered that does not lie in that plane. Once the search has been terminated in both ends, an edge is added to connect the termination nodes. All the vertices discovered by this search lies in the same plane and can be triangulated using a simple 2D algorithm, such as the protruding point algorithm described in [43]. This process is recursively repeated on the remaining hole boundary until the hole is filled. Subsequent mesh refinement and smoothing is also performed here.

[19] is another method that fills holes by iteratively adding new vertices and edges along the hole boundary. What separates this method from the others is its ability to recreate sharp edges. Before hole filling is performed all edges are classified as either smooth, sharp or indefinite by analyzing the normals of surrounding triangles. An indefinite edge is a boundary edge while smooth and sharp edges are inner edges. If a vertex is adjacent to a sharp edge is it classified as a sharp vertex. When the hole filling algorithm runs

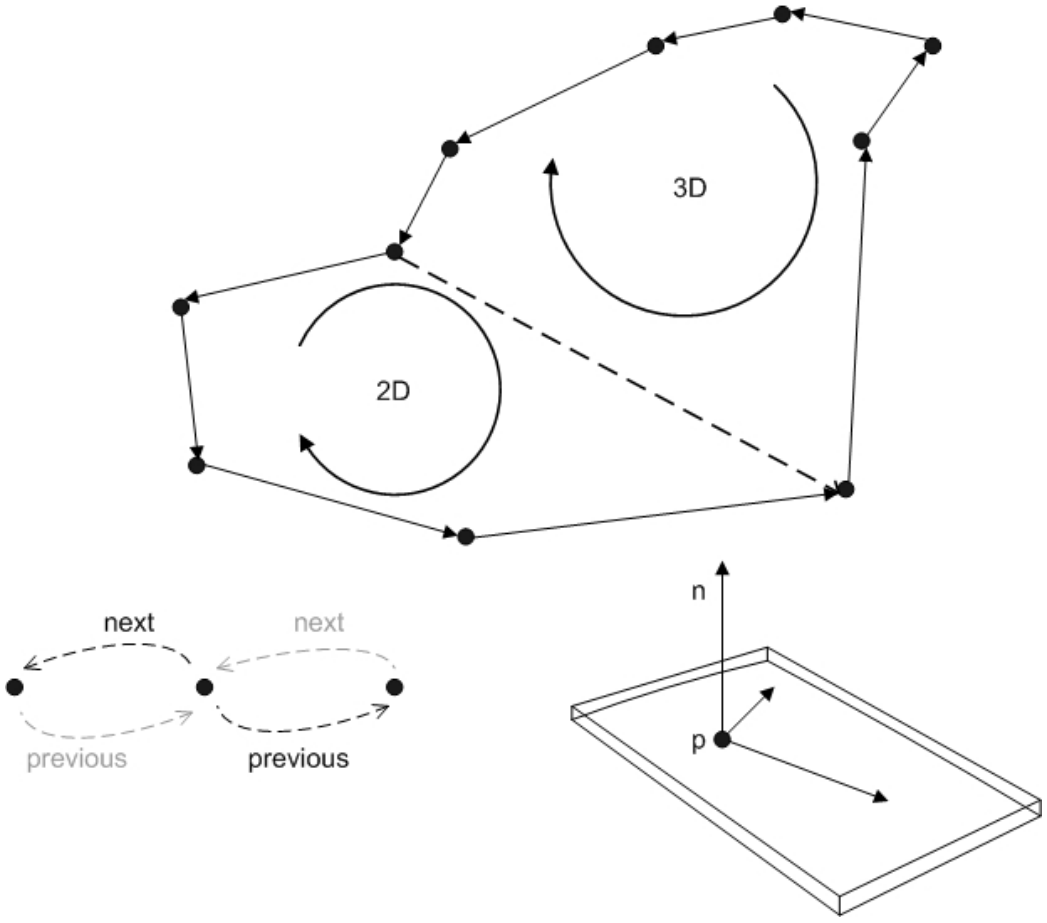

Figure 2.7: On the way to water-tight mesh

into a sharp vertex is employs a different strategy for adding vertices in that area. The new vertices are added in such a way that the sharp edge in the surrounding mesh is extended into the patching mesh. As a final result the filled hole preserves the sharp edges that were indicated by the vicinity of the hole.

Volumetric Filling

Volumetric methods are closely related to how the data is represented. This is described in section 2.2.1. A popular method is the volumetric diffusion algorithm described in [14]. It was developed at the Stanford University in relation to The Digital Michelangelo Project, presented in [26]. Source code is available at [21]. The program only accepts .vri files, which are created with the model reconstruction technique described in [13].

As mentioned in 2.2.1 the space is partitioned into a regular grid with signed distance function values defined near the surface. The surface is the zero set of this function. Weights, representing confidence, are assigned to each function value. These weights are derived from the algorithm in [13] which is also used to create the signed distance function. This function can also be derived from other sources. The basic algorithm is described as a series of alternating steps with blurring and compositing. Equations 2.3 and 2.4 illustrate these steps,

$$(\hat{d}_i, v_0) = h * (d_{i-1}, v_{i-1}) \tag{2.3}$$

$$d_i = w_s d_s + (1 - w_s)\hat{d}_i \tag{2.4}$$

where $(d_0, v_0) = (d_s, [w_s > 0])$, $w_s \in [0, 1]$ is the confidence value, d_s is the original signed distance function, h is the convolution filter, and v_i is a boolean value indicating whether the function is valid at that point.

As a result of this approach the boundary data is diffused into the empty region until it finally closes the hole. The new surface patch blends smoothly with the rest of the model, but it does not necessarily interpolate original data. The method is said to guarantee convergence, but at a slow rate. This is confirmed in [42], where the algorithm fails to converge within 48 hours. Only local information is used, which enables the method to scale nicely with large input models, but can cause problems when boundary edges point away from the hole.

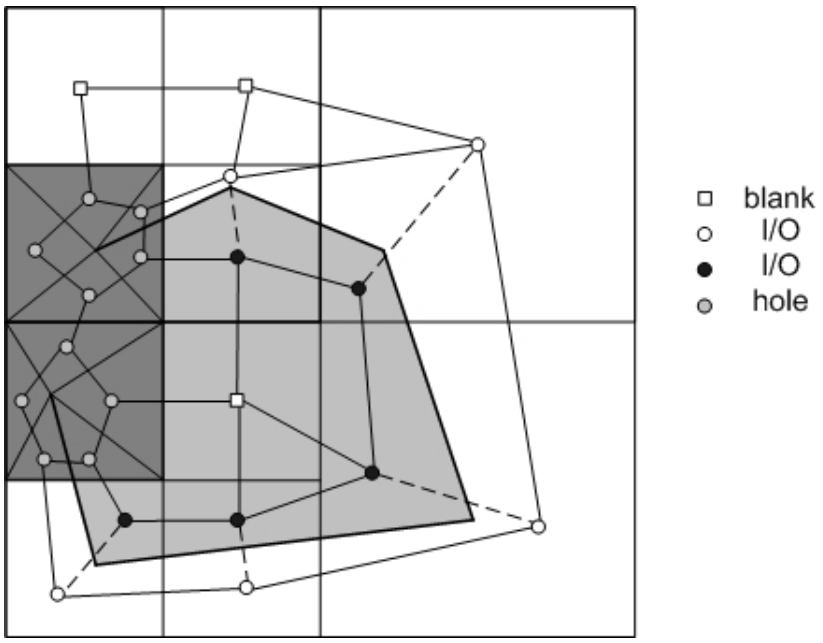

Figure 2.8: Atomic Volumes for Mesh Completion (2)

When convergence is a problem or interpolation of original data is required another method might prove more useful. One such method is described in [32]. The data structure is described in section 2.2.1. After labeling all the octree nodes a graph is created. An example is shown in figure 2.8. The surface in a I/O cube is already defined and should not be altered. Its graph edge is therefore only indicated with a dotted line. The algorithm seeks to cut this graph into two parts using a min-cut algorithm. This cut represents the separation of inside and outside volumes. All nodes connected to the source node is labeled as inside while the nodes connected to the sink node is labeled as outside. Source and sink nodes are not shown on the figure. Also, remember that a node represents an atomic volume and that an edge represents the cost of adding a face between the connected nodes. Only neighboring nodes share an edge and the sum of all node volumes is the same volume as the octree root. Being a global method the performance depends on the size of the model and the quality of the repaired hole depends on the octree depth. The octree is only expanded near the hole boundary and a consistently oriented surface model is required to consistently subdivide all areas effected by a hole.

A consistently oriented mesh or signed distance functions may not be readily available. The method in [23] works with polygon soups. An executable version of this algorithm can be found at [33]. It reads .ply files and outputs the result in the same format. The algorithm is performed in three steps: scan conversion, sign generation and surface reconstruction. Section 2.2.1 briefly describes the scan conversion and partially the sign generation step. When a polygon intersects an octree edge, that edge is marked as an intersection edge and a sign configuration is stored with it. Since an intersection means the edge crosses over the inside-outside boundary an intersection edge must have different signs associated with its two endpoints. All intersection edges should therefore exhibit a sign change. By finding a consistent sign configuration the holes in the model are also filled. This is based on the fact that a consistent sign configuration only exists if (and only if) the dual surface of the set of intersection edges does not have any boundary edges. The last step of surface reconstruction is shared with other volumetric methods, such as [14]. A contouring algorithm such as Marching Cubes are needed to extract a polygon representation of the surface.

Other Approaches

There are some surface reconstruction algorithms that directly produce hole-free surfaces. These algorithms are normally not considered hole filling algorithms, but their output is very similar to what the hole filling algorithms aim for. Because of this they are worth some attention. Any detailed presentation of these methods is not intended, only the general idea of how the methods work are mentioned.

Given a point sample from an object's surface the goal is to produce a watertight surface model of this object. Algorithms like *The Power Crust* [2],[3] and *Tight Cocone* [15] accomplish this by exploiting the delaunay triangulation and the voronoi diagram. These two constructs are thoroughly researched and provides numerous theoretical guarantees under certain assumptions. One of the earlier examples of these ideas can be found in [1].

The power crust is defined as the surface separating inner and outer cells in a power diagram, a sort of weighted voronoi diagram. The power crust is intended to represent the surface of the modeling object. This definition guarantees a watertight surface and is similar to the volumetric approaches in that manner. The challenge is to ensure that the surface really approximates the real surface.

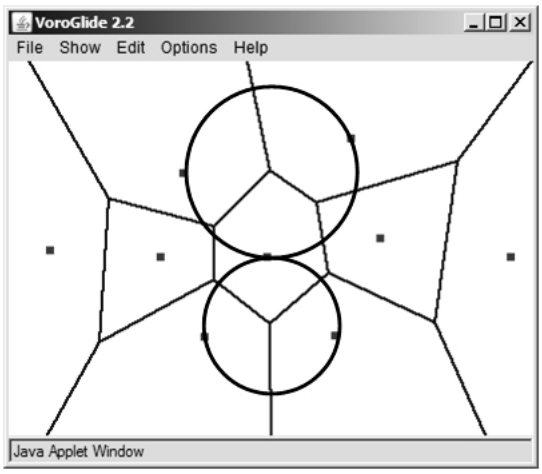

Figure 2.9: Voronoi diagram with two polar balls

The cells in the power diagram are convex polyhedra and are derived from the polar balls of the data set. A polar ball is centered at a voronoi vertex and have a radius such that it touches the closest sample points. Not all voronoi vertices are sources of polar balls. More precisely, only the two furthest voronoi vertices of a sample point are called poles, representing an inner and outer pole respectively. It is therefore the poles that determines the size and shape of the power diagram cells. The labeling of inner and outer power diagram cell determines the shape of the power crust, the model surface. With a dense and noiseless sample this method produces watertight surfaces that models the original object nicely.

However, scanned and merged data samples rarely fulfill this requirement. A simple, but powerful approach to obtaining a robust method is to discard poles that belong to badly shaped voronoi cells. The remaining poles is still sufficient to approximate the so called medium axis transformation, the set of maximal balls of a point set. A maximal ball can intuitively be described as the biggest empty ball for a given point sample. By implementing a labeling algorithm that propagates information by using a heuristic based on belief and confidence values, good approxmations of the object surface can me constructed even when previous the assumptions are not met. There is, of course, a limit to how much noise and undersampling this implementation can handle. A slightly different version of this algorithm is briefly described in [24] and claims to be even more robust.

In Tight Cocone the delaunay tetrahedra of the input sample is considered directly. The tedrahedra is marked as either inside or outside and the triangles that separate the inside tetrahedra from the outside tetrahedra are output as the final surface. If the sampled data is sufficiently dense the surface normals can be approximated by the voroni vertices called poles. Figure 2.9 shows a voronoi diagram with the two polar balls of the middle sample point. The algorithm is constructed to only produce a mesh for the areas where this assumption holds, temporarily leaving holes in the surface. To repair these holes another assumption is made: the undersampling is local. To extract a complete water tight surface, a careful process of marking the inside and outside tetrahedra and a subsequent peeling process of the outpolyhedra, are performed. When the last assumption is invalid the algorithm fails to produce a surface that is geometrically similar to that of the original object. It may even fail to output a surface all together.

When a surface mesh is produced by tight cocone it consists of triangles from the delaunay tetrahedra of the input sample, unlike the power crust which consists of random polygons that not necessarily interpolate the input data. Both these methods have the advantage of doing it all-in-one and thereby skipping several intermediate steps normally required in hole filling and surface reconstruction algorithms. However, calculating both the delaunay triangulation and the voronoi diagram are time consuming and usually considered too computationally expensive for large data models.

2.2.3 User Interaction

The level of involvement from the user during the hole filling process varies greatly between different methods. It is neither always straight forward to incorporate user input into the algorithms. Different kinds of users may have different information to contribute with and will need different ways of communicating with the system. Below follows some examples of how some of these problem have been addressed.

Some methods are fully automatic. They take the incomplete model as input and produce a filled surface as output. Most methods, however, require some user input to yield good results. Automatic versus manual methods is also a question of efficiency versus quality. Fully automatic methods are usually very fast, but normally produce a lot of artifacts for difficult data sets. Highly user controlled methods are more robust and can produce good models in most cases, but can be very tedious and time consuming to complete.

Some trade off between automatic methods and user assistance is often a good solution.

The simplest form of interaction is by parameter values. By adjusting different parameters used by an algorithm the output surface can be controlled to some degree. For instance, changing the weightening criteria in the triangulation method in [27] will have an impact on the way the hole is triangulated. Algorithms controlled by parameter values usually requires the user to have some fundamental understanding of how the algorithm works. As the number of adjustable parameters increase the method becomes more flexible, but also more complex. This normally limits the user group of this type of algorithms.

Popular objects for 3D scanning are historical buildings and artifacts. To reconstruct such models the insight of for instance an archeologist might be valuable. To exploit this expertise the surface reconstruction and hole filling algorithm needs to have an intuitive user interface that is easy to learn and use. Slide bars or radio buttons to adjust or change parameters might be an idea for the type of methods mentioned above, but unless the user understands the consequences of adjusting those values it is rather pointless. Making sure the user understand what is going on will be a challenge.

A popular idea for user assisted model repair is too allow the user to draw or sketch in the defective or missing regions. However, the surface model is three dimensional while the user is limited to drawing on a two dimensional projection with a normal desktop mouse. Ideally a virtual environment using a input device with six degrees of freedom sounds optimal for the task. These kinds of resources are not freely available, so simpler methods will have to suffice in most cases.

[6] is one such method. Once a defective area is identified the user inserts a template that roughly resembles the area that is to be reconstructed. Automatic methods are applied to deform and translate the template such that it coincides with the boundary of the defective region. The area outside the defective region is in no way altered by the algorithm. The user may edit the template by point, line or region transformations, both rigid and non-rigid. The changes in the template also need to be updated at an interactive framerate. When the user is satisfied with the initial surface approximation an automatic hierarchical surface completion algorithm finishes the job, recreating surface details at finer and finer levels until completion.

A similar approach can be found in [20]. There are, however, some notable differences. Instead of using a normal desktop mouse to draw on the model

the system utilizes a Phantom device. More information about this device can be found at the developers website [38]. It gives the user more degrees of freedom and a more intuitive *feel* of the object that is being modified. With this tool the user may draw lines to represent missing edges or connect islands to the surrounding mesh. There are many options available. The goal of the user interaction is to simplify the holes such that an automatic hole filling algorithm can faithfully close the remaining gaps.

Even with such a tool drawing on a three dimensional object on a normal screen still is tricky. [37] approaches the problem a bit differently. This algorithm is, however, not a hole filling algorithm, but a surface reconstruction algorithm that produces water tight surfaces, and thereby implicitly solves the hole filling problem. The algorithm aims to construct a continuous function that approximates the data samples. In addition to the data points there are also points known to be inside or outside the surface. These points may be specified by the user or obtained by some other means. The algorithm uses the finite element method (FEM) to solve the set of equations set up to specify the surface. For more information about FEM refer to [31]. These equations may not contain sufficient information to give a satisfying solution. The algorithm, therefore, searches the topology of the surface for unstable or weak regions. Once such a region is found the user is prompted to draw on a two dimensional cross section of the problem area to specify further inside/outside points. This process is repeated until no more weak regions are found and the system of equations can be solved yielding a topological correct solution.

Chapter 3

Criteria

Hole filling algorithms exhibits many different qualities and the results can vary greatly. Depending on the situation some properties might be more important than others. Even with all this variety there is still some common properties that are desired. Below follows an attempt to define a list of criteria a general solution to the hole filling problem should aim to meet. Each criterion is briefly explained.

A solution to hole filling problem should ...

C1 - be robust: There will always be some noise in data from scanned 3D-models. Small perturbations in the input data should have little or no effect on the outcome. Numerical and geometrical stability is crucial for algorithms like these. Achieving complete robustness is however very hard in most cases. It is therefore important to identify the types of problems the algorithm can and cannot handle, so that other methods can be employed when a specific algorithm fails.

C2 - produce visually pleasing and plausible models: A filled hole should blend smoothly into the original geometry. Ideally, you should not be able to tell which parts are from sampled data and which are constructed just by looking at the model. This requirement is often too strict but the newly added geometry should not be disturbing in any way. It is in many cases better to leave a hole unfilled than to add poorly constructed fill surfaces. Plausibility is also dependent on the observer. An expert on the objects being scanned will more easily spot

artifacts in the model. The intended user group should therefore be taken into consideration when evaluating the quality of the hole filling algorithm.

C3 - distinguish between original data, constructed data and modified data: It might be useful to know which part of the model that has been created as a result of the hole filling algorithm and which parts belong to of the original model. Some hole filling algorithm also resample the entire mesh or just areas surrounding the holes. Being able to tell which parts are modified might also prove useful. With this information confidence values can, for instance, be assigned to data points if the model is being processed further.

C4 - not alter samples in the original data set unless it is permitted: This criteria works under the assumption that the original data set is an fairly accurate description of the object being modeled and should therefore not be changed. However, this data may also be noisy or have lower accuracy is certain areas of the model. Some algorithms therefore argue that changing these data values will not degenerate the final model in any significant way. Ultimately, it should be the users choice to permit data to be altered.

C5 - be able to produce a surface that is hole free, 2-manifold, non-self-intersecting, closed and connected: The first part of this criteria seems pretty obvious. A hole filling algorithm should of course remove holes in the surface. The expression watertight is also often used in this context. However, there may be occasions where certain holes should be left unfilled. Some applications require surface models to be manifold. In a 2-manifold surface every edge is shared only by two polygons. Surface models that self intersect may cause problems for further processing and should therefore be avoided if possible. All real objects have a closed surface and a digital model should aim to reflect this, but in certain cases it may suffice to only model parts of the surface limited by some surface boundary. Finally, all parts of a object should be connected to each other. If not they should, strictly speaking, be designated as separate objects.

C6 - scale for larger input models: 3D models of objects are normally large data sets. Time and space complexity is an important issue.

Global methods usually suffer when the input models are large while local methods scale better. The choice of method will therefore be important for run times on different models. Care must also be taken during implementation to avoid too much memory usage and time consumption. A straight forward naive implementation is usually not satisfactory.

C7 - support user interaction and incorporate user knowledge: There are several tasks that still are easier to perform manually than automatically. User expertise may also prove difficult to incorporate into an automatic algorithm. To ensure the quality of the resulting mesh, allowing the user to interact with the program is probably the optimal solution. However, creating a meaningful and intuitive interface to facilitate user interaction is not trivial.

C8 - offer automatic hole filling: This is in conflict with the criteria above but is still a valid statement. Manually filling holes can be very tedious and should be made automatic to speed up the process. As long as the quality of the model does not suffer automatic methods is preferred.

C9 - consider all information available: For the model to be as realistic as possible any additional information about the model might be helpful. If such information is available it should ideally be incorporated into the model.

C10 - make sure added geometry matches the sampling density of the original model: Matching the sampling density helps with blending the filled hole with the rest of the surface. If the sampling is too sparse you may get visible artifacts. The patches filling the holes are usually estimated from surrounding geometry and is unlikely to contain any more detailed information than the original mesh. Sampling at a higher density in this area will therefore not contribute any extra information or increase the quality of the model, and should therefore be avoided.

Chapter 4

The Test Case

The test case for this paper is the Nidaros Cathedral in Trondheim, Norway. In earlier works, [25], the building was scanned and the data was processed to produce a stereoscopic video. This chapter will look at this model in correlation with the theory and researched mentioned earlier. A few simple methods will also be tested.

4.1 The Environment

This section will briefly describe the situation for this phase of the project. This includes available data, tools and hardware. Limitations caused by these circumstances will also be addressed.

4.1.1 The Model

The data available is a polygon model of the cathedral and point cloud data from separate scans. Since this project is about hole filling and not surface reconstruction it seems natural to work with the polygon model which already has been through considerable processing. This is by no means guaranteed to yield better results, but it helps to keep focus on the scope of this project. After employing a hole filling algorithm the results can be seen directly in the resulting model without the need for excessive post-processing.

The polygon model is stored in a .ply file. The PLY format is described in [16]. It is a simple format with a header followed by a list of vertices and faces. It also supports storing of additional information, but it is the vertices

and faces that are of importance to this project. It is available both as ASCII and binary formats. The PLY format offers compact storage and portability across platforms and software.

There are two versions of the model available. One with approximately 1.2 million vertices and 2.2 million faces, and a simpler model with approximately 0.6 million vertices and 1.0 million faces. The smallest model will be used for testing purposes to slightly ease the hardware requirements and reduce the time consumption of the different algorithms. Again, this might not be the optimal choice for the quality of the final model.

4.1.2 Tools

The focus of this project is the methodology of hole filling. There are numerous tools for editing 3D models, several of which were used in [25]. Most of these tools are not freely available and their algorithms can be hard to relate to the existing research. The choice is therefore not to use these tools. Given access to them, there is probably some steps of the modeling not directly related to hole filling that could be done more easily.

To view the model a simple open source 3D editor was chosen. It is called MeshLab and can be downloaded from [30]. It is a simple tool that also offers some editing operations. It actually offers a hole filling procedure too, but running this caused the program to crash. Seeing as there obviously still are some bugs in this program careful use is recommended. As a consequence the program was mostly used to visually investigate the model and take screenshots.

The obvious choice for programming with 3D models would be to use C++. It is highly flexible, fast and numerous open source libraries are available on the Internet. However, having no prior experience using this language, the choice was still made to use Java and MATLAB. MATLAB offers simple programming for mathematical operations and great opportunities for easily presenting data visually. Java offers more flexibility being an object oriented language. Open source code for read/write operations in MATLAB with .ply files are available at [29]. A counterpart in Java could not be found.

4.1.3 Hardware

The machine used to execute these programs was a Acer Aspire Laptop with a 1.7GHz Intel Pentium processor, 1GB RAM and a Intel Graphics

Media Accelerator 900 graphics card. This hardware is probably lower than most current system used for processing large amounts of graphical data. Algorithms will be slower to process, but overall this is not viewed as a problem. A faster or better computer may execute the programs faster, but when dealing with large amount of data the time and space complexity of the algorithms are far more important than the hardware itself. But it does pose some limitations to what can be accomplished.

4.2 Implementation

Implementation was done in multiple stages. First a suitable filereader had to be found and then a data structure had to be selected. Several preprocessing steps were also performed before hole filling could be done. This section presents the different stages of the implementation: model viewing, selecting a data structure, identifying the holes, mesh preprocessing and hole filling.

4.2.1 Viewing

Throughout the experimental stages it is necessary to look at the model to get an idea of how different choices change the model. It is also useful to indicate which steps to should be taken next. Several tools have been tested for displaying the model, but in the end only three were used. These were MeshLab, MATLAB and Java3D.

As mentioned in section 4.1.2, MeshLab is an open source tool that is fairly easy to use. It offers different options for viewing polygons and wire-frames. Different shading techniques can be used and lightning can be turned on and off. Turning and rotating the model is easy and intuitive. It offers an option for taking snapshots of the model and it is also the tool that suffers the least in performance due to the size of the model. Unfortunately, it does not support the version of ply that MATLAB produces.

MATLAB is mainly used for reading and writing to and from the ply format. But it also offers the possibility to display the triangle mesh and is therefore the chosen alternative in some situations. As with MeshLab it is easy to use and navigate and performs fairly well. More diversity is also supported through different MATLAB functions.

Java3D is by far the most versatile tool but the complexity is also a lot higher. It can be downloaded from [22], which also offers tutorials for learning

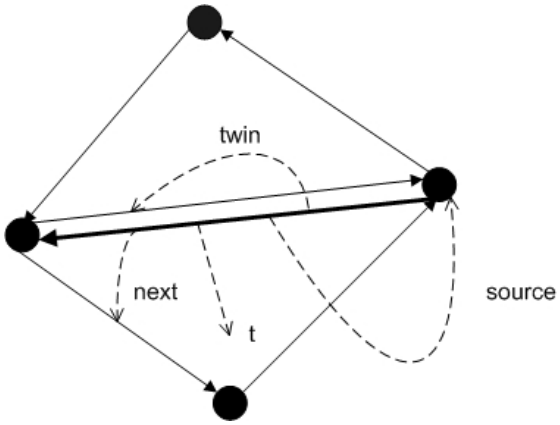

Figure 4.1: The half edge data structure

how to use Java3D. It was chosen in this project for mainly two reasons: the implementation is already done in Java and it offers the possibility to change the colors of individual edges. A key navigator class is already implemented in the Java3D package, along with several other utility classes, so supporting navigation is fairly easy. However, the performance suffers heavily because of the size of the data to the point where navigation becomes useless. A smarter implementation (such as using triangle and line strips) might alleviate the problem slightly, but hardware is also an issue. Because of this the simplest way to alter viewing parameters is to rerun the program with new camera coordinates and rotation angels for the model.

4.2.2 Data Structure

As mentioned earlier the test case is a polygon model, specifically a triangulated model. The model is represented as a list of vertices and faces in the .ply format. However, vertices and triangles alone are slow to work with. To simplify the processing of the polygon model a more informative data structure is advisable. The half edge data structure was chosen for this purpose.

The half edge data structure offers fast traversal of the polygon mesh through its edges and has a reasonable storage requirement. This is elaborated further in [43]. Because holes in triangle meshes normally are identified as loops of edges belonging to only one triangle, fast access to neighboring edges is important. There are several minor variations to this data structure. Figure 4.1 shows two triangles and their half edges. The highlighted edge is

displayed with its pointers. Using an object oriented approach with the Java language the following attributes were used:

- **Class Node** - Represents a vertex or point

  double x - the x coordinate

  double y - the y coordinate

  double z - the z coordinate

  int id - the identification number of the vertex, corresponding its position in the original vertex list

- **Class Triangle** - Represents a triangle in the mesh

  Node i - the first vertex in the triangle

  Node j - the second vertex in the triangle

  Node k - the third vertex in the triangle

  int id - the identification number of the triangle, corresponding its position in the original polygon list

- **Class Edge** - Represents a directed half edge in the mesh

  Node source - the node the edge is originating from

  Edge next - the next edge in the triangle

  Edge twin - the edge in the neighboring triangle, the other half edge

  Triangle t - the triangle the edge belongs to

  int id - the identification number of the edge

The most time consuming part of creating this data structure is identifying twin edges. Since the data set is quite large it is not very appealing to redo this process every time the program is executed. To avoid this the edge information is written to a text file very similar to the input file with the vertex and triangle information. After the first initial run where twin edges are allocated, the data structure is read from this file using the following format:

<number of vertices> <number of triangles>

<x> <y> <z>

...

<vertex1> <vertex2> <vertex3>

...

<source_node> <next_egde> <twin_egde> <triangle>

...

4.2.3 Identifying Holes

Given a triangle mesh, a hole is identified as a loop of boundary edges. Using the half edge data structure described above a boundary edge is a edge with its twin set to **null**. A boundary edge will always share a boundary with at least one more edge as single boundary edges cannot exist given correct assignments of values in the data structure. A boundary with only two edges is a degenerated hole and is a trivial problem to solve. A boundary edge can also only be a part on one boundary under the assumption of a 2-manifold mesh. Given these constraints locating holes is a straight forward operation and the process in outlined in the pseudocode below.

1. For all (half) edges e
2. if e is a boundary edge and have not been linked to a hole yet
3. create a new hole and link it to this edge
4. track all the edges sharing boundary with e and link them to the new hole

At the end of this procedure all holes have been found, but not all loops of boundary edges are holes. This leaves the process of separating hole boundaries from mesh boundaries. To ensure correct classification user input is needed. However, manually inspecting every single boundary loop can be tedious work as the number of loops increase. Therefore, an educated guess is made assuming all short loops are holes and all long loops are boundaries.

This leaves the middle range of the boundary lengths left for user classification. The length of a boundary loop is defined as the number of boundary edges it contains. The actual length of individual edges is not considered.

Even with this simplification the number of ambiguous loops can still be large. As a result the goal is shifted. Instead of trying to classify all loops, the aim is to identify a hole or holes that are ideal candidates for further testing. In a final application that aims to produce a complete hole free model this simplification cannot be made. That is, however, beyond the scope of this project, in which case the simplification is sufficient to reach the goals outlined for this paper.

4.2.4 Cleaning up

Most algorithms work under certain assumptions regarding the input data. Some fail to perform entirely if an assumption is not met, while others rely on the assumptions to ensure quality, with degrading quality as more assumptions are violated. A closer inspection of the test case reveals that a number of common assumptions does not hold for this particular data set. One approach to deal with this problem is to adjust the input data in such a manner that the assumptions hold.

The most common assumptions for triangle meshes are :

A - the mesh is connected

B - the mesh is 2-manifold

C - the mesh does not self intersect

D - the mesh does not contain degenerated or close to degenerated triangles

A - Connectivity

There are a few different connectivity issues with the test case. There are some small components that are not connected to the rest of the mesh. These can usually be regarded as noise and be removed from further consideration. With the current implementation only disconnected single triangles are automatically detected and subsequently marked as invalid. An invalid triangle will have its own identification number set to -1, together with all its edges'

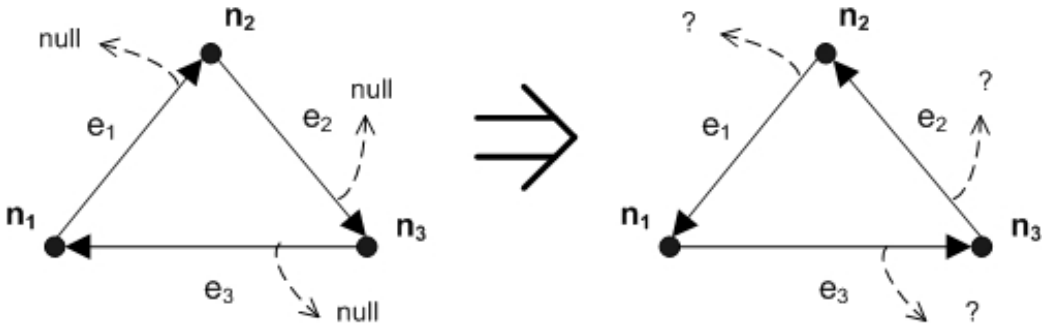

Figure 4.2: Reversing edge directions

identification numbers. With this scheme no information, such as connectivity, will be deleted that cannot be recovered, if that is desirable. A large scale connectivity test has not been implemented because it would be very time consuming and yield only minor improvements at this stage.

Also, the file reader assumes that all triangles are consistently oriented, i.e. the edges point from vertex 1 to vertex 2 to vertex 3 to vertex 1. For most triangles this assumption is met. There are, however, some exceptions and these triangles must have their orientation reversed. Single disconnected triangles will therefore have their edges reversed and then retested for twin relationships. If this test fails to find any twins, the triangle will be removed. Figure 4.2 illustrates the reversal process. The algorithm that locates twin edges searches for an edge that shares vertices with the current edge and is defined in the opposite direction. If a triangle is inconsistently oriented a twin edge will never be found, unless it shares the same flaw. Notice that this can lead to different orientations in disconnected parts of the mesh.

When larger parts of the model does not seem to be connected the problem is more complicated. Solutions may be to bridge components that are not connected. This is not a straight forward procedure. Simply adding an edge between the components does not yield a valid solution, since all edges need to be a part of a triangle. The half edge data structure relies on triangle information (next and twin properties) to traverse the data. And graphical models are normally polygon models, where edges are a part of a polygon but not a graphical primitive in itself. This problem has not been addressed in the current implementation, but is considered as a potential expansion to a hole filling tool.

B - Manifoldness

The half edge data structure only supports edges that are shared by a maximum of two triangles. i.e. a 2-manifold mesh. When a third (or more) edge share the same vertices as a pair of half edges the data structure fails to perform correctly. All edges and their triangles that violate the assumption of a 2-manifold mesh has to be removed. When three triangles share an edge it is sufficient to remove one triangle. Ideally the triangle that cause the greatest visual artifact should be removed, but without manually inspecting all such triangles it is difficult to know which triangle to remove. The solution is therefore to remove the triangle that is the most convenient to remove or the least connected triangle. Two different schemes where used.

First, the twin edge relationships are implemented such that when an edge has found its twin, the pair is removed from the list of possible twin edges. Subsequently, when a pair of twin edges are found they will not be tested against the remaining edges and any possible third edge that share the same vertices will not be discovered at this point. Afterwards, a test is performed to check whether any of the remaining boundary edges share vertices with any of the paired inner edges. When such an edge is found its triangle is removed.

At last, a test is performed to check whether three or more boundary edges share the same vertices and are defined in the same direction. If they share vertices, but are defined in opposite directions they would have been identified as twins in the pairing process. And the third edge would have been discovered in the first test.

All these processes are time consuming, with a run time in the magnitude of $O(n^2)$, with n being the number of edges. It may be faster to check for manifoldness during the search for twin edges, but this was not done due to two reasons. First, it is conceptually easier to do this step by step. And second, the main reason is the fact that pairing was already completed when the problem with non-2-manifold edges was discovered. It was therefore faster to perform the checks afterwards than rewriting and rerunning the entire process. As an example, pairing took approximately 48 hours to complete, while the first manifold-test took 12 hours and the second test took a few hours. These run times are fairly large and any optimizations should be attempted before any further use of this implementation.

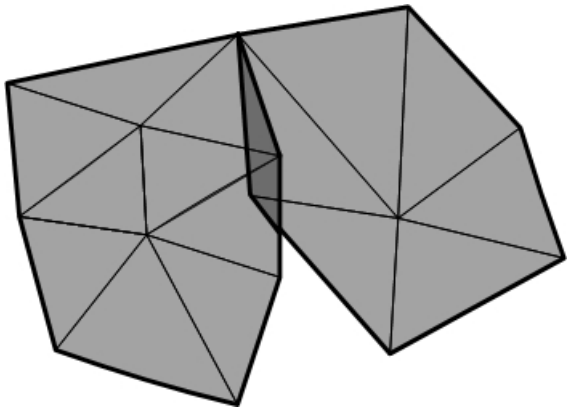

Figure 4.3: Intersecting components

C - Self-intersection

Self-intersection is mainly a visual artifact that does not inhibit algorithms like the ones used in this project to execute properly. It is therefore not taken into consideration in this implementation, but it is worth noting possible problems and solutions related to this issue. The mesh may have self-intersecting geometry resulting from merging different data sets or other operations. A good approach to dealing with self-intersections is to address the problem at its source, i.e. when it occurs. Maybe implement a scheme that avoids the intersections all together or allow them to intersect and then resolve the problem afterwards. Figure 4.3 shows two merged meshes that intersect. The surface reconstruction algorithm described in [40] deals with cases like this.

Another source for self-intersecting geometry are local hole filling procedures. Local methods usually only consider the hole and its vicinity when filling the hole. A part of the original geometry may be many edges away from the hole and therefore not considered a part of the vicinity, but still intersect the hole surface in the physical space. Figure 2.5 on page 13 illustrate this problem. To detect intersections such as these the entire mesh has to be considered, substantially increasing the computation time. To justify the additional workload added to completely avoid self-intersecting geometry it must either cause disturbing visual artifacts or prevent algorithms or applications from working properly. Neither is the case for the triangle mesh used in this paper.

Figure 4.4: A very skinny triangle

D - Degeneration

Not only does degenerated triangles consume unnecessary space, but it can also lead to errors. This is directly linked to the concept of geometric stability. Long and thin triangles are a common source for such problems. Figure 4.4 shows a very skinny triangle. A typical example would be to check whether the angle between two vectors (edges) are exactly 90 degrees. Theoretically, this is a simple calculation to check if the vectors' dot product equals zero. But due to finite precision arithmetic numerical round off errors will occur and most of the time yield a result that is close to zero, but not exactly zero. To address this problem it is common practice to check whether the angle is within some predefined threshold of zero. As a result, close to perpendicular vectors are interpreted as perpendicular. With numerous skinny triangles an edge might wrongfully be classified as perpendicular. It is therefore desirable to try to avoid such skinny triangles.

But even with a triangulation scheme that emphasizes equilateral triangles, skinny triangles may still occur due to the given point distribution. One solution is to resample the mesh or add more data points to difficult regions. This may also be undesirable in some situations. Typically if original data should not be altered or when model size is a real issue. For this test case remeshing or resampling would not be a good solution since the mesh is already a subsample of a much larger point distribution. Going back and altering the original subsampling and triangulation procedure would yield a much more accurate result.

Regardless, any implementations of geometric algorithms should aim to identify unstable steps. And subsequently try to compensate by finding alternate approaches or trying to discover and deal with all possible outcomes. One option may be to use infinite precision arithmetic. Further elaboration on geometric instability due to skinny triangles can be found in section 4.2.5.

4.2.5 Hole filling

After preprocessing the test case is ready for hole filling. Several methods were planned for testing, but due to time shortage there was only time to have a good look at one method. This method is described below.

Choosing an approach

The literature presented several different methods for filling holes. Some considerations must be taken when choosing which algorithms to test further.

Obviously, a method that is capable of filling the holes in the test case should be chosen. However, meeting the assumptions the algorithms are built upon can be tricky. In most cases some kind of preprocessing is required. The extent of this preprocessing must therefore be taken into consideration.

It is also generally a good idea to start with a method that is simple and easy to understand. If a simple method can solve the problem there is very little reason for choosing a complex method. This is of course not a very strong argument since the reality often is more complicated, but it is interesting to see what a simple algorithm can perform compared to a more involved one. The main reason, however, for starting with an easy approach is being able to understand what is going on. After working with the data and hopefully having discovered all its flaws and strengths, does it make sense to consider more complex methods.

With these two considerations in mind the choice fell upon exploring the method described in [28]. A few other properties also speaks in favor of this method. It works directly on a triangle mesh, which is the current format of the test case. With a triangle mesh the definition of a hole is clearly stated as suppose to point sample data where holes are loosely defined as undersampled regions. The method only considers the region surrounding the holes for calculating the hole filling surface. With large data sets local methods are often preferable to global methods. The method has also been previously tested and shown useful on architectural structures, such as the one in the test case.

The Concept

As briefly mentioned in section 2.2.2 this algorithm splits a complex 3D hole into smaller 2D holes and then recursively fills the 2D holes until the entire

hole is filled. Given a loop of boundary L edges the following pseudo code illustrates the process:

1. remove two subsequent edges e_1, e_2 from L and add them to the list H
2. compute the plane p defined by $e1, e2$ and the vertex v that they share
3. pick the next e_n and the previous edge e_p from the L
4. do
5. if e_n lies in the plane p move e_n to H from L and set $e_n = e_n.next$
6. if e_p lies in the plane p move e_p to H from L and set $e_p = e_p.previous$
7. until L is empty or (e_n and e_p no longer lies in the plane p)
8. if L is not empty add an edge e to both (one of each half edge) L and H that closes both loops
9. fill the hole in H with a 2D hole filling algorithm
10. if L is empty terminate the algorithm
11. move to 1. and repeat the process with the updated L

The lists L and H should be maintained such that given an edge, the next edge on the boundary is also the next edge in the list. When the end of the list is reached the next element should be the first element in the list. The same idea goes for previous edges. The basic constructs of the algorithm is also displayed in figure 2.7 on page 17. The dotted line represents the edge added in step 8.

The key to this algorithm is to recognize that all connected edges which lie in the same plane can be viewed as the much simpler problem of filling a two dimensional hole. By allocating all the edges in the original 3D hole to smaller 2D holes the 3D hole is easily filled. Any simple algorithm for filling 2D holes will suffice. The protruding (or ear clipping) algorithm described in [43] was used in this case.

In [28] this process is followed by a refinement and smoothing step. For large holes it is sensible to add new data points along the new edges and remesh the hole. These new points does not add any extra information to

the model but it alleviates the problem of elongated and skinny triangles. As described in section 4.2.4 such triangles may cause problems. Smoothing may increase the visual quality but it may also smooth sharp edges. Smoothing should therefore be applied with care. The main reason for not implementing mesh refinement and smoothing in this case is the lack of a proper test hole. Holes in the model that could be filled with this algorithm were usually very small, either with respect to total absolute edge length or total number of edges.

Pitfalls

When a hole is filled it does not check whether any existing geometry intersect the new hole surface. Self-intersections may therefore occur. As mentioned in 4.2.4 it should not be considered a problem unless it decreases the visual quality of the model. If self-intersection must be avoided choosing a global hole filling algorithm instead is a lot safer option. User interaction is also a way to resolve self-intersection problems.

As with any recursive algorithm there is a risk of running out of memory, i.e. getting a stack overflow. Increasing the stack size is one solution, but a safer option is to change the implementation to an iterative approach. If, however, the maximum problem size is known and the stack size can be increased to a reasonable size and accommodate for the recursive implementation, that should be sufficient for testing purposes.

The algorithm has another weakness. As indicated in 4.2.4 geometric stability is an issue when checking whether two vectors are perpendicular. Checking if an edge lies in the plane is equal to checking the angle between the edge and the plane normal. Since all edges processed so far (the list H) already are contained in the plane and the edge must be directly connected to one of these edges, the edge has at least one vertex in the plane already. If the vector pointing to or from this vertex is perpendicular to the plane normal the edge lies within the plane.

The problem occurs when choosing a threshold value to interpret close-to-perpendicular edges. Simply setting the threshold to zero does not solve the problem either due to finite precision arithmetic. The problem is illustrated in figure 4.5. Given four edges in a loop three of them, e_1, e_2, e_3, may be identified as lying in the same plane while the forth, e_4, is not, even though both its vertices already are contained within the plane. The algorithm will therefore bridge the two loops , $H = e_1, e_2, e_3$ and $L = e_4$ respectively, by

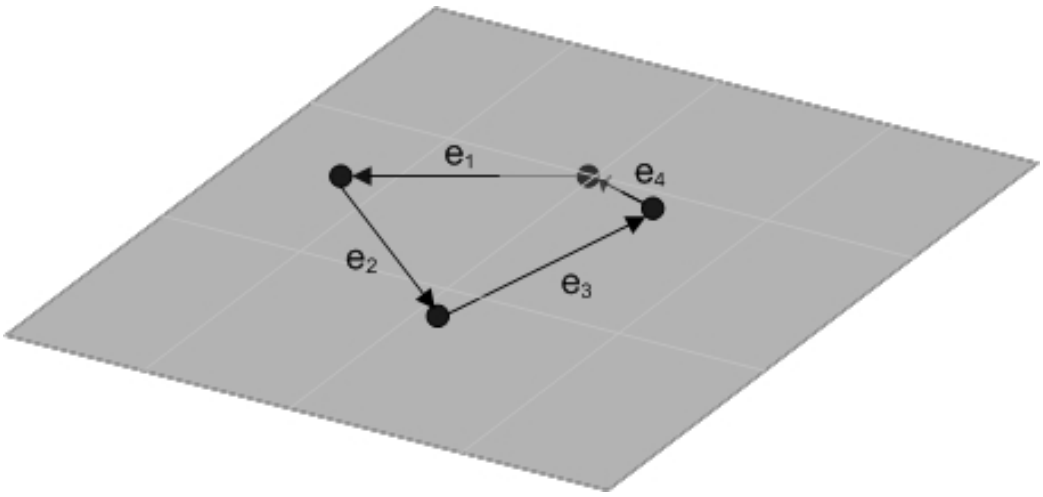

Figure 4.5: Edges in a plane

adding an edge between the endpoints. But since there already exists an edge between these vertices, problems arise.

The main reason for errors with this kind of misclassification is the fact that the edges which are added to bridge loops are not associated with any triangles to begin with. As stated in 4.2.4 an edge without a triangle cannot exists. But since it is unknown at its time of creation which triangle the edge belongs to, it has to be allowed to exist temporarily. With misclassification and wrongful subdivision of the complex 3D hole these bridge edges may never be associated with any triangles. Trying to traverse such an edge will ultimately cause the algorithm to fail. The easiest way to compensate for this problem is to adjust the error threshold until a safe value is found, however it is not an ideal solution.

4.3 Results

The testing revealed several obstacles that was not taken into consideration before the implementation phase started. The result were more time spent one a single method as suppose to testing multiple methods, which would have been desirable. Poor planning can partially be blamed for this, but lack of prior experience in this field of research also makes it harder to estimate time usage. Lack of proper hardware for development also turned out to be

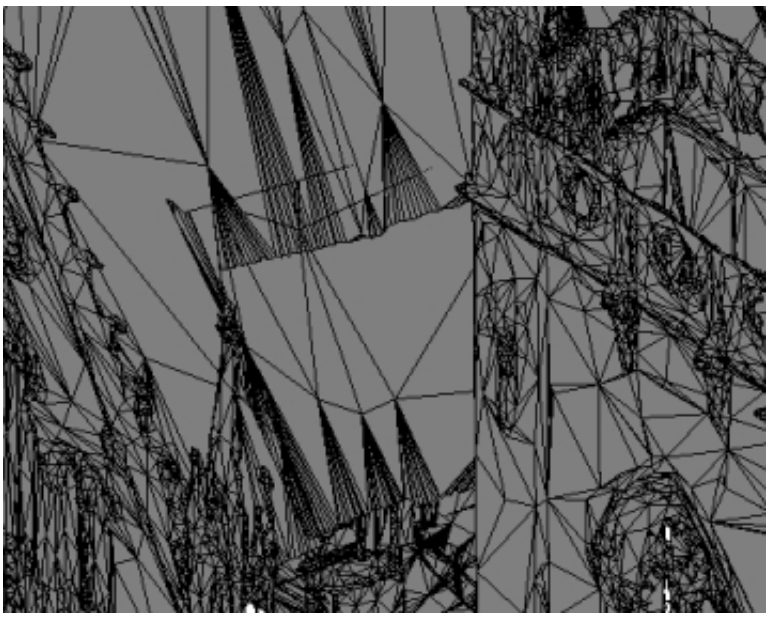

Figure 4.6: Skinny triangles

an issue. Even with these constraints there were some valuable lessons to gain from the experiments.

4.3.1 Examining the Model

Just by careful visual inspection some issues present themselves. The triangulation is far from the ideal close-to-equilateral result a delaunay triangulation can produce. [25] claims to use delaunay triangulation to produce the surface so the assumption must be that this is the case. The bad meshing is therefore probably a result of downsampling the point clouds to get a manageable data set. Knowing the problem exists extra care must be taken to ensure geometric stability. An example of poor meshing due to sampling distribution can be seen in figure 4.6.

Several smaller holes can also be observed, but the real issue is the larger holes. Some parts of the model completely lack data. The most problematic area can be seen in the lower right section of figure 4.7. It is highly doubtful that any of the automatic hole filling methods mentioned in the researched literature can deal with this problem. Some other holes seem more feasible to fill with semi-automatic methods due to presumably simple geometry, such

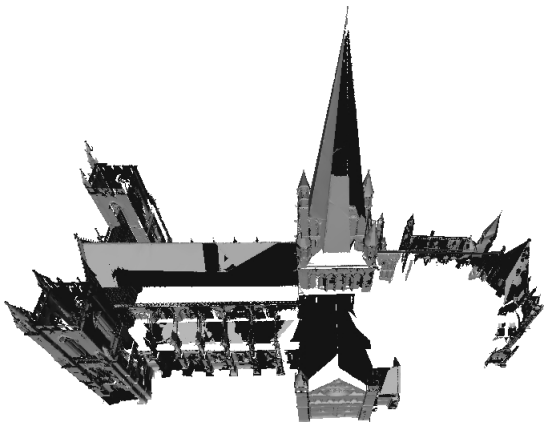

Figure 4.7: Bird view of the test case

as roofs. The initial idea then becomes to examine whether that is the case.

4.3.2 Testing

The test case consists of 638271 datapoints and 999669 triangles. Before any editing was performed tests reveal that there is 4092 separate boundary loops. Maximum boundary length is 13342 edges long while the minimum length is 3. The mean loop length is 70,68 edges. This suggests a heavy representation of small boundary loops. Figure 4.8 confirms this. It shows a bar plot of the lengths of the individual loops. The majority of boundary loops is less than 2000 edges long.

After removing all triangles that violated the 2-manifold property the number of loops were reduced to 3883. After deleting all single disconnected triangles this number is reduced even further to a total of 3866 loops. A choice was made to fill all holes with a boundary length less than 200 with the hole filling algorithm described in section 4.2.5. This effectively reduces the number of boundary loops to 168. The smallest boundary loop is now 200 and the longest is still 13342, but the mean length has been increased to 1119. The boundary loop lengths after hole filling is plotted in figure 4.9. The majority of boundary loops still contain less than 2000 edges.

After hole filling the model contains 1093097 triangles. The number of data points remain unchanged. Figure 4.10 shows a wireframe view of the original model. All boundary edges are marked in red. Figure 4.11 displays the model after hole filling. All edges added by the hole filling algorithm is

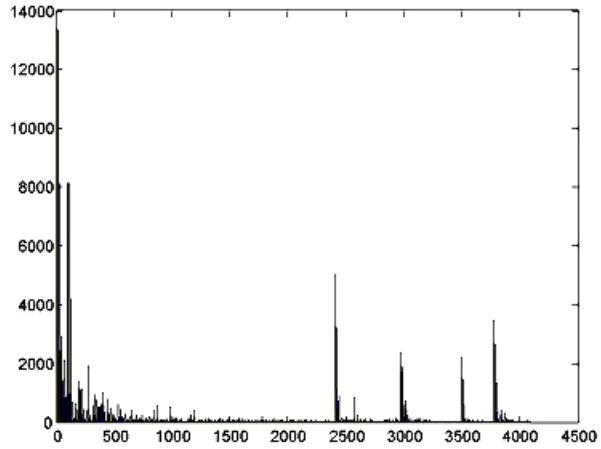

Figure 4.8: Number of edges per boundary loop

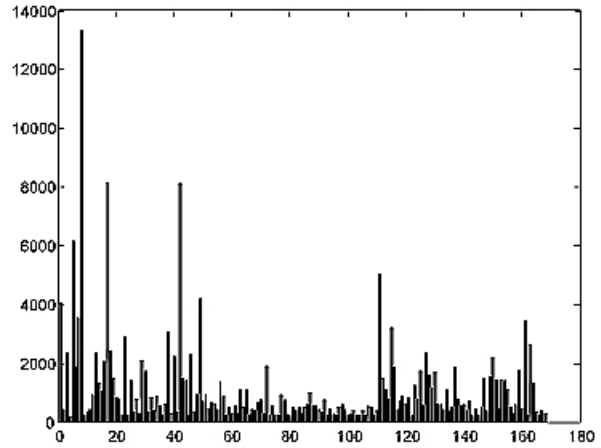

Figure 4.9: Number of edges per boundary loop after hole filling

Figure 4.10: Original model with boundary edges

Figure 4.11: Filled model with boundary edges and new edges

displayed in blue. Remaining boundary edges are still shown in red. Notice the roof on the right side of the model. A boundary segment has been wrongfully classified as a hole and has been filled like any other boundary loop. This happened because the boundary loop was less than 200 edges long. Assuming all loops with less than 200 edges enclosed holes was therefore not correct. Lowering this requirement to, for instance 100 edges, will remove these artifacts, but will also leave more holes that need to be filled with an alternate approach. Changing the value does not address the real issue of separating boundary and hole loops.

Finding a hole with a single connected boundary, which would have been suited to properly test the algorithm described in 4.2.5 failed. The holes in the rooftops that seemed good candidates for testing turned out to be adjacent to multiple disconnected mesh pieces. To fill these holes it is not sufficient to define a hole as a loop of connected boundary edges. Other approaches need to be taken.

The volumetric algorithm described in [23] does not rely on the simple definition of a hole used so far. It can deal with arbitrary hole boundaries and aims to produce a closed, watertight surface model. The executable implementation of this algorithm, called Polymender, was downloaded from [33] and tested. For surface extraction there is a choice between Marching Cubes and Dual Contouring. Dual Contouring require more memory but can produce more accurate surfaces, and where therefore the chosen method to test. Several octree depths were tested, but setting the depth to 9 resulted in a model with approximately the same number of primitives as the current model. Increasing the size further caused the model size to grow unreasonably large.

Figure 4.12 shows a close up of an area with many holes that could not be filled with the algorithm described in 4.2.5. Figure 4.13 shows the same area filled with Polymender. Some artifacts can be observed. Especially where the mesh changes geometry and merges with the rest of the model. The filling is smooth and surprisingly good despite some obvious problems. Polymender resamples the entire mesh in order to fill holes. Figure 4.14 and 4.15 shows an original piece of a wall and its resampled version, respectively. As shown, the algorithm is not capable of conserving the original data and the quality is degraded. Polymender also fails to fill the larger missing parts of the model, as expected.

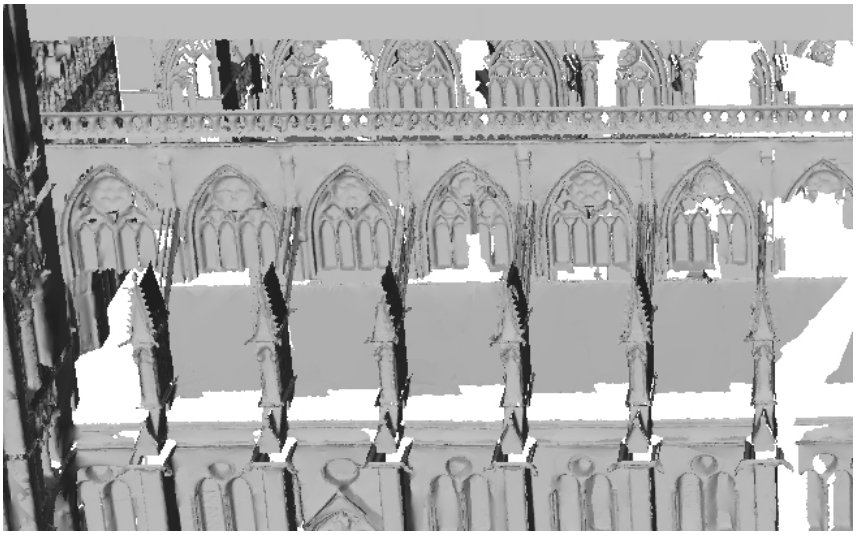

Figure 4.12: Original model, side view

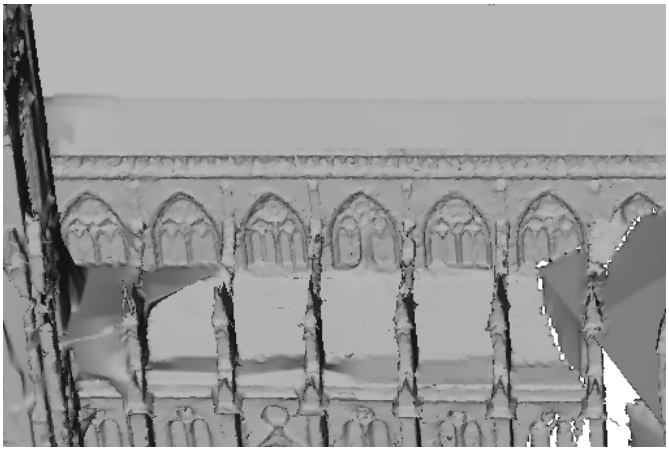

Figure 4.13: Holes filled with Polymender

Figure 4.14: Original model, front view

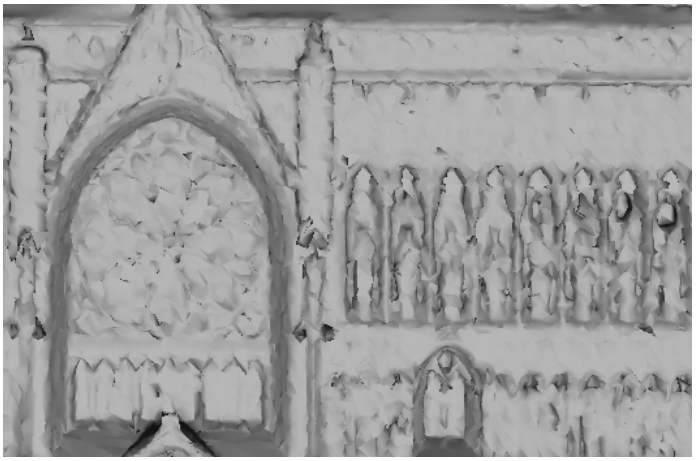

Figure 4.15: Resampled with Polymender

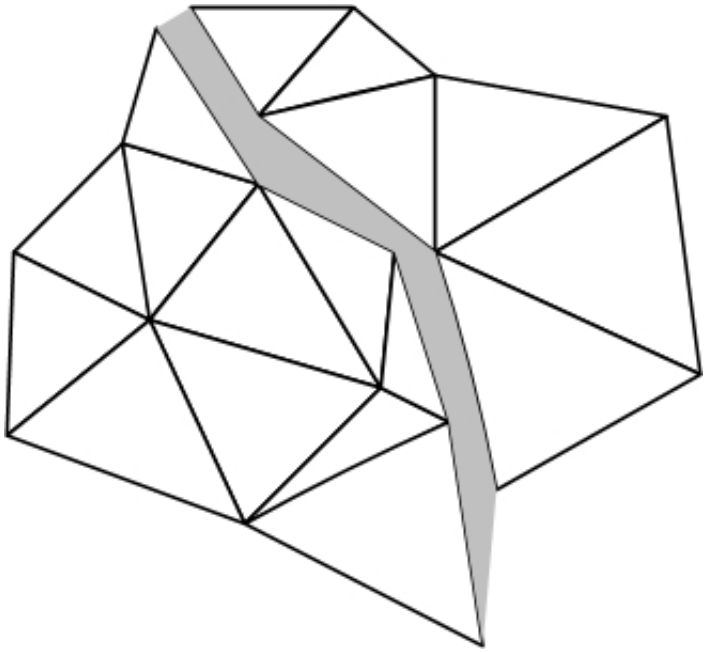

Figure 4.16: Two disconnected components

4.3.3 Problems

Two major issues where discovered during the testing phase. The first being lack of available hardware to support graphical interactivity. And the second was the failure of identifying large holes that coincided with the definition of a hole that were used.

User interaction or guidance is desired to increase the quality of the model, but there is an issue with incorporating this information in a user friendly and intuitive way. Even though the current implementation is not optimized for performance it is unrealistic to expect response times greater than what, for instance, MeshLab can produce. MeshLab seemed to operate with a framerate in the area of 0.3 to 0.8 frames per second when responding to view changes of the model. Optimizing the code to achieve this performance is not really justified time usage for this project. Less than one frame per second is not an ideal situation to work with either. Performance needs to be improved to facilitate user interaction. This concept is elaborated further in section 5.1.2.

Even with the drastic decrease in the total number of holes after elim-

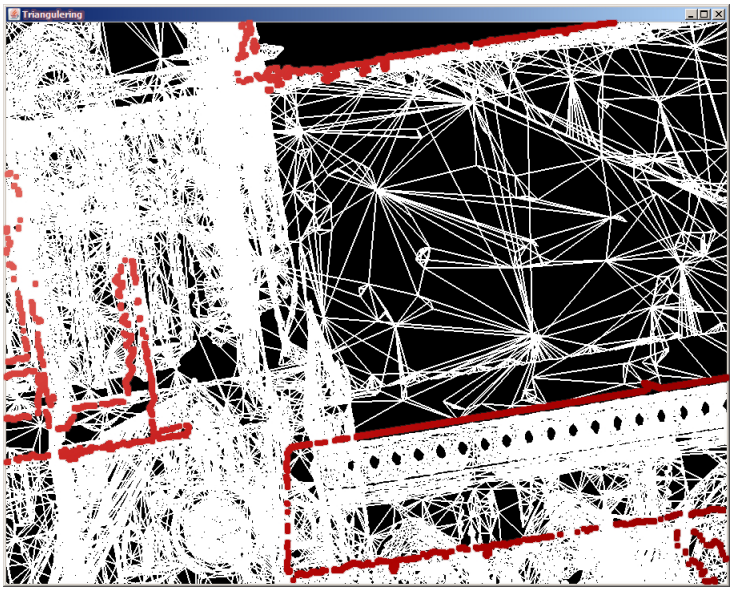

Figure 4.17: A long boundary loop

inating small holes, there is still a fairly large amount of holes left. Every single hole was not checked, but for all those that were examined the hole boundaries did not appear where they were anticipated. The problem that was encountered is illustrated in figure 4.16. The gray region represents the area that should be filled. Instead of finding boundaries enclosing holes, the boundaries enclosed separate and apparently disconnected components of the mesh. An example is showed in figure 4.17. One boundary loop is draw in red and it encloses the roof in addition to continuing in twists and turns into the center tower. The holes in the roof that was so apparent figure 4.7, do in fact not have a connected boundary. This greatly complicates the situation and discards the definition of a hole used so far. Suggestions on how to overcome problems like this is discussed in chapter 5.

4.3.4 Solution

Due to the problems stated above and the limited time available to complete the project the actually testing phase turned out to be shorter than originally planned. Making mistakes, wrongful assumption and decisions, makes a development process slow, but resolving these flaws also offers extremely

valuable insight on the problem. By studying the results that the testing did reveal and reexamining the literature on hole filling with this new information in mind, a new scheme for hole filling can be outlined. The implementation of this scheme is left for future work, but it incorporates the lessoned learned in this project. This scheme is presented in chapter 5.

Chapter 5

A Possible Solution

In chapter 3 a set of criteria for a solution to the hole filling problem where defined. With the knowledge and experience from the previous phases the question of how to try to meet these criteria can be addressed. This chapter is attempting to do just that. This is done by presenting an outline for a hole filling tool. This presentation is followed by an evaluation of how this tool aims to fulfill these criteria.

5.1 A Hole Filling Tool

This paper is revolved around the test case of the Nidaros Cathedral. But it is also a paper about the general problem of filling holes in scanned 3D models. This tool will therefore not present a specific solution for the test case, but try to address the general hole filling problem. Being a general solution it should also be capable of resolving the specific case of the Nidaros Cathedral.

5.1.1 Framework

With all the available methods for hole filling out there, the question becomes what can a hole filling tool contribute with. To justify making a complete tool instead of just offering a library of implemented algorithms the tool needs to offer something additional. The goal is allowing the user to simplify the problem areas such that automatic methods become sufficient to complete the task. The graphical user interface is therefore the key. It needs to be

intuitive and easy to use but still offer flexibility.

One of the lessons from the testing phase was the need for more user interaction. This gave rise to the idea of a graphical software tool to ease the hole filling process. This could be a stand-alone application or an addition to an existing tool for processing large amounts of three dimensional graphical data. Since hole filling can be an integrated process in the model reconstruction from scanned data, it seems reasonable to exploit existing software. A lot of base functionality will already be implemented, such as reading files, displaying the model with different viewing parameters and basic editing operations. Finding suitable software will probably require some research and testing. The challenge is to find a working application that supports customization by independent programmers and offers sufficient functionality.

Another issue is knowing when to perform the hole filling. Going from raw scanning data to a complete model is typically a stepwise procedure with many different paths to reach the goal. Hole filling can be performed before or after surface creation, noise removal, downsampling and several other cosmetic changes. The optimal time to perform hole filling may very well be case dependant. An application that can use various levels of data is therefore desired.

When developing a graphical user interface it is important to know who the end users are. The general idea with this tool is to allow others than just computer scientists to use it. People with backgrounds in for instance archeology or architecture, which may contribute with their knowledge to the model completion process, is a suiting user group to aim for. The assumption is then that the user has no prior knowledge on the subject of hole filling. The user should not need to understand the mechanics of a function to be able to use it. However, the user need to understand the impact of changing parameter values for different functions. This is where the challenge is. Developing a user interface that offers diversity in functionality and user friendliness.

5.1.2 Functionality

A large amount of functionality can be added to a tool such as this. Listing everything would be time consuming and outside the scope of this presentation. The focus is on main functions that contributes directly to an easier and better hole filling process. This is not intended to be an absolute requirement specification either, but rather a guideline to help develop one.

Most of the functionality is already presented in the literature. The power lies in bringing all these ideas into one common tool.

Viewing

It is obvious that a graphical editing tool should be able to display the model. However, it is still worth a few comments. To get a good view of problematic areas it is important to be able to view the model from an arbitrary position. One example is MeshLab, that allow for zooming in and out and rotation of the model. But it does not allow for vertical or horizontal displacement of the camera. This is effectively limiting the possibility to look closer at areas away from the center of the model. Arbitrary camera positioning is therefore important.

Altering viewing conditions such as graphical primitives, shading, lightning and textures will have a large impact on the impression of the model. Lightning adds a depth feeling to the model and combined with shading or texturing it gives a clear indication of the final look. While editing it may however be easier to look at a wireframe model, with or without polygons. In some cases even a point cloud might be the desired representation. The user should therefore be able to easily change viewing conditions while working on the model.

Simple hole filling

With all the focus on filling complex holes it is easy to forget about the simple holes. They still need to be filled and different options are available. The algorithms in, for instance, [27] and [8] serves this purpose by handling both holes in surfaces and point clouds respectively. Other algorithms can be used.

However, the definition of a *simple* hole is of relevance. It is not always clear where the line between a simple and a complex hole should be drawn. A hole can have a wide variety of defects. This is important because the purpose of using this tool is to simplify difficult and complex holes such that simple, robust and automatic hole filling algorithms can complete the process. The most common definition of a simple hole is a disk shaped hole. A relatively flat and often convex hole with a boundary pointing towards the center of the hole. In some cases it is enough that the projection of the hole is has this shape. Other small variations exists too. It is therefore important

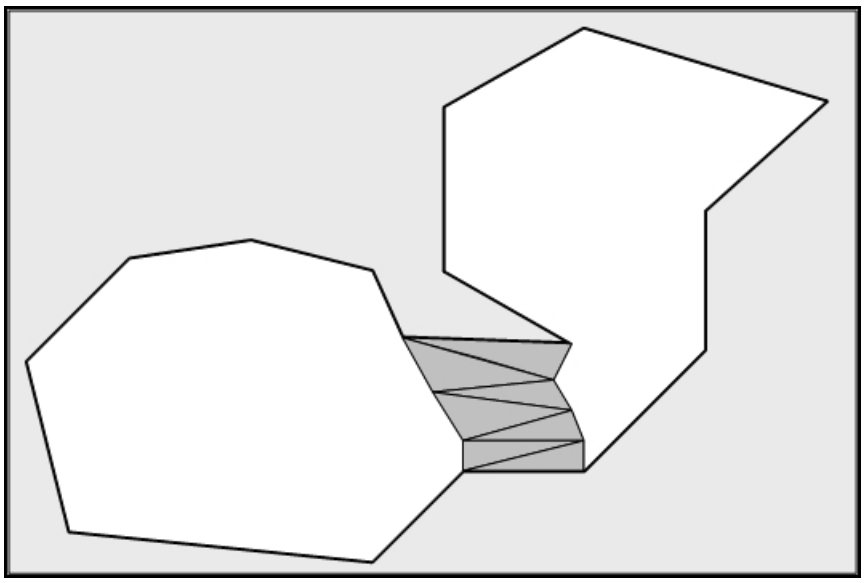

Figure 5.1: A bridged hole

for the user to know how much a hole needs to be simplified before it can be filled with a simple algorithm.

Bridging

Bridging, in this context, refers to connecting two boundary edges by adding geometry to an empty region between the edges. The only assumption is that the edges are not directly connected to each other. They may, however, be a part of the same boundary loop. Geometry is usually a series of triangles, but can also in some cases be limited to point samples only. Single edges are normally not allowed. Figure 5.1 shows a large complex hole that has been bridged. The idea of interactive bridging is taken from [20], which uses a Phantom device to ease interaction. Manipulating 3D virtual objects with a such a device is a field of research in its own, and might be worth some attention. A normal desktop mouse with two buttons and a scrolling wheel is assumed for now.

This is quite a versatile tool. It can be used to connect disconnected components, such as small islands and large separated parts of the model. It can also be used to split existing holes into smaller holes. Typical situations would be to create a bridge where the geometry of the hole changes

drastically. Thereby splitting the hole in two where each part have different boundary conditions. A bridge should not be limited to the shortest path between two edges, even though that will probably be the most common scenario. A bridge will have both its endpoints restricted to coincide with the *boundary* of the current model. This should inhibit the user from creating edges that are 3-manifold (or higher). The phrase *current model* refers to the original model with the editing operations done so far. If a bridge does not have both its endpoints connected to the model, it does no longer fall under the term bridge. The restriction to connect both endpoints is not a restriction on the functionality the tool should offer, but a restriction on the what the term bridge is covering.

Adding geometry

Holes may cover regions of the original object that contain complex and unique geometry. Simply bridging and subsequently filling simple holes may not be sufficient to give a plausible solution within a reasonable amount of time. The idea is that the user should be able to insert geometry into the missing pieces similar to the concept of Constructive Solid Geometry (CSG). For more information on CSG refer to [18].

Consider a scenario where the user knows that a wall is missing. A wall can in most cases be represented by a polygonized square. The user may also know that there is a window on that wall, at a certain position and with certain dimensions. Figure 5.2 illustrate an example. Using relatively simple geometric primitives more complex shapes can be constructed. If a mapping from virtual space units to physical world units exists, it should be utilized. This would allow the user the measure, for instance, the size of the wall in the physical world.

Added geometry should be under the same restriction as bridges regarding boundary edges, i.e. new geometry can only be attached to boundary edges. The user should be able to roughly position the new geometry and then use an automatic procedure to make the precise attachment. This is similar to template insertion as described in [6] and the same algorithms can probably be used successfully. The user should also be given a choice to manually bridge the new geometry to the existing model. This can be rather tedious work, so in most cases the automatic attachment will be preferred.

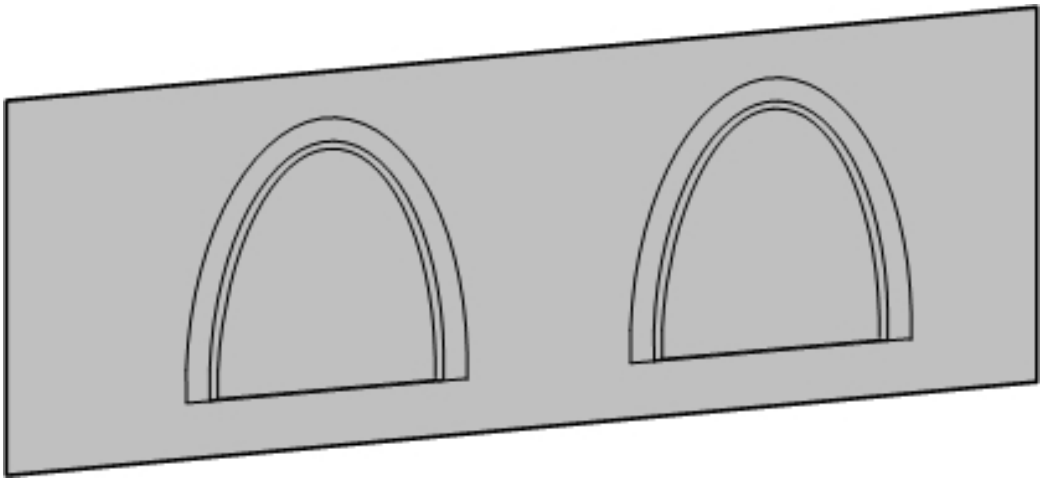

Figure 5.2: Simple constructed geometry

Sculpturing

In some cases newly added geometry or even defective parts of the original model require some editing. This will typically be click and drag operations. The idea is to deform the model until it gains the desired shape. An editing operation of this class should not alter any connectivity information or cause gaps in the model. It is a pure deformation operation and does not necessarily follow any physical rules, such as preservation of volume.

However, to get a smooth and intuitive deformation it is normally a good idea to let a change in the selected primitives also spread to their nearest neighbors. The effect should be limited too a small area and decrease with the distance to the changed area. This topic strays away from the topic of hole filling and requires some more research to be able to handled properly. Further specifications of this functionality is left until more information on the topic has been processed.

Inpainting

Inpainting refers to the methods mentioned in section 2.2.2. More specifically the method described in [5] and [4] seems promising. It belongs to a group of methods that fill holes by finding a similar region somewhere else on the model and copies that information into the hole. Automatic detection of similar regions is already described and can lead to good results. It can,

however, also lead to weird filling surfaces that does not seem meaningful for a human eye. By letting the user indicate where a similar region should be found, the output of the algorithm becomes more predictable and stable.

The strength of the algorithm is being able to copy data at different levels. The coarse shape may be copied from one part of the model while the finer structure may come from a completely different part. For large models it may be too extensive to search the entire model at several different sampling densities to find a good match. This also speaks in favor of allowing user interaction to narrow down the search.

Even though this method looks really promising on paper, it should be verified by testing it on a few difficult data sets. Unfortunately, there was not enough time to do that in this project, and is therefore left for future work.

Filtering

Much like in image processing applying filters can improve the result. The most common filters used are smoothing and sharpening filters. The same applies for 3D model processing. Newly added geometry may be faceted and require smoothing to blend properly with the existing geometry. Newly added geometry may also be too smooth in certain areas. Applying a sharpness filter to enhance sharp edges is also useful. Implementing additional filters is also possible.

Even though filtering of three dimensional point cloud data is conceptually similar to two dimensional filtering, the implementation is quite different. Parameterization is trickier in 3D and filtering requires more complex algorithms. [19] describes a method for applying a sharpness filter. Smoothing is mentioned frequently, for instance in [27].

Region of influence

When editing a model it is important to have control over which parts of the model that are altered and which are not. An editing operation is typically local and should therefore only have consequences for the neighborhood of the edited sections. In some cases it might be desirable to tag data points as read-only and not allow any editing operations change them. Since the neighborhood can still change, connectivity information may still require updates.

When altering only a selection of the data the boundary of the selection needs to plausibly blend with the unaltered data. This may be tricky to deal with depending on the editing operation which was performed. It is probably a good idea to investigate the extent of this problem before deciding on any course of action.

Comparable data

When the user is working to improve the model it would be beneficial to have some visual data for confirmation. This can be images, videos, drawings or blueprints. Any form of visual data might be helpful. The intention is to compare the model with the additional data using the visual perception of humans. No automatic comparison is intended. Human perception and interpretation still outperforms any automatic algorithm that can be easily programmed.

In some cases this type of data may not exists and the user is working blindly to construct a plausible hole filling surface. Data from similar objects may still prove useful. A database for visual data could therefore be incorporated. This functionality is not very different from having paper copies of pictures or having digital images opened with a secondary application, but it is expected to ease the modeling process. It is however reasonable to give this feature a low priority.

Performance

The editing process is intended to be interactive. Response times must therefore be reasonably fast. A guess based on experience is that framerates between 15fps and 60fps is comfortable to work with. For large models there might be some complications while rendering. It is outside the scope of this project to investigate fast rendering methods for large models, but in an implementation this cannot be ignored.

Most editing operations are expected to be local. Producing fast results for small areas should be feasible. Also, certain algorithms should also be allowed to have longer response times. It is difficult to set a distinction between the nature of fast and slow algorithms. A few examples will illustrate the point. If the user want to, for instance smooth the entire model, appropriate parameters should be selected and the procedure will execute. Waiting a few seconds for the result seems reasonable. Now consider a click and drag

operation. The user selects an area, drags it, wait for the model to update and then deselects or releases the area. Waiting for updates while dragging the model or parts of it is not an ideal situation. The user needs to get a feel of the operation being performed and an interactive framerate is important to achieve that.

5.2 Evaluation

Below follows an attempt to evaluate how the hole filling tool described above can meet the criteria defined in chapter 3. Depending on the model that is being reconstructed these criteria may be weighed differently. The criteria is repeated below.

5.2.1 Restatement of Criteria

A solution to the hole filling problem should:

C1 - be robust

C2 - produce visually pleasing and plausible models

C3 - distinguish between original data, constructed data and modified data

C4 - not alter samples in the original data set unless it is permitted

C5 - be able to produce a surface that is hole free, 2-manifold, non-self-intersecting, closed and connected

C6 - scale for larger input models

C7 - support user interaction and incorporate user knowledge

C8 - offer automatic hole filling

C9 - consider all information available

C10 - make sure added geometry matches the sampling density of the original model

5.2.2 Analysis

Each criteria is briefly discussed in relation to the proposed hole filling tool.

C1 As the complexity and flexibility of a program increases it becomes harder to maintain robustness. Some methods work under certain assumptions and cannot solve the problem unless these assumptions are met. It is therefore, the users responsibility to make sure that the methods that are used actually are capable of solving the problem. Regardless, users may always make mistakes or attempt to use solutions that in theory does not work. When that occurs, the program still need to behave in a predictable and robust manner. The program should not crash and should give an informative error message if he procedure cannot run. In some cases the procedure may still execute but yield results that are clearly wrong. The user should in that case be able to see that the results did not turn out as expected and then return to the previous state. For the hole filling tool to be robust the programmers need to make these considerations.

C2 Given proper use of the tool, it is very capable of producing visually pleasing and plausible models. In fact, that is one of the strongest arguments for a user interactive tool. The user has complete control and any desired level of plausibility can be achieved.

C3 This criteria is also dependant on implementation. It is however a lot easier to fulfill. The original data should be tagged as *original*. When original data is being altered the tag should be changed to *modified*. When new data is created it should be tagged as *constructed*.

C4 This criteria is a bit complicated. It is very case specific and depends completely on what the user wishes to do with the model. If the user wants to resample the entire model that should be possible. But if the user wants to only edit a small portion of the model, then other parts should not be changed. The program should allow the user to select a region of influence for different editing operations and define a limit on how much a change can propagate to neighboring samples. The goal is to let the user have complete control on which samples are being changed. The description of the tool supports such a solution.

C5 With a wide variety of hole filling methods to choose from this requirement can be met. Ensuring all these attributes often come at a cost and the user need to evaluate whether it is worth the added cost. Accuracy is often sacrificed when all the properties are needed. Lowering the requirement to only a few properties will often lead to better visual results at the cost of operational and mathematical guarantees. Depending on which qualities the user emphasizes, appropriate methods should be available to choose from. The tool is intended to support multiple hole filling methods and conceptually supports this criteria.

C6 Different algorithms scale differently as the sample size increases. The tool will therefore also scale differently for different functionality. However, to allow for really large models some kind of subdivision scheme is most likely needed. By splitting the model into smaller components the data set becomes more manageable, but also complicates the use of global methods. Global methods tend to scale badly with increasing sample size and will in most cases not be the preferred choice when the model is very large. Including this functionality into the tool should be feasible.

C7 This criteria is one of the fundaments of the tool. Its design is based on user interaction and relies on user knowledge to perform well. User knowledge can be incorporated at different levels depending on the extent of knowledge the user have.

C8 Even though user interaction is important and give tailored results there are still some parts that benefit from being automated. Specifically simple hole filling in this case. The goal of this tool is to let the user simplify holes such that automatic procedures can faithfully complete the job. The criteria is therefore supported.

C9 If any additional information is available, such as line of sight constraints it should be utilized to improve the model. However, incorporating additional information is very method specific. The tool does not have any specific means to incorporate any type of information, but extending the tool to incorporate new functionality should be possible. Mapping different types of information that usually are available might be needed.

C10 Again, a criteria that relies on the implementation to be fulfilled. Adding more data points or downsampling the data set to match surrounding geometry is however a fairly straight forward task and should be fully supported.

Chapter 6

Conclusions and Future Work

This chapter discusses the experience gained from this project. It talks about the conclusions that were reached and possible future work on this project.

6.1 Conclusions

This project has granted a lot of knowledge and expertise on the subject of hole filling. Even though the test case did not receive any significant improvements, there were some valuable experiences. The discovery that came as the greatest surprise was the fact that the test case consisted of several disconnected mesh fragments. In the literature a single connected data set were often presumed and this misleading assumption caused problems, as described in chapter 4. When surveying research it is therefore important, not only to understand and get a good impression of what is being stated, but also to realize what is not being said. Even such a simple method as the triangulation algorithm described in [28] turned out to have some problems that were not addressed by the article. For even more complicated methods this will probably also be the case. This is why practical testing of the different methods is so important.

The Nidaros Cathedral turned out to be quite a difficult test case. A simpler test case would probably have made implementation easier and allowed for testing of more methods, but the experience gained might not have been as useful. Duplicating previous work might be excellent for learning a procedure, but it does not contribute with anything new. For scanned models that does not fulfill the assumptions of the classic textbook example, auto-

matic methods proved to be insufficient. This did not come as a surprise, but rather the extent of the failure were unanticipated. With this is mind, an assumption is made that interactive and user guided methods will perform better. This, however, remains to be tested and confirmed.

Another issue were performance, hardware requirements and scalabilty. Scanned models contain vast amounts of data and need to be stripped or downsampled to become manageable. As hardware improves the model complexity usually increases to exploit the new technology. Algorithms should therefore not rely on better hardware to improve performance, but rather find an alternative solution that scales better for larger models. Of course using old and outdated hardware should be avoided. The test case consisted of approximately 1 million triangles, while models found at the Stanford 3D Repository contains between 1 million and 116 million triangles. To process a model that large scalabilty is extremely important.

6.2 Future Work

This project has been twofold. One side focuses on general hole filling and another side focuses on the specific case of the Nidaros Cathedral. These two problem definitions are closely linked and by solving the general problem the specific problem also gets solved. However, it is also reasonable to consider the future of both problems separately. Below follows some suggestions for future work on the proposed hole filling tool and the model of the Nidaros Cathedral, respectively.

6.2.1 The Hole Filling Tool

The hole filling tool has a potential to be developed further. Still, a lot of work remains before a full scale implementation can begin. A proper prestudy is needed to establish what kind of 3D editors exist and find a suitable application to incorporate the hole filling tool into. A proper requirement specification is also needed. This task sounds fitting for a system development course like for instance Customer Driven Project (TDT4290).

Also, it would be advisable to do some more testing of the individual hole filling algorithms. If a method is to be included in the hole filling tool it needs to be robust and have predictable behavior. As mentioned in chapter 5 there are several subjects that need more research before any final decisions

should be made. This applies to, for instance, interactive 3D model editing or sculpturing.

At this stage the hole filling tool is an idea. Several stages of planning, testing and implementation is expected before the product gets completed. This is a fairly large project, but in the end it would be a very powerful tool to enhance models from scanned 3D objects.

6.2.2 The Nidaros Cathedral

The 3D model of the Nidaros Cathedral also requires a lot more work. In the interest of creating a model for viewing purposes, such as visual presentations a plausible and visually pleasing model should suffice. But for the purpose of research, data storage and later model reconstructions the accuracy becomes more important. A plausible fill surface does not really contain any additional information. To get more ground truth data and a higher accuracy of the model, taking additional scans is recommended. By identifying regions with sparse or no information, these regions can be scanned. There will always be some regions that cannot be reached by a scanner, but limiting the size and amount of holes is necessary to obtain a realistic model. [35] suggests a scanning system that gives a real time update of the model as it is being scanned. The results are rather poor when it comes to quality, but it does give a good indication of which parts of the model that are covered by the scanner and where additional scans are needed.

When a plausible and visually pleasing model is sufficient other means to get a complete model can be utilized. The Nidaros Cathedral is still a difficult data set and current automatic methods are not capable of returning a complete model. However, methods based on user interaction seems promising. A smaller version of the proposed hole filling tool is likely a good solution. Allowing the user to draw on, insert rough shapes and edit the model is needed. The user should also have a good set of reference data, such as images, to guide the process. A combination of user controlled and automatic procedures such as the method proposed in [6] seems suited to fulfill the task.

Bibliography

[1] Nina Amenta, Marshall Bern, and Manolis Karnvysselis. A new voronoi-based surface reconstruction algorithm. *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, 1998.

[2] Nina Amenta, Sunghee Choi, and Ravi Krishna Kolluri. The power crust, union of balls and the medial axis transform. *Computational Geometry: Theory and Applications*, 2000.

[3] Nina Amenta, Sunghee Choi, and Ravi Krishna Kolluri. The power crust. *SMA '01: Proceedings of the sixth ACM symposium on Solid modeling and applications*, 2001.

[4] G. H. Bendels, R. Schnabel, and R. Klein. Detail-preserving surface inpainting. *The 6th International Symposium on Virtual Reality, Archaeology and Cultural Heritage*, 2005.

[5] G. H. Bendels, R. Schnabel, and R. Klein. Fragment-based surface inpainting. *Poster proceedings of the Eurographics Symposium on Geometry Processing 2005*, 2005.

[6] Gerhard H. Bendels, Michael Guthe, and Reinhard Klein. Free-form modelling for surface inpainting. *Afrigaph '06: Proceedings of the 4th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, 2006.

[7] Kristin Bjørlykke. Computational mechanics in civil engineering (cmc): 3-dimensional scanning and structural analysis, case-study nidaros cathedral. Technical report, Sintef, The Reasearch Council of Norway, 2002. pages 53-57.

[8] John Branch, Flavio Prieto, and Pierre Boulanger. A hole-filling algorithm for triangular meshes using local radial basis function. *Proceedings of the 15th International Meshing Roundtable*, 2006.

[9] J. C. Carr, R. K. Beatson, J. B Cherrie, T. J. Mitchell, W. R. Fright, and B. C. McCallum. Reconstruction and representation of 3d objects with radial basis function. *SIGGRAPH 2001, Computer Graphics Proceedings*, 2001.

[10] Chun-Yen Chen, Kuo-Young Cheng, and H. Y. Mark Liao. A sharpness dependent approach to 3d polygon mesh hole filling. *Proceedings of EuroGraphics 2005, Short Presentations*, 2005.

[11] NP complete. `http://en.wikipedia.org/wiki/NP-complete`.

[12] Marching Cubes. `http://en.wikipedia.org/wiki/Marching_cubes`.

[13] Brian Curless and Marc Levoy. A volumetric method for building complex models from range images. *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, 1996.

[14] James Davis, Stephen R. Marschner, Matt Garr, and Marc Levoy. Filling holes in complex surfaces using volumetric diffusion. *First International Symposium on 3D Data Processing, Visualization, and Transmission*, 2002.

[15] Tarnal K. Dey and Samrat Goswami. Tight cocone: A water-tight surface reconstructor. *SM '03: Proceedings of the eighth ACM symposium on Solid modeling and applications*, 2003.

[16] The PLY File Format. `http://www.cc.gatech.edu/projects/large_models/ply.html`.

[17] Radial Basis Function. `http://en.wikipedia.org/wiki/Radial_basis_function`.

[18] Constructive Solid Geometry. `http://en.wikipedia.org/wiki/Constructive_solid_geometry`.

[19] Tongqiang Guo, Jijun Li, Jianguang Weng, and Yueting Zhuang. Filling holes in meshes and recovering sharp edges. *SMC '06. IEEE International Conference on Systems, Man and Cybernetics, 2006.*, 2006.

[20] Xue J. He and Yong H. Chen. A haptics-guided hole-filling system based on triangular mesh. *Computer-Aided Design & Applications*, 2006.

[21] Volfill: A hole filler based on volumetric diffusion. `http://graphics.stanford.edu/software/volfill`.

[22] Java3D. `http://java.sun.com/javase/technologies/desktop/java3d/`.

[23] Tao Ju. Robust repair of polygonal models. *ACM Transactions on Graphics (TOG)*, 2004.

[24] Ravi Krishna Kolluri, Jonathan Richard Shewchuk, and James F. O'Brian. Spectral watertight surface reconstruction. *SIGGRAPH '03: ACM SIGGRAPH 2003 Sketches & Applications*, 2003.

[25] Hans Martin Langø and Morten Tylden. Surface reconstruction and stereoscopic video rendering from laser scan generated point cloud data. *Master's thesis, The Norwegian University of Science and Technology*, 2007.

[26] Marc Levoy, Szymon Rusinkiewicz, Matt Ginzton, Jeremy Ginsberg, Kari Pulli, David Koller, Sean Anderson, Jonathan Shade, Brian Curless, Lucas Pereira, James Davis, and Duane Fulk. The digital michelangelo project: 3d scanning of large statues. *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, 2000.

[27] Peter Liepa. Filling holes in meshes. *SGP '03: Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, 2003.

[28] Rui Liu, Darius Burschka, and Gerd Hirzinger. On the way to watertight mesh. *http://www.commission5.isprs.org/3darch07/*, 2007.

[29] MATLAB central MathWorks. `http://www.mathworks.com/matlabcentral/fileexchange/`.

[30] MeshLab. http://meshlab.sourceforge.net/.

[31] Finite Element Method. http://en.wikipedia.org/wiki/Finite_element_method.

[32] Joshua Podolak and Szynmon Rusinkiewicz. Atomic volumes for mesh completion. *SGP '05: Proceedings of the third Eurographics symposium on Geometry processing*, 2005.

[33] Polymender. http://www.cs.wustl.edu/~taoju/code/polymender.htm.

[34] The Stanford 3D Scanning Repository. http://graphics.stanford.edu/data/3Dscanrep/.

[35] Szymon Rusinkiewicz, Olaf Hall-Holt, and Marc Levoy. Real-time 3d model acquisition. *ACM Transactions on Graphics (TOG)*, 2002.

[36] Andrei Sharf, Marc Alexa, and Daniel Cohen-Or. Context-based surface completion. *ACM Transactions on Graphics (TOG)*, 2004.

[37] Andrei Sharf, Thomas Lewiner, Gil Shklarski, Sivan Toledo, and Daniel Cohen-Or. Interactive topology-aware surface reconstruction. *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*, 2007.

[38] SensAble Systems. http://www.sensable.com.

[39] Lavanya Sita Tekumalla and Elaine Cohen. A hole-filling algorithm for triangular meshes. *Tech. Rep., University of Utah*, 2004.

[40] Gerg Turk and Marc Levoy. Zippered polygon meshes from range images. *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, 1994.

[41] Jianning Wang and Manuel M. Olivera. A hole-filling strategy for reconstruction of smooth surfaces in range images. *XVI Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI'03)*, 2003.

[42] Songhua Xu, Athinodoros Georghiades, Holly Rushmeier, Julie Dorsey, and Leonard McMillan. Image guided geometry inference. *3DPVT '06: Proceedings of the Third International Symposium on 3D Data Processing, Visualization, and Transmission*, 2006.

[43] Øyvind Hjelle and Morten Dæhlen. *Triangulations and Applications*. Springer, 2006.