NTNU – Trondheim
Norwegian University of
Science and Technology

Merging Meshes from Different 3D Scanners

Dimitry Kongevold

Master of Science in Computer Science
Submission date: June 2013
Supervisor: Theoharis Theoharis, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

NTNU – Trondheim
Norwegian University of
Science and Technology

Faculty of Information Technology
Mathematics and Electrical Engineering
Department of Computer and
Information Science

Merging Meshes from Different 3D Scanners

Dimitry Kongevold

June 28, 2013

NTNU
Norwegian University of Science and Technology

Master Thesis

Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Computer and Information Science

Supervisor: Prof. Theoharis Theoharis, Dept. of Comp. and Inf. Science, NTNU

Abstract

Creating digital representation of physical objects is becoming more and more popular as scanning technology becomes more available on the consumer marked. High precision close range scanners, based on structured light, can produce high precision meshes in short amount of time, while automated photogrammetric software can process images taken with ordinary cameras to re-create a three dimensional scenes. Unfortunately the quality of digitalization is bound to the volume of the captured scene, creating a trade of between quality and speed of acquisition. In this master thesis we present software that can merge the output of two different scanners to create a combined mesh with regions of different resolution. We propose two methods to identify corresponding regions in meshes and a variation of greedy triangulation algorithm. At the end we perform tests on combined meshes to assess the quality of merging and present the results.

Norsk Sammendrag

Å lage digitale modeller av komplekse fysiske gjenstander har blitt mer og mer populært ettersom skanningsteknologien har blitt mer tilgjengelig på forbrukermarkedet. Skannere basert på strukturert lys, med høy presisjon, kan skape maskenett på kort tid, mens automatiserte fotogrammetriske programvare kan prosessere bilder tatt med vanlig kamera for å gjenskape tredimensjonale scener. Beklageligvis er kvaliteten av digitaliseringen avhengig av størrelsen til den gjenskapte scenen, noe som tvinger brukeren til å velge mellom kvaliteten og hastigheten på prosessen. I denne masteroppgaven presenteres en programvare som kan fusjonere produktet fra to forskjellige skannere for å skape et kombinert maskenett med områder med ulik oppløsning. Vi foreslår to ulike metoder for å identifisere tilsvarende områder i maskenett, og en variant av grådig algoritme for triangulering. Til slutt utfører vi tester på kombinerte maskenett og presenterer resultater.

Preface

This report is submitted in partial fulfilment of the requirements for the master's programme, MSc, of Engineering Computer Science at the Norwegian University of Science and Technology, NTNU. It represents the results of my master thesis carried out February 2013 - June 2013. My supervisor has been Professor Theoharis Theoharis of the Department of Computer and Information Science, NTNU.

Acknowledgements

There are several people I would like to thank, first of all my supervisor Professor Theoharis Theoharis of the Department of Computer and Information Science, NTNU, and Associate Professor Emeritus Torbjørn Hallgren of the Department of Computer and Information Science, NTNU, for their help and guidance.

I thank my family and close friends for their great support during my studies.

I would like to thank the computer lab at my elementary school, Moscow School 45, for waking up my interest for computer science and information technology.

Finally I thank the Department of Computer and Information Science, NTNU, for making this master thesis possible.

To people who feel left out - my deep apologies.

Dimitry Kongevold
Trondheim, June 2013

Contents

List of Figures

List of Tables

List of Algorithms

Part I

Introduction

Chapter 1

Introduction and problem description

This chapter provides some perspective on the work described in this thesis, defines the problem, states main contributions of this thesis, and finally presents an outline of the rest of the master thesis.

1.1 Introduction

In last decades we have experience a major increase in computing power. The size of devises has reduced while computational power they offer has increased. The cost of the hardware has also reduced dramatically. All those factors play a role so that visualization and digitalisation of physical scenes and objects is becoming increasingly popular. For example in the field of archaeology high precision scans are used to measure erosion, constriction sites and oil-rigs are scanned to document and observe construction process, museums try to digitalize artefacts in order to preserve the past, artists digitalize there art to share it online or manufacture it in different scales and quantities.

When using a 3D scanner, based on optical sensing the resolution of the surface and amount of the captured per frame is dependent on the distance from the scanner to the object. With increasing distance the quality of the captured surface decreases, while size of scanned surface increases. The user is forced to choose between lower complexity of the scan and high precision of the result. In this thesis we try to address this problem by proposing software that can merge two different scans of

the same object, originated from different scanners and or of different resolutions.

1.2 Literature Overview

Using existing meshes to produce a combined result, is often used by artists to create exotic 3D models, as art, entitlement and for manufacture. The tools created for this purposes is based on research as (Biermann et al., 2002; Sorkine et al., 2004; Huang et al., 2007; Schmidt and Singh, 2010). Though creating astonishing results, there methods change topology of the meshes, which is not acceptable if accurate measurements are required. Using multiple over lapping meshes from the same source to create a mesh representation of a physical object was proposed by Pito, R. in paper (Pito, 1996), this method is based on the knowledge about the scanning process, and only considers meshes of the same resolution. Paper by Morooka, K. and Nagahashi, H. (Morooka and Nagahashi, 2006) merge meshes of different resolution, originated from the same scanner with known orientation to the scanned surface. Our approach differs from the others, as origin of the triangular meshes is not taken into consideration. The only requirements for our approach are that meshes contain an overlapping region of interest, and the data mesh is of a higher resolution.

1.3 Problem Description

In the digital world the value range is always limited. The maximum and minimum bounds are often restricted by the precision and architecture of the hardware. In the case of photogrammetry the hardware limit comes from the captured images, in digital case this would strongly rely on image-sensor chip size. The same can be applied to structured-light scanning and laser scanning based on trigonometry.

The process of taking photographs with a digital camera can be looked upon as creating a discrete representation of continues function. The precision of discrete representation is based on the number of samples and how accurate they are. Due the pin-point-camera model of today's sensors, the aria that is being sampled depends on the distance to the surface and the optics of the sensor. In figure 1.1 we can see a sketch of a sensor and the area that is sampled in position A and B.

The rate with which the sampled aria increases with increasing distance is based on the angle beta. Beta represents the maximum angle between the rays of incoming

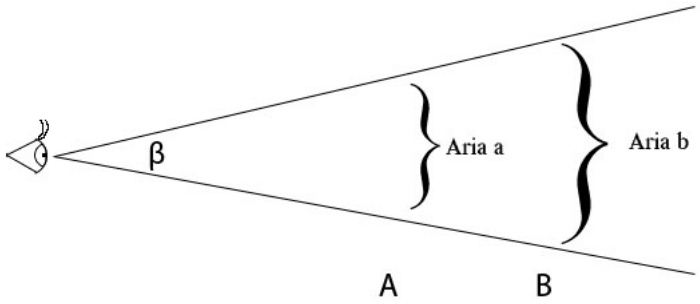

Figure 1.1: Scaned aria increases with the distance to the scanner

light that are observed by the sensor.

The resolution of a captured image can be described as sampled aria of the surface divided by the number of samples. The maximum number of samples is based on the sensor chip and is constant for images taking with the same camera, while the sampled aria on the other hand can vary, leading to variance in resolution. This creates a trade-off between the size of the aria captured per one frame and the resolution of captured frame. In real world the precision of the captured image depends on many more factors as focus length and circle of distortion, global illumination, reflective properties of the surface, and so on. But the basic principle stays the same; the maximum precision of a frame is based on number of samples and size of sampled aria.

Many of scanning technique suffer from this trade of, ether scan large aria with few frames, or scan a smaller aria with higher resolution. In practice this choice can stand between choosing right type of scanner, or between different settings for the same technology. Photogrammetry is a modelling technique that is easiest to scale. For example the photogrammetry software Photomodeler Scanner can take as an input both photographs of Eiffel tower and photographs of a hundred times smaller replica, and produce the same digital result. This scalability is the result of correct camera calibration in each situation.

Human made objects have often varying grade of details. An example to this would be a cathedral wall in gothic style, where as we can imagine the most of the surface

is smooth with little details but some parts of the wall have often sculptures or figurines carved out. In the terms of topology the smooth part of the wall contain less information than parts with high amount of details.

When scanning a large surface with high variance the question, of the size versus the resolution of a frame stands out. High intensity sampling of smaller arias will give a best result in the terms of precision, but the practical problems arise, as large time consumption and logistics. Usually better resolution leads to a shorter distance to the object, and large volume of data to store.

If we return to the gothic-style cathedral-wall example, given that the resolution of the mesh is constant but information amount vary from aria to aria, the information per point will vary as well. Giving us arias where each individual point contains little or no additional information to the mesh. The points that contribute little to the meshes geometry can be removed by mesh simplification algorithms. Though mesh simplification is ongoing research some good results have been archived as (Hoppe, 1999), (Daniels et al., 2008) and (Wei and Lou, 2010) Quad meshing has shown the best results in mesh simplification in resents years, the overview of the quad mesh advantages and draw backs can be found in (Bommes et al., 2012). Mesh simplification will reduce the number of faces and vertexes, reducing storage size and processing time, but this technique will not solve the problem with logistic, and acquisition time.

The Idea behind *MeshCombine* project is to use multiple scanning techniques to produce a final model. Create a total overview with a low resolution mesh and keep level of details high in the regions of interests (ROI) with a high density mesh. This approach is not new in image compression, and is strongly researched for medical image processing, some examples of literature can be found in (Gokturk et al., 2001). The compression standard JEPEG2000, has the possibility to encode ROI (Bradley and Stentiford, 2002), (Christopoulos et al., Sept. 2000). When simplifying mesh with the right cost/feature function, region of Interest effect will be automatically archived, as simplification often remove edges with low or no information, but as mentioned above mesh simplification will not increase simplicity of scanning process.

The vision of the *MeshCombine* approach is that the user scans the large object producing a rough approximation, then by using different settings, or hardware the user captures the details of an object that are considered important. A similar approach is used in (Morooka and Nagahashi, 2006) where the authors propose a method to recreate a model by combining scans originated from the same scanner but with different resolution. When the meshes are combined in the *MeshCombine* software, they produce single mesh with arias of different resolution and level of

details. This way user only needs to perform complicated scan in high resolution in regions that require high level of detail, and exploit the practical simplicity of a lower resolution scan. In our case we will use photogrammetry to create less detailed overview mesh and structured light scanning to perform a high resolution scan of details, chapter 2.

1.4 MeshCombine

On the marked today you can find several products for mesh manipulation like zBrush (Pixologic, 2013), Autodesk 3Ds Max (Autodesk, 2013a), and MeshMixer (Autodesk, 2013b). These products provide powerful tools for a 3D artist to create exotic models for entertainment purposes, in video games and animations, they produce astonishing visual results but in the process they change the mesh's topology. This is unacceptable in the case of extracting exact real world data as, in visualization of a construction site or preserving archaeological artefacts.

MeshCombine is a simple prototype of a software that can be used to merge two scans of different resolution and origin together into one model. The lower resolution mesh, a model, will work as overview to determent the position, for the high precision scans of the details, lately referred as data. The merging process can be summarized in few steps:

- The user marks the aria on the model, where the data mesh approximately should be and start the algorithm.
- The software runs a registration algorithm on in the user selected subset of the mesh, to find optimal position for the data.
- When the model and data mesh are registered, the vertices in the model that are represented by the data are deleted and data vertices are added to the result.
- The gap created by the deletion is then filled and new model is represented.

The only alterations done on meshes are registration of meshes, deletion of vertices in the model, and filling of the gap between model and data meshes. These are two sources of error that we introduce to the new mesh. Because the vertices deleted and hole filled belong to the lower resolution mesh the error does not propagate to the data mesh. This preserves high resolution in the regions that are important for the user, at the same time provides positional information of the data mesh in

respect to the model mesh.

MeshCombine can be useful in applications as forensics, constructions and restoration, preservation of the past, and when modelling a scene with multiple resolutions.

1.4.1 Similar Open Source Projects

Open source projects give possibility to other developers to use their source code in their own projects, with some legal limitations. Open source community tries to focus research work on new thinking, rather than implementing well known algorithms, by providing easy access to previously implemented work. In our case two open source projects has been evaluated in possibility to reduce the workload, Point Cloud Libraty (PCL, 2013) and CloudCompare (CloudCompare, 2012). Both projects are immense and cover many aspects of *MeshCombine*.

Under closer examination it was determent that the use of ether implied to use a specific data structure and C++ programming langue, and since previously it was decided that *MeshCombine* would be written in C, the use of libraries would require wrappers or bridges. Using the open source libraries would also complicate the architecture of the product. *MeshCombine* only requires some of the functions implemented in the libraries. It is possible to only include the required source files to archive the results. Including only functions that are necessary would require having a total over view of the third party software. Due many dependencies and the immense size of the projects, to gain an over view would require more resources then implementing required functionalities from scratch.

The Point Cloud Library and CloudCompare were used as a guide and a source of inspiration during the developing phase of this project.

1.4.2 Registration by ICP

Iterative Closest Point algorithm was proposed by besl and Mc kay in 1992 (Besl and McKay, 1992). The purpose of the algorithm is to register two shapes, model and data. The algorithm tries to find optimal transformation for data set, to minimize mean root square distance between data and model.

Since 1992 many alterations and improvements have been proposed, the main focus was to reduce computing time, ether to reduce number of calculation or to reduce

cost per iteration. The other direction was to make ICP more robust to noise, and avoid local minima. In the paper (Rusinkiewicz and Levoy, 2001) Szymon Rusinkiewicz and Marc Levoy perform comparison of the different ICP variants, based on run time and quality of the result.

1.4.3 Mesh Merging

The idea of using multiple range images to recreate a model is not new in computer science (Dorai et al., 1998; Rutishauser et al., 1994; Turk and Levoy, 1994; Sappa and Garcia, 2000) Though there are few that consider model integration with different resolution. Papers like (Pito, 1996) and (Morooka and Nagahashi, 2006) use the a priori knowledge about the scan acquisition process to determent the quality of a vertex or a path, this information about range image acquisition differs from our work, to make a more general tool, information about capturing process is not available for our purpose. The only knowledge given about the input is that model has lower value for the user, than data, leading us to preserve all of information in the data mesh, and apply changes only to the model.

1.5 Main Contribution

Scanning complex physical objects can be difficult. Practical problems like logistics, data storage, and time consumptions can lead to the choice of hardware that recreates the scene with lower overall resolution. The idea of *MeshCombine* project is to create a small Graphical User Interface, GUI, where the user can merge meshes with different resolutions, from different scanners, to archive a higher resolution in regions of interest. Our approach can not add noise to the region of interest, and only deletes vertices in model mesh. The main contributions of this thesis are considered to be:

- Implementation of a small GUI, where meshes are displayed and user can navigate in virtual scene.
- Implementation of merging tool where two meshes are merged based on user input.
- An algorithm for duplicated region estimation.
- Proposal for a greedy triangulation algorithm variant.

1.6 Thesis Organization

This thesis is divided into five parts: Introduction, Technology Overview, Implementation, Tests and Procedures, and Conclusions and Future Work, followed at the end by Appendix.

Technology overview part contains chapters two to five. The scanning technologies used in this master thesis are presented in chapter two. Chapter three goes through visualization. This chapter explains how output on a screen is drawn, orientation of virtual scene, using OpenGL and FreGLUT to render graphics, techniques on interpreting user mouse input, and tree structures often used in computer graphics. In chapter four the ICP algorithm proposed by Besl and McKay in 1992 is summarized, as well as some of the variations of the algorithm are presented. The approach to merge two meshes of different resolution from different scanner is proposed in chapter five.

Implementation part describes implementation of the proposed tool for mesh merging. We start with the Graphical User Interface in chapter six, where possible actions that a user can take user are described. Chapter seven is dedicated to explain how meshes are stored, by presenting Wavefront .obj format and Stanford's .ply format, and describing how meshes are stored in memory during run-time of the program. In chapter eight, nine and ten, three vital parts of merging process are described. Registration by ICP chapter nine, greedy triangulation chapter ten, and overall mesh fusion with duplicated region estimation algorithm chapter eight.

In part four, Tests and Procedures, test and test objects are introduced, followed by the results of testing. Chapter eleven describes tests and objects, while chapter twelve presents the results.

Conclusions and Future Work, firstly, in this part we interpret and discuss the archived results in Chapter thirteen, and, secondly, draw the conclusion and propose possibilities for future work in chapter fourteen.

In appendix we provide some auxiliary information, generally about implementation of *MeshCombine*. Appendix A provides insight to the composition into *MeshCombine*. Appendix B explains how octTree.h is implemented and how it used, while Appendix C focuses on KD-tree library implementation. In Appendix D usage of dynamic List implementation is explained. Finally standard library's function memcpy is described in Appendix E.

Part II

Overview of the Technology

Chapter 2

Scanning Technologies

This chapter gives a perspective on scanning technologies used in this thesis. Short overview of structured light scanning and photogrammetry is given, and used modeling technologies are presented.

2.1 Structured Light Scanning

Structured light scanning is a 3D modeling technique that projects a known pattern into the scene, and then captures one or multiple 2D images from a different perspective, to extract 3D geometry information. The method is based on the principle that if the narrow band of light is projected on uneven surface it appears to be distorted from a different perspective. Some early work in the field dates back to 1971 and 1981 (Will and Pennington, Winter 1971), (Minou et al., 1981) Since then the research has been driven in two directions: reducing the acquisition time and increasing the depth resolution, with the significant results in both (Salvi et al., 2010).

The drawbacks of structured light scanning are that the technique relies on interpreting the observed distortion of a pattern from a known position. The scenes that contain large amount of global illumination, reflective surfaces or transparent objects, reduce the quality of the results. So do all other 3D capturing techniques that rely on optical sensing.

2.1.1 Artec EVA 3D Scanner

Artec EVA is a 3D Scanner designed and manufactured by Artec Group. The scanner is state of the art technology based on structured light triangulation. Figure 2.1 shows an image of 3D scanner, it consist of three cameras. Two on each end of the scanner are used for geometry capturing and one in the middle for texture. A ring of flash diodes placed in a circle around camera in the middle produce a flash of structured light. This pattern captured by geometry cameras and analysed to calculate 3D data. To operate the scanner the user needs to install the software provided by the Artec-group, 'Artec 3D Studio v8.1'. The scanner is inoperative without the software and will not start scanning before it detects Artec Studio installation on users' working station.

Figure 2.1: Artec EVA 3D Scanner. (ArtecGroup, 2012)

In principle the scanner operates as a video camera the difference is that the output is not a stream of video frames but a stream of point-cloud frames with optional texture frame (standard video frame). This stream is then send to the computer connected to the scanner by a USB2 cable, where the frames get auto aligned, or registrated by the software, in real time.

From the mathematical perspective the scanner uses the a priori knowledge of angles between two depth-cameras and a known pattern projected on the object, to triangulate the depth of each pixel in the picture. More detailed information can be found in literature as, (Blais, 2004), (Devrim et al., 2006) and (Georgopoulos et al., 2010) to name a few.

The real time registration of point-cloud frames is handled by the software on users

working station. The Artec 3D Studio provides user with 3 options for registration, or as it reefed in the manual 'Features to track'. The scanner can use geometry of point clouds, it can use textures of frames or it can use both to align the frames compared to each other. The last option geometry and texture will provide the best result, but requires most CPU resources.

Artec 3D Studio also provides post-processing opportunities as the alignment of the scans (a set of 3D frames), merging of scans, and editing. It gives global methods as 'Global registration', where all of the frames in all the scans get aligned, 'Fusion' where all scans are merged together and all the unnecessary or duplicated points are eliminated so the result is a simple triangulated mesh.

Artec EVA can produce a highly detailed result, with the 3D resolution up to 0.5mm in accuracy (ArtecGroup, 2012). The process is highly optimized and automated. The limitations of the scanner comes with the focal length of the cameras, as EVA can only capture points that are in distance from 0.1m to 0.4m from the cameras.

Structured light scanning becoming more popular as technology improves, it can provide 3D images with accuracy close of laser scanners, in a fraction of the time. This opens the possibility to scan objects that cannot remain still for long period of time, as humans and animals.

2.2 Photogrammetry

Photogrammetry is a science of retrieving geometric measurements from photographs. The basic principle is, that from different positions and perspectives the same object will appear differently, based on the distance to the camera and cameras orientation. By finding intersection of converging rays in space, the precise location of a point in the scene can be determined. The resume of some of industrial applications of Photogrammetry can be found in (Fraser, 1993). Today with the adoption of digital sensing and computer vision techniques as (Lowe, 1999) and (Matas et al., 2002), photogrammetry can be automated to produce faster, and more accurate results.

Photogrammetry is easily scaled from modeling large constructions as buildings and ships, to small parts and ornaments, making it applicable for a wide variety of applications. The resolution of recreated models is highly dependent on the resolution of acquired images, leading to lower accuracy when recreating large objects.

2.2.1 PhotoModeler Scanner

> *"PhotoModeler photogrammetry software provides image-based modeling, for accurate measurement and 3D models in engineering, architecture, film, forensics. . . ."* (EosSystems, 2012a)

PhotoModeler Scanner is 3D modelling software based on photogrammetry, developed by Eos Systems. Eos Systems delivers to versions of the program PhotoModeler and PhotoModeler Scanner. More detailed information is available on (EosSystems, 2012a). In this project the PhotoModeler Scanner will be used as it provides SmartMatch feature that provides fully automated 3D recreation.

In the resent years photogrammetric modelling became more attractive led by development of automation of the close range photogrammetric procedures due the adoption of computer vision methods. Quoting (Athanasios et al., 2012) on cooperation between photogrammetric and computer vision communities

> *"Even though the two communities have been working almost independently till the year 2000 (Forstner, 2009) this quickly changed as it became clear that the combination of the techniques used by both communities could lead to serious advances in the automation of the close range photogrammetric procedures. The introduction of tools like SIFT (Lowe, 1999) or MSER (Matas et al., 2002) that can reliably extract dense features from overlapping images. . . "*

Scale Invariant Feature Transform(SIFT) (Lowe, 1999) is highly relevant to PhotoModeler Scanner. Though it is impossible to investigate, it is save to make an assumption that 'SmartPoints' used in 'SmartMach' projects are variations of SIFT, that makes automatic camera orientation possible.

The program could be operated by the user unknown to photogrammetry, as many of the algorithms are automated and do not require any user input. The algorithm for camera calibration is example of such.

The Photomodeler Scanner and similar photogrammetric technology provide an easy way to capture and digitalize a 3D scene. The quality of the created 3D mesh relies on the resolution of the captured images, which is dependent on factors as, lens and lens distortion, size of the cameras photometric sensor, and focal length and distance to the object. With the right setup of all factors large scenes can be captured in determined resolution just in a few hours.

Chapter 3

Visualisation

In this chapter the reader will be introduced to, modern computer graphic techniques, how they work, the control flow, and the math behind. We start from the basic principle of primitive rasterization. Then proceed with object orientation and position in world's coordinate system. Then we will explain clip space and the magic behind perspective projection.When we are known to the theory we will take a look at modern graphics hardware pipeline, shaders and shading languages. At the end will introduce the reader to OpenGL specification and how it can be used to produce astonishing 3D graphics.

3.1 Todays Computer Graphics

3.1.1 Rasterization

No matter what the programmer does in software it all comes out on the same screen.

Devises that could produce images drawn on the screen in continues lines were Vector-monitors. They were a type of CTR(Cathode Ray Tube) the difference was that they drew a line from point to point, not a in a vertical line pattern as rest of CTR based televisions sets. Close to all of today's computer graphics displacing devises are discrete. Meaning that devise can only draw a point on a screen not a connected line. To draw a line it must be divided in a set of connected points, this process of diving in to pixels is called rasterization. (Bresenham, 1965) is a paper written by computer graphics pioneer Jack Elton Bresenham, describes

what is now known as a Bresenham line algorithm, which draws continues line between two points..

Drawing complex images with only lines can be complicated, this is why usually the programmers provide other primitives, triangle and hexagons are most popular. To draw a filled triangle on a screen some math must be done. A triangle shell is described by its three vertexes, $\mathbf{p}_1$, $\mathbf{p}_2$, and $\mathbf{p}_3$. Every point in a triangle can be described in a parametric equation

$$p(s,t) = \mathbf{p}_1 + s\mathbf{v}_1 + t\mathbf{v}_2 \tag{3.1}$$

Where s and $t \in [0,1]$ and $\mathbf{v}_1$ and $\mathbf{v}_2$ are edges $\overline{\mathbf{p}_2\mathbf{p}_1}$ and $\overline{\mathbf{p}_3\mathbf{p}_1}$ respectably. Drawing a triangle can be done incrementing s and t values, to produce two points on each side of the triangle, and rasterizing a line in between.

3.1.2 Digital 3D Models

In computer graphic the focus is to recreate the visually observed part of an object. When representing a volume it is enough to recreate a shell of a model, the surface.

Surfaces can be represented discrete or continuously depending on what purposes the model serves. In computer simulations continues surfaces are often used, parametric and implicit surfaces. The continuity is often required when running calculations. Some of the parametric curves secure the continuity over second and third derivative, as b-splints and NURBS.

Though it is possible to approximate a parametric curve and draw it on a screen pixel by pixel, most of today's graphics implementations split curves into primitives, like square slices or triangles, before displaying it on the screen, producing a discrete surface. This is because pipe line to draw many simple primitives is heavily optimised by hardware manufactures.

Both discrete and parametric surfaces have a set of points to contain the information about surface orientation in the world. A parametric surface is described by set of control points, equation, and sometime weights given to points. A discrete surface contains a list of vertexes and connectivity information, as a list of edges or primitives.

3.1.3 Geometric Transformations

To recreate a realistic looking world in computer graphics the artist need to have a possibility to move and rotate objects. As previously mentioned 3D models are often a set of point, or equations that describe a surface of a model. As long as we try to model realistic world, those points, will it be vertexes in 3D primitives or control points in parametric surfaces, can be described by Cartesian coordinates.

A point described by Cartesian coordinates can be easily translated by adding or subtracting a vector. Rotation on the other hand is more complicated. A point can be rotated around and axes by performing trigonometric operations. For example the rotation around the z-axes for a point $\mathbf{p} = [x, y, z]^t$ by and angle θ is given by set of equations

$$\begin{aligned} x' &= xcos(\theta) + ysin(\theta) \\ y' &= xsin(\theta) + ycos(\theta) \\ z' &= z, \end{aligned} \tag{3.2}$$

where x', y', z' are coordinates after rotation.

By using the linear algebra transformations can be stored in matrixes and applied to points coordinates in a vector form, we can write rotation operation as a 3 by 3 matrix

$$\mathbf{M}_{rot} = \begin{bmatrix} cos(\theta) & sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.3}$$

Then the rotated point will be

$$\mathbf{p'} = \mathbf{M}_{rot} * \mathbf{p} \tag{3.4}$$

The combination of translocation and rotation can produce any kind of orientation of an object in 3D world. Rotation of an object about a vector given by points **P1** and **P2** is just a set of 5 operations, 2 translocation and 3 rotations. In pseudo code this will look something like this:

1. move the object so p1 is at the origin
2. rotate so that $\mathbf{p}_2$ lies on z-axes
3. rotate the wanted amount of degrees
4. reverse rotation applied in step 2

5. reverse translocation in step 1

This can be summerised into this set of operations

$$\mathbf{p'} = \mathbf{T}^{-1} * \mathbf{R}_{xy}(\alpha)^{-1} * \mathbf{R}_z(\theta) * \mathbf{R}_{xy}(\alpha) * \mathbf{T} * \mathbf{p} \tag{3.5}$$

The rotations can be rewritten using quaternion to produce the transformation matrix (Hearn et al., 2010)

$$\begin{bmatrix} u_x^2(1-\cos\theta)+\cos\theta & u_xu_y(1-\cos\theta)-u_z\sin\theta & u_xu_z(1-\cos\theta)+u_y\sin\theta \\ u_yu_x(1-\cos\theta)+u_z\sin\theta & u_y^2(1-\cos\theta)+\cos\theta & u_yu_z(1-\cos\theta)-u_x\sin\theta \\ u_zu_x(1-\cos\theta)-u_y\sin\theta & u_zu_y(1-\cos\theta)+u_x\sin\theta & u_z^2(1-\cos\theta)+\cos\theta \end{bmatrix} \tag{3.6}$$

Multiple transformations can be applied simultaneously on a point, because of associative property of linear algebra. But the order of multiplications must be preserved. The expression in 3.5 can be rewritten as

$$\mathbf{p'} = (\mathbf{T}^{-1} * \mathbf{R}_{xy}(\alpha)^{-1} * \mathbf{R}_z(\theta) * \mathbf{R}_{xy}(\alpha) * \mathbf{T}) * \mathbf{P} = \mathbf{M}_{orientation} * \mathbf{p}$$

The orientation of the camera in the world is often saved as one matrix, to save memory and computational time. When the orientation matrix needed to be updated by new transformation it can be done by multiplying new transformation matrix by old orientation matrix

$$\mathbf{M}'_{orientation} = \mathbf{M}_{transform} * \mathbf{M}_{orientation} \tag{3.7}$$

3.1.4 Virtual Scene and Spaces

The scenes in computer graphics are assembled by objects, will it be in a computer game, structure visualisation, or virtual reality. The objects them self are combination of smaller objects and they themselves are set of even smaller objects, this can go on to the level of details the artist requires to recreate a realistic scene. When animating objects some parts have to move relatively to the rest of an object, an example of that will be feet on a running humanoid. To cope with that the scene is divided into multiple spaces. A humanoid foot has it coordinates in the model space of a humanoid, while the humanoid object itself has coordinates in the world space. This way an artist can move the humanoid, with all of its feet around the scene, while ensuring that each foot is in correct position towards the

body. Coordinates in World-space describe the orientation in the total scene and coordinates in the Model-space orientation towards the objects origin.

Another important space in the scene is camera space. As the world is observed from the cameras perspective, orientation of the camera decides what objects are visible and what are not. The camera space, coordinate system is defined by cameras tilt, rotation, and position. Usually the x and y axis define the viewing plane and z-axis define negative viewing direction.

Most of the displaying devises today produce a 2D frame. To be able to transform 3D scene into a 2D images, World-coordinates must be converted to Normalised-clip-space coordinates. It is called clip space because here the parts of the scene that are not within a specific bound are removed, usually the point with the coordinates x, y, and z that are $\notin$ [-1, 1] are clipped.

When the points are in Normalised-clip-space are easy to draw in 2D, if your system does not support alfa-blending then the colour of the pixel $p(x, y)$ is the colour of the point in Normalised-clip-space with least z value where x and y coordinates equals to the pixels $C_{pixel}(x, y) = \min_{z\in[-1,1]} C_{point}(x, y, z)$

3.1.5 Perspective projection

The human eye is analogical to a pin-point-camera, and so human observe the world in a perspective projection manner, making objects far away appear smaller then in close. To create this illusion on the 2D screen, points must undergo a transform. As light passes through the iris of a human eye, we can imagine that light passes through the viewport on the screen, creating a frustum, figure 3.1.

The shape of the frustum are determined by aspect, ration between the length of viewport in x and y direction, and z-near and z-far plane. The transformation between world coordinates and perspective coordinates is given my matrix

$$\begin{bmatrix} \frac{-2z_{near}}{xw_{max}-xw_{min}} & 0 & \frac{xw_{max}+xw_{min}}{xw_{max}-xw_{min}} & 0 \\ 0 & \frac{-2z_{near}}{yw_{max}-yw_{min}} & \frac{yw_{max}+yw_{min}}{yw_{max}-yw_{min}} & 0 \\ 0 & 0 & \frac{z_{near}+z_{far}}{z_{near}-z_{far}} & \frac{-2z_{near}z_{far}}{z_{near}-z_{far}} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{3.8}$$

For more detailed information about perspective projection please refer to the page 351 in the text book (Hearn et al., 2010)

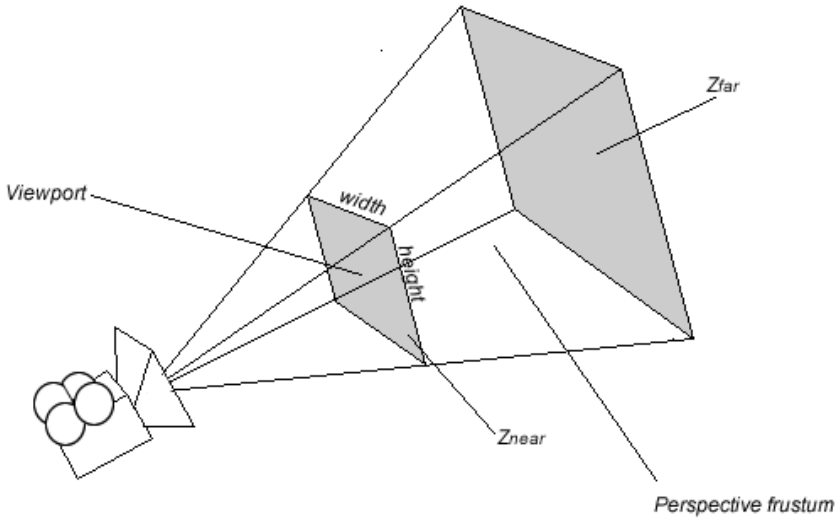

Figure 3.1: Frustum created by viewport at z-near and z-far plane.

The homogeneous coordinate, $\mathbf{p} = [x, y, z, w]^t$ have not been mentioned yet. The reason why homogeneous coordinates are used is to cope with affine transformation as scaling and in this case perspective transform. The perspective transform does exactly scaling based on the distance from the z-near. The reason for the name Normalised-clip-space is, because points undergo normalization, the division by w, before they are passed to Normalised-clip-space.

The general pipe line of transformations will be:

MC => WC => PC=> NC

Model-coordinates are transformed into Word-space, from there they are transformed by perspective projection, and after that they are normalised into Normalised-clip-space.

3.1.6 Z-buffer

Z-buffer or depth buffer is a way to identify occlusion when rendering a digital scene. The idea is weary simple. Every time a pixel is to be drawn to a frame buffer the system checks if there for a value on the same 2D position in z-buffer. If the value in the z-buffer is higher that the pixel's z-coordinate value, than no occlusion occurred and the colour of the pixel is drawn in a frame buffer. On the other hand if the value in the z-buffer is lower than the pixel is occluded by the previously drawn pixel and thereby is ignored.

3.2 OpenGL

OpenGL (Open Graphics Library) is a non-profit multi-platform graphics API that provides support for rendering 2D and 3D graphics. The API was developed by Silicon Graphics Inc. in 1992, and from 2006 to the present date the Khronos group has the control of the OpenGL specification.

OpenGL can be looked upon as a wrapper around the hardware specific implementation of a GPU. This separates implementation based on hardware from general logic implemented in software, giving the possibility for hardware manufacturers to write a highly optimized code for their product, without the lost of portability for software developers.

OpenGL is a common ground between hardware that puts images on the computer screen and software that decides what images should be drawn on the screen.

3.2.1 OpenGL Drivers

It is important to point out that OpenGL is not an implementation, but a specification, a common langue between software program and a GPU. The implementation of OpenGL on for the hardware is based on the driver provided by manufacture. OpenGL specification only formulates the input and results of the function calls leaving the freedom of implementation to the hardware manufactures. This not only provides software developers with the portability and efficiency for their software, but it also gives driver programmers possibility to change the implementation of OpenGL, due optimization or bug fixes through the driver updates, without the need of informing software developers about changes.

3.2.2 Loading OpenGL Functions.

Though all of the Windows versions above Windows 95 support OpenGL through the opengl32.dll library (KhronosGroup, 2013a), it only exposes direct control over OpenGL 1.4 functions. To gain access to the functions from the higher versions of GL the developer must load functions pointers manually from the running OpenGL context. More information can be found at (KhronosGroup, 2013b). The Khronos group highly recommends using OpenGL Loading Libraries like, GLEW and GL3W.

3.2.3 OpenGL Evolution

The demand for the computer graphics has changed during last two decades. GPUs became more powerful and advanced. The norm of what is considered "Good" graphics has changed dramatically in the game industry. To cope with this continues change new versions of OpenGL specification were created; up to today the latest version is OpenGL 4.3. OpenGLs status and evolution is governed by The OpenGL Architectural Review Board (ARB), which decides what extensions will be adopted in the next version or promoted to a ARB_Extention, an extension considered useful and specified by ARB but does not necessarily must be implemented by hardware manufacture in a OpenGL implementation.

OpenGL 3.1

OpenGL 3.0 and 3.1 deserves some special mention, as they brought the most noticeable change. In 2008 Khronos group announced that some of the functions that were supported in previous versions are deprecated in OpenGL 3.0, meaning that those functions will be removed from future versions of OpenGL. The biggest change was to automated primitive pipeline that, enabled user with minimal knowledge of computer graphics and rasterization, ability to render objects on the screen. For instance Begin/End primitive specification was deprecated and later removed. In exchange the user was given more direct access to OpenGL pipeline, where rendering should be programmed by using shaders. Making it easier to render complex graphics, but for trivial tasks, as render a 2D-rectangular on the screen, the user must invoke the same machinery as for a complex 3D model.

A complicate history of OpenGL changes can be found on OpenGL website (KhronosGroup, 2013a).

3.2.4 OpenGL Programs and Shaders

Shaders are small programs that are compiled and send to the GPU, where GPU runs the given shaders on user described input. There are two main types of shaders vertex shaders and fragment shaders.

Vertex shaders execute calculations based per vertex, while fragment shaders' responsibilities are pixel output per primitive. The main task of a vertex shader is to apply geometric changes to individual vertex, as rotation and translocation, and projection, while the main task of a fragment shader is information based on each fragment (pixel) of a primitive, as colour, visibility, and depth in Z-buffer.

OpenGL program is a combination of a vertex shader and a fragment shader. Each shader must be compiled and linked together, to form an OpenGL program.

In OpenGL 3.1 and higher versions the only way to render arrays of vertexes to the output buffer, or the screen, is to use an OpenGL program, and a VAO (Vertex Array Object).

3.2.5 VAO and VBO

Vertex Array Object(VAO) and Vertex Buffer Object(VBO) are important part of OpenGL pipeline. Buffers contain data needed to create output and VAOs contain information on how to use the buffers.

Vertex Buffer Object is first of all a buffer, an allocated memory on GPU. Buffers are designed to store data; two mainly used types of buffers are GL_ARRAY_BUFFER and GL_ELEMENT_ARRAY_BUFFER. Typical use of an array buffer is to store data about the each vertex, as position, colour, and normal, element array buffer is a special case that only contains information about each primitive, index of each vertex in a triangle, line segment, triangle-fan, and so on.

Buffers can contain all sorts of information, and not only information needed per vertex. A buffer can be uniform, GL_UNIFORM_BUFFER, meaning that every vertex in rendering will have the same access to the uniform buffer. A perfect example for that will be camera-to-clip-space transform. In a case of perspective projection every vertex's coordinates need to be transformed from camera to clip space where OpenGL's geometry shader operates. This can be done by multiplying perspective-projection-transform-matrix with vertex's coordinate's vector. This matrix can simply be stored in a uniform buffer. It is only developer's imagination that limits the kind of information you can store in a VBO.

Vertex Array Object stores relations between buffers and programs attributes. You can use same buffer multiple times in one or multiple VAOs. It is up to the developer to know what to put in a buffer and how to correctly interpret stored data in a buffer.

While VBO is just a place holder for data, VAO is the record of what data to read and how to read the required data.

3.3 FreeGLUT

FreeGLUT is an open source project that used OpenGL Utility Kit (GLUT) as ground model. It is used to create and maintain multiplatform windows context, as well as to provide user input in form of mouse, keyboard, and joystick.

FreeGLUT library is not a part of OpenGL it is a wrapper around operating system window, and OpenGL contents. It provides user with abstraction from communication with operating system and holds control over input and output buffers where users input or output of rendering is stored.

To avoid confusion we define a programmer as the person who writes program's source code using freeGLUT library and a user as a person how operates the programmer written software.

3.3.1 Main Architecture

FreeGLUT can be described as complete machinery with the possibility to change functionality by manipulating settings and adding functions pointers at given entries.

3.3.2 Function Call-back

To understand how freeGLUT operates we must understand the concept of call-backs. In C programming langue, in which freeGLUT, GLUT, and GLU are originally written, every function has a function pointer. A function can be executed by calling function-name or function pointer in C. The function name is declared by a programmer in the source code of the program, while the function pointer is given to every declared function at the run-time.

The principal behind the call-backs is that instead of calling function names, that are static at the run-time of the program, pointers that store the addresses to functions are called. The value of a function pointer can be changed dynamically during run-time.

This brings us back to GLUT. GLUT provides programmers with functions like *glutDisplayFunc*, *glutReshapeFunc*, *glutMouseFunc*, and so on. Those functions change a specific function pointer to the one send as a function argument.

Given that in a specific situation a function pointer *fp* is called, the programmer can set this pointer, with a function *setFp* to point to a desired function, *myFnc*. Now in a given situation the function *myFnc* is called. This is a call-back.

3.3.3 Main Loop

After programmer specified the set of setting that are desired for execution, the process must be started by calling function *glutMainLoop()*, this is starts traditional update/draw loop.

In the update phase of the main loop freeGLUT checks the input buffers for new values, if they are found an appropriate function call back is called, and a programmer defined function is executed. The draw phase is the last in the pipe line. *GlutIdleFunc* on the other hand will always be executed, and is used to create continues animations.

The draw call, set by the GLUT function *glutDisplayFunc*, is only executed if during the update phase a function *glutPostRedisplay* is called.

3.4 Graphical User Interface

Graphical user Interface (GUI) is strongly bound to visualisation part of a computer program, highly because GUI itself is a set of images and text visualisations. To correctly interpret users input a program needs to track what has been displayed and in what manner. This section will focus on technical view of a user interface, what are the key elements and how they communicate.

3.4.1 Window

A window is a main part of graphical interface as it is today. Most of modern graphical operating systems, like MAC OS X, Microsoft Windows, and Ubuntu, operate with windows to present data to human users. A window is a rectangular aria on the screen dedicated to a specific program. This aria can often be moved, resized, and closed by the user. A window can contain many sub-windows, but from technical point of view it is programs responsibility to maintain control of sub-windows.

OpenGL Window and mouse input coordinates

When operating system reports the position of a mouse click, the coordinates are given in the window coordinates, with the origin in the left-upper corner, while the origin of the OpenGL window is in the left-lower corner of the window. To translate window to OpenGL-window coordinates, y-coordinate needs to be reverted by a simple formula, equation 3.9.

$$\begin{aligned} x_{GL-window} &= x_{mouse} \\ y_{GL-window} &= h_{window} - y_{mouse} \end{aligned} \tag{3.9}$$

If user interface contains many viewports, or windows, then the "active" viewport is identified by checking for bounds of each view port. The the viewport's mouse coordinates can be calculated simply by taking the difference between OpenGL window coordinates and origin of current view port.

$$\mathbf{P}_{viewport} = \mathbf{P}_{GL-window} - \mathbf{O}_{viewport} \tag{3.10}$$

3.4.2 Viewport, in Terms of OpenGL

In an abstract point-of-view a viewport can be described as a window, through which we can observe the scene. In a technical perspective viewport is a part of the window where the output of drawing is displayed. This can cover the whole window dedicated to the program or it can only cover a small fraction. In most operating systems a user can change the size of the window the program has created, this does not change the size of the viewport automatically in OpenGL, however many of the developers adjust the viewport accordantly to the window size.

It is not possible to draw outside viewports bounds. A viewport is useful to divide the window in designated arias, a complicated user interface can contain several

drawing arias dedicated to a specific task, for example to display a drop down menu of imported meshes, or an overview of possible commands.

3.4.3 Mouse Input Devise

Because of the technical limitations user input from the mouse devise can only be in two dimensions. To select corresponding aria of the mesh in three dimensions, some mathematical transformation and assumptions needs to be done.

To correctly perform selection of the sub-mesh we need to understand how the objects are observed by the user. In the section 3.1.5 we have mentioned that the scene is displayed by using perspective projection, as it appears most natural for human observers. We have also discussed clip-space in the section 3.1.4. By using knowledge of those to principles we can create selection tools that can be operated intuitively by a user.

3.4.4 Selecting Object in a Scene by Bsing Ray-casting

The task of determining the sub-mesh that user selects can be restated as to determent what is visible for the user in a given aria. Since user input from the mouse is only in two dimensions we do not know the z-coordinates of the user intended input.

By using Ray-casting, a process to determent the first intersection between a given ray and the scene, we can check for intersection between the ray determined by mouse position on the screen and objects in the scene, and approximate Z-coordinates of user input. This solution works well when the task is to determent if a user has clicked “on” as specific object in the scene.

To calculate the coordinates the user intended to select we need to transform the input from viewport to world space. First we need to convert viewport coordinates to clip space. Since clips space origin lies in the centre and viewport’s origin in OpenGL is in the lower-left corner, we need to translate the mouse position by the value of half width and height of the viewport in x and y direction respectably. Then we can normalize the coordinates by dividing by half of the viewport width and height to produce values in the range of $[-1, 1]$. A total equation for trans-

formation from view port coordinates to clip space is given by

$$\begin{aligned} x_{clip-space} &= \frac{x_{viewport} - \frac{w_{viewport}}{2}}{\frac{w_{viewport}}{2}} \\ y_{clip-space} &= \frac{y_{viewport} - \frac{h_{viewport}}{2}}{\frac{h_{viewport}}{2}}. \end{aligned} \tag{3.11}$$

Where $h_{viewport}$ and $w_{viewport}$ are viewport height and width, respectably.

The rays that are used in ray-casting are determined by the projection used to create a scene. When using orthogonal projection we can imagine that the rays of light travel parallel to each other. It is not so when using a perspective projection. If we refer to figure 3.1 we can observe a perspective projection frustum with the eye, camera origin, at the top of the pyramid. All of the visible rays in a perspective projection pass through camera origin. This gives a starting point for the ray we will project into the scene.

To determent the direction of the ray we need at least two points on the same line. The first point is given, camera origin. Second point's x and y coordinates in clip-space can be calculated by equation 3.11. Given that the mouse pointer is always visible, and cannot be occluded by objects in the scene it has to have the smallest possible z-coordinate value, but jet visible. The lowest possible visible value in perspective projection is z-near plane. This gives a reasonable assumption that mouse position in camera space lies somewhere on z-plane. Using the knowledge about clip space and perspective projection, we know that the size of the view port is the size of the z-near plane. This produces this equation for mouse position in camera space,

$$\begin{aligned} x_{mouse-camera-space} &= x_{clip-space} * w_{viewport} \\ y_{mouse-camera-space} &= y_{clip-space} * h_{viewport} \\ z_{mouse-camera-space} &= z - near. \end{aligned} \tag{3.12}$$

With two points, camera origin and mouse position, we can determent the direction of the ray. Algorithm 3.1 summarises the procedure to create a ray that corresponds to user click input, used for ray-casting.

Algorithm 3.1 Algorithm to identify ray corresponding to user input

Require: user input **p and** viewport information $w_{viewport}$, $h_{viewport}$, and z_{near} **and** world-to-camera-space troansformation matrix $\mathbf{M}_{wc}$
transtorm user input to camera coordinates:$\mathbf{p}_c = [p_x w_{viewport}, p_y h_{viewport}, z_{near}]^t$
transform camera origin and $\mathbf{p}_c$ to world coordinates:
$\mathbf{o}_w = \mathbf{M}_{wc}^{-1} * [0,0,0]^t$, $\mathbf{p}_w = \mathbf{M}_{wc}^{-1} * \mathbf{p}_c$
return user selection ray in world-space coordinates: $[\mathbf{o}_w, \mathbf{p}_w]$

3.4.5 Selecting Sub-mesh by a Frustum

To describe a rectangular aria only coordinates of two appositive corners are needed. For example it can be implemented as, first corner is position when the user pressed down the left-mouse-button, and second corner is where the user realised the left-mouse-button, while holding shift-key down. We combine the ray-casting technique with this rectangular aria by creating a ray at each vertex of the selection rectangle. Since the rays in perspective projection are not parallel, those four rays create a "pyramid", which represents user selection in three dimensions.

If this selection shape is used to determent the selected objects, two problems arise. Objects that are between camera origin and z-near plane are selected but were not visible for the user in the moment of selection. Similar problem arise with infinite depth of the selection, the objects that are far away from the camera origin are selected as well. Both problems are solved by introducing two planes, minimum, and maximum bounds, creating a frustum. The minimum bound of selection is the z-near plane, as it is the closest visible value to the camera origin. It is up to the user to define value of maximum bounding plane from the camera.

Checking if the object is inside the frustum can be done by simple dot product check. In mathematics a plane can be described by equation

$$Ax + By + Cz + D = 0 \tag{3.13}$$

where A, B, C are x, y, z components of the planes normal. Or as

$$N \cdot (\mathbf{x} - \mathbf{x}_0) = 0 \tag{3.14}$$

where '$\cdot$' represents a dot product.

The four rays that are given by use selection can also be interpret as four vectors, $\mathbf{v}_1 = [x_1, y_1, z - near]^t$, $\mathbf{v}_2 = [x_2, y_1, z - near]^t$, $\mathbf{v}_3 = [x_2, y_2, z - near]^t$, $\mathbf{v}_4 = [x_1, y_2, z - near]^t$, where $\mathbf{p}_1 = [x_1, y_1]^t$ and $\mathbf{p}_2 = [x_2, y_2]^t$ are appositive corners that represent the rectangular selection. By grouping two and two vectors we can create four planes that form a square cone. As mentioned before to limit selection in camera view direction, z-near plane and maximum-bound plane are introduced. Every point with in this frustum is selected by the user, see figure 3.2 for illustration. A plane can be described by a point on the plane and a planes normal, equation 3.14. The normal of each plane can be calculated by taking the cross product between vectors representing the plane. Since the rays are originated from camera origin all of the planes contain that point. With the normal and a point on a plane we can represent planes defining the frustum in the form shown in equation 3.13.

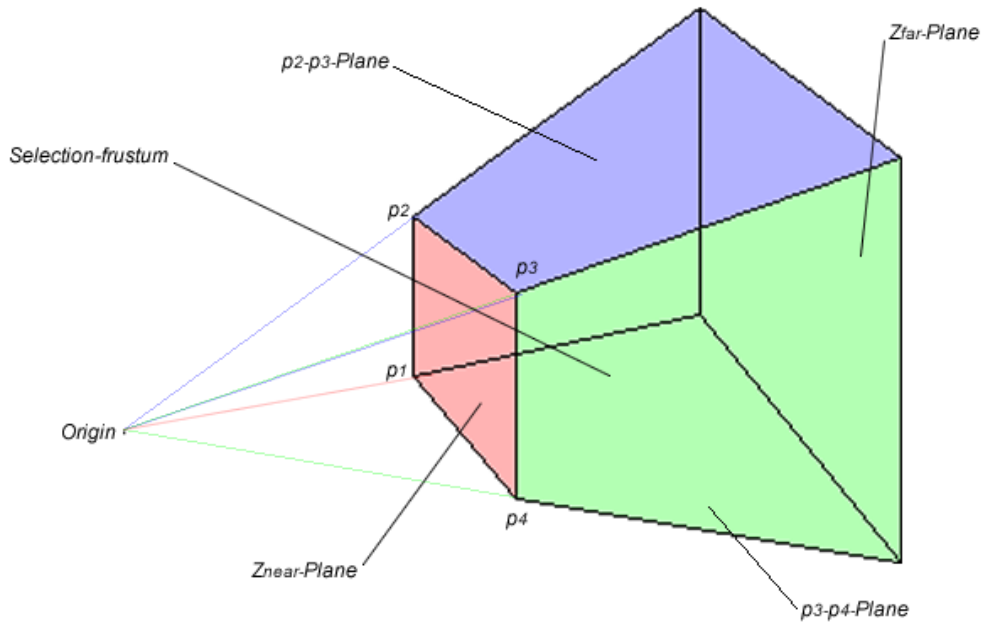

Figure 3.2: Frustum created by camera origin, four points on z-near-plane, z-near-plane and z-far-plane.

To rewrite z-near and maximum-bound plane in a form given by equation 3.14, we need to calculate at least three points on each plane. As shown in section 3.4.4, the vertices of the user selected rectangular lie in z-near plane. By using three of the vertices we can represent z-near in the same Cartesian form as the side planes of the frustum. With three points on the z-near plane we can find the maximum-bound plane, by magnifying the corresponding vectors, so the length would be as of the maximum bound. And repeat the same process we done with z-near plane.

This frustum is described in camera space coordinates. Before we can check in the object is inside or outside the frustum we need to transform frustum and model to the same space. It requires fewer calculations to transform the frustum to world coordinates, then transforming each model to camera space. A transformation from camera to world space is reverse of transformation from world to camera space. Let us call the transformation matrix from world to camera space an orientation matrix. As it is mentioned in section 3.1 that reverse transformations can be described by inverse matrix of original transformation. By multiplying

each point and vector describing the selection by inverse of orientation matrix we transform frustum to world space.

If we define vector $\mathbf{v}_d$ as a vector between $\mathbf{p}$, an arbitrary point, and a $\mathbf{p}_0$, point on a plane, we can calculate on which side of the plane a given point lies by calculating the dot product between the normal vector and a $\mathbf{v}_d$ vector. If the dot product is zero then $\mathbf{p}$ lies on the plane, giving us equation 3.14. If dot product is positive then the point $\mathbf{p}$, lies on the side of the plane on which plane's normal's direction is positive, and if the results are negative then the point lies on appositive side of the plane.

This can be used to determent if point $\mathbf{p}$ is inside a space bounded by planes. Given that planes normals, describing the space, are all pointing in the positive direction towards the centre of the space, then the point is inside the frustum if and only if the directions from all six planes to point p are all positive. The same is if normals point away from the centre of the frustum then point p is inside if and only if all of the directions between planes and point are negative. Figure 3.3 shows an illustrative example in 2D.

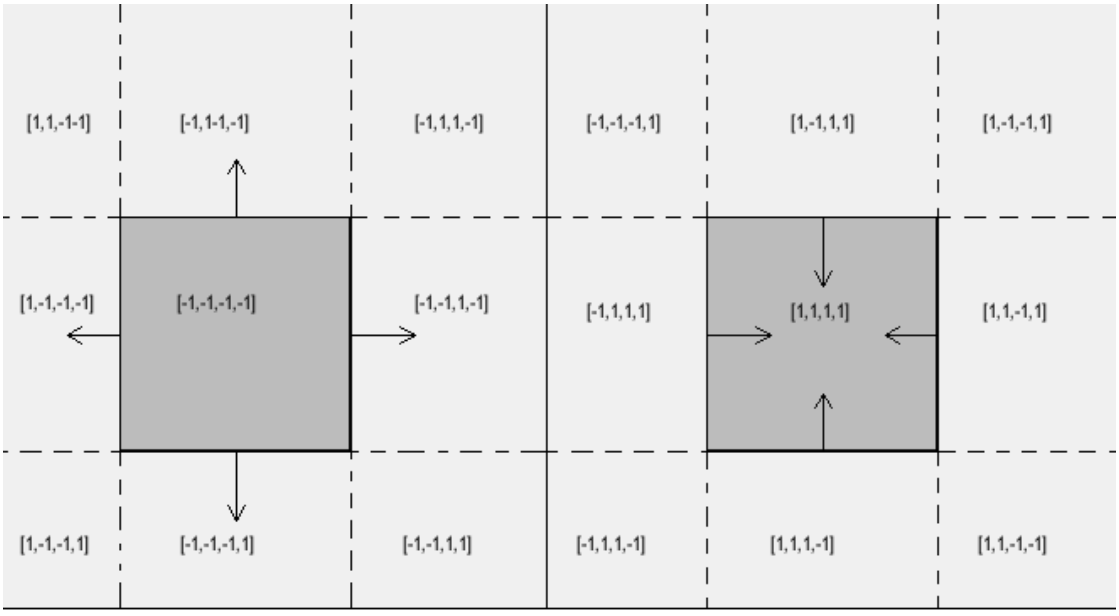

Figure 3.3: Method to determen if point is inside a region boundet by planes.

The calculation of the planes that determine the frustum has high performance

cost it, but only necessary to calculate it once per user selection. The complexity of the sub-mesh selection follow linear run time, O(N). By running prepossessing on the whole scene the number of points that are needed to be checked can be reduced greatly this can be done by using techniques like AABB, BSP trees, simple grid, and so on.

3.5 Tree Structures

3.5.1 KD-tree

A kd-tree, k-dimensions-tree is a version of a binary tree, where at each level the data is divided by two, left and right children node. Each non-leaf node can be looked upon as a hyper-plane that divides the data set in two. The direction of the hyper plane is chosen so it is perpendicular to the dimension the node is associated with. This way the data set can be simply compared based on the nodes value of one dimension at the time.

There are many ways to choose the split axis of the node in kd-tree, most frequently used is canonical method of kd-tree construction. When a creating a kd-tree in a canonical way, one cycles through all possible dimensions when selecting the splitting axis. For example in a case of 3D data in x, y, and z direction one may start at x dimension at the root, then use y axis when dividing root's children, then z axis, then return to x again, and so on in this manner.

A balanced kd-tree means that at each node, after the split by a hyper-plane there are the same amount of data at each side of the hyper-plane, this division leads to a tree that is tightly poked and have fewest possible levels. One of the ways to create a balanced kd-tree is to choose the value of the node based upon median value of the node's associated data set. Kd-tree is mainly used because of the advantage it provides when searching for nearest neighbour, because of the binary tree structure the nearest neighbour search is $O(logN)$ complexity. (Friedman et al., 1977) is a paper that proposed an algorithm for searching in kd-trees that guaranties a logarithmic complexity.

3.5.2 Oct and Quad Trees

A quad-tree is a tree structure that divides each node in exactly four children, by a given criteria, this is mostly useful when interpreting two-dimensional data. Quad trees are more efficient that simple grid division when dealing with non-uniformly distributed data. Oct-trees are tree structures where each node has exactly eight children. Oct trees are often used in 3D-parsioning in computer graphics application. In computer games oct-trees are used to simplify collision detection. Because of the high branching factor of oct and quad trees, the search for points within specific bounds can be implemented very efficient. This is use in collision detection and ray-intersection algorithms.

Chapter 4

Iterative Closest Point

Iterative Closest Point algorithm or ICP is a method to register two point-clouds, meshes, implicit or parametric surfaces. The original version of the approach was first proposed by Paul J. Besl and Niel D. McKay (Besl and McKay, 1992). The basic principle of ICP is to reduce RMS distance between point sets, by iteratively reduce individual distance between point pairs. Since 1992 the ICP has been thoroughly researched and improved. Both speed and precision has been in the focus of the research with significant results in both. In this chapter we will describe the original algorithm by Besl and McKay from 92, and some of the improvements of the method.

4.1 Original ICP

ICP is an algorithm used to register to point-clouds or meshes. The registration process of two sets can be described as finding an optimal transformation so that the error between them is minimal.

4.1.1 Distance Metric

Distance metric is fundamental feature of ICP. In original algorithm distance is used to determent the point pairs that are lately used to find optimal registration.

A distance from point p to the set A with N_a points, is given by:

$$D(\mathbf{p}, A) = \min_{i \in 1,\ldots,N_a} d(P, a_i). \tag{4.1}$$

Where distance $d(\mathbf{p}, \mathbf{p}_1)$ is Euclidean distance. Distance to a line $\mathbf{l}$ between two points then the distance between point $\mathbf{p}$ and the line segment $\mathbf{l}$ is

$$D(\mathbf{p}, l) = \min_{u+v=1} ||u\mathbf{r}_1 + v\mathbf{r}_2 - \mathbf{p}|| \tag{4.2}$$

The distance between a set of lines L with N_l lines and the point P can then be described as

$$D(\mathbf{p}, L) = \min_{i \in 1,\ldots,N_l} d(\mathbf{p}, l_i) \tag{4.3}$$

This brings us to the last closed form distance proposed by Besl and McKay. The distance between a point and a triangle set. Let t be the triangle defined by three points the distance between the point and the triangle t is

$$D(\mathbf{p}, t) = \min_{u+v+w=1} ||u\mathbf{r}_1 + v\mathbf{r}_2 + w\mathbf{r}_3 - \mathbf{p}||. \tag{4.4}$$

Where $u \in [0, 1]$, $v \in [0, 1]$, and $w \in [0, 1]$. The closed form distance to the set T of N_t triangles is given by

$$D(\mathbf{p}, T) = \min_{i \in 1,\ldots,N_t} d(\mathbf{p}, T_i). \tag{4.5}$$

The distance to parametric and implicit surfaces can be found by firstly compute a triangular set and then follow procedure to calculate the distance to a triangle set.

4.1.2 Optimal Registration

As it mentioned above the goal of ICP is to find an optimal registration between two data sets. As it described in the original paper (Besl and McKay, 1992).

> *The unit quaternion is a four vector $\boldsymbol{q}_R = [q_0 q_1 q_2 q_3]^t$, where $q_0 \geq 0$, and $q_1^2 + q_2^2 + q_3^2 = 1$. ... Let $\boldsymbol{q}_T = [q_4 q_5 q_6]^t$ be translation vector. The complete registration state vector $\boldsymbol{q}$ is denoted $\boldsymbol{q} = [q_r | q_t]^t$. Let $P = p_i$ be measured data point set to be aligned with a model point set $X = x_i$, where $N_x = X_p$ and where each point p_i corresponds to the point x_i with the same index. The mean square objective function to be minimizes is*

$$F(q) = \frac{1}{N_p} \sum_{i=1}^{N_p} ||x_i - R(\boldsymbol{q}_R)p_i - \boldsymbol{q}_T||^2$$

This shows that registration is a combination of rotation and translation of the data set so that Euclidian distance between, model and data is minimal.

The method proposed by Besl and McKay uses singular value decomposition (SVD) based on covariance matrix, to find optimal rotation and translation. The detailed information is found in the same paper.

4.1.3 ICP Algorithm

ICP algorithm estimates an optimal registration of a data point-cloud P in respect to a given model X. If data is represented in a form other then point-cloud, then it must be converted to so. In the case of simplex-based representations, as line segments or triangles, the decomposition to a point-cloud is simple: vertexes that are used to denote edges of primitives are points in a converted point-cloud. In the case of implicit or parametric surfaces then the vertices of line or triangular approximation are used. Algorithm 4.1 describes a local registration procedure.

Algorithm 4.1 Iterative closest point by Besl and McKay, 1992.

Require: point set P with N_p points **and** shape X
  $P_0 = P$
  $\mathbf{q}_0 = [1, 0, 0, 0, 0, 0, 0]^t$
  $k = 0$
  while $\tau > 0$ **or** $d_k - d_{k+1} < \tau$ **do**
    Compute closest points: $Y_k = C(P_k, X)$
    Compute the registration: $(\mathbf{q}_k, d_k) = Q(P_0, Y_k)$
    Apply the registration $P_{k+1} = \mathbf{q}_k(P_0)$
  end while
  return Q_k

4.1.4 Local vs. Global Registration

The convergence theorem for ICP algorithm is formulated by Besl and McKay as

> *The iterative closest point algorithm always converges monotonically to a local minimum with respect to the mean-square distance objective function.*

For the proof see (Besl and McKay, 1992).

The ICP convergence theorem only guaranties convergence to a local minimum, resulting in a non-optimal orientation. One of the approaches to reduce the chance of algorithm getting stuck in a local minimum is to restart the ICP with random orientation and chose the one orientation that result in a lowest error value. This is requires a vast amount of calculations. It was shown in the paper (Besl and McKay, 1992) that the initial registration state that will result in convergence to a global minima will result in a faster rate of convergence that lead to a local minima. This gives possibility, on a given set of random initial registrations to run Algorithm just a given number of iteration, and only proceed to full convergence on the registration set that resulted in the lowest error value.

4.2 ICP Improvements

In this section we use paper by Szymon Rusinkiewicz Marc Levoy (Rusinkiewicz and Levoy, 2001) in which they compare different versions of ICP algorithm in a respect to run time registration procedure. Some interesting result were represented in there's paper, showing both speed and accuracy of some of the variants of the ICP algorithm on different types of scenes.

The changes to original algorithm can be categorised in four groups.

Point Selection

Point selection is a first stage of ICP loop, where a set of points to operate on is selected. It can result in points from one or both meshes. The variations include, using all points (Besl and McKay, 1992), use random selected points at each iteration (Masuda et al., 1996), and use points with high intensity gradient (Weik, 1997).

Matching of Point Pairs

In this stage for each point in the determent point set the corresponding point in appositive shape is found. The original approach is to find closest point. Normal shooting variant is when the correspondent point is found by intersection of the ray in the direction of the normal, (Chen and Medioni, 1991). Reverse calibration or projection method is based on projecting the point onto the destination mesh, (Blais and Levine, 1995). Latest method has also the alternative to search

the neighbourhood for a better match, based on the colour and topology information. Some papers representing this approach are: (Benjemaa and Schmitt, 1997; Weik, 1997; Pulli et al., 1997)

Weighting

When creating the covariance matrix in the process of computing the registration, each point pair can be weighted differently. The weights can be constant, based on the point-to-point distance, based on the dot product between normals of the point pair, (Godin et al., 1994), and it can be based on expected scanner noise value.

Error Metric

Error metric describes what is considered optimal registration. In simple form it can be sum of squared distances. It can contain core information as colour value of point-pairs (Johnson and Kang, 1996). The other possibility is to take a distance to the plane that is perpendicular to the destinations normal (Chen and Medioni, 1991)

Chapter 5

MeshCombine

In this chapter we propose an approach to merge two triangular meshes of different resolutions, originated from different scanners, to produce a combined mesh with regions of different resolutions.

MeshCombine is a simple prototype tool to merge meshes of different resolution and origin. The idea behind *MeshCombine* is to replace a region in the old surface mesh with representation of high resolution. The Icp algorithm can provide us with optimal registration, meaning that we will move data mesh to the corresponding position in the model mesh. This registration will result in the two meshes that will ether completely or partially overlap each other. Let say the region X' is the region of the model surface X represented by the data mesh P. Given that the mesh P is captured with higher precision that region X, the region X' contains no additional information to the combined mesh and there by can be safely removed. The task of mesh manipulation can be identified as follows: firstly, identify region X based on registered data mesh P, secondly, remove region X from model mesh, thirdly, add the registered data to the model mesh, and finally, re-triangulate the hole between model and data part in the merged model.

5.1 Identifying the Region

Identifying the region X in model that corresponds to the data mesh P, is an important task, as the quality of the combined model is dependent on precision of this operation. If the marked region is too big, then when deleting it from the old model we lose data, if the region is too small, we corrupt additional information

we obtain from the data set. We propose three solutions formulated as:

1. The vertex $\mathbf{x}$ of the model set X is in the region X' if there is a vertex $\mathbf{p}$ in the set P so that $d(\mathbf{x}, \mathbf{p}) < limit$.
2. The vertex $\mathbf{x}$ of the model set X is in the region X' if there is a triangle t in the set P so that $d(\mathbf{x}, t) < limit$.
3. The vertex $\mathbf{x}$ of the model set X is in the region X' if there is a triangle t in the set P so that normal distance $d_{normal}(\mathbf{x}, t) < limit$.

5.1.1 Spherical Distance

The solution 1 describes a point to point distance metric. With other words it can be described as every point that lays within in a given radius of any point in the data set is belonging to the region X'. From the abstract point-of-view this scenario can be visualised as every point in the model set X is also in the set X' if and only if there is a point within a sphere around the point with a given radius. Solution 2 follows the same pattern, the difference is that if any point belonging to the triangle is with is previously describes sphere, then vertex $\mathbf{x} \in X'$

5.1.2 Directional Distance

Directional distance prioritise in one direction more than in the other. Back to a three dimensional example, spherical distance is unidirectional as a sphere has the same distance from the centre to outer bounds, while selecting point by an ellipsoid shape is directional, as an ellipsoid can be describes as a sphere expanded in one direction.

In the case of identifying the duplicated surface that is generated by scanning a real life object, we propose to use a directional distance in the direction of the surface normal. Given the knowledge about the scan acquisition we can assume that the noise has greater variation in the direction of the surface's normal.

5.1.3 Directional Distance in Triangular Meshes

It can be said, that a vertex $\mathbf{x}$ in a model is represented in the data shape if there is a data vertex $\mathbf{p}$ that obscures it from the camera, same idea is represented

in (Pito, 1996). In our case we do not have information about orientation of the scanner and there by assume, that, the surface is captured by the scanner from a ninety-degree angle. The normal of the vertex then can approximate the viewing direction. We rephrase the definition of region X' as: a vertex $\mathbf{x} \in X'$ if there is a point $\mathbf{p}_i \in$ shape P, that lays in a line described by point $\mathbf{x}$ and normal N_x, within a given distance. In a case of triangular meshes shape P is represented by triangular set T, and the point p_i is a point on a triangle $t \in T$. Any point p on a triangle $t = [\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3]$ can be described as

$$\mathbf{p} = u\overline{p_2p_1} + v\overline{p_2p_3}, \tag{5.1}$$

where $u, v \in [0, 1]$. The plane R that is parallel to the triangle can be written as

$$R = s\overline{p_2p_1} + t\overline{p_2p_3} + \mathbf{p}_2. \tag{5.2}$$

Let vector N_p be the normal vector associated with point $\mathbf{p}$, a straight line L that is parallel to the $\mathbf{n}_p$ and goes through point $\mathbf{p}$ can be represented by parametric equation

$$L = k\mathbf{n}_p + \mathbf{p}. \tag{5.3}$$

Let $\mathbf{p}_i$ be the point of interception between line L and the plane R. The distance d from point $\mathbf{p}$ to point $\mathbf{p}_i$ is given by absolute value of the vector $\overline{pp_i}$

$$d = |\overline{pp_i}| = |\mathbf{p}_i - \mathbf{p}|. \tag{5.4}$$

Combining equation 5.4 and 5.3 gives

$$d = |k\mathbf{n}_p + \mathbf{p} - \mathbf{p}| = |k\mathbf{n}_p| = k|\mathbf{n}_p|. \tag{5.5}$$

Since point $\mathbf{p}_i$ can be represented by equations 5.3 and 5.2, combining those equations will result in

$$s\overline{p_2p_1} + t\overline{p_2p_3} + p_2 = k\mathbf{n}_d + \mathbf{p} \tag{5.6}$$

Solving the equation 5.6 for k gives

$$k = \frac{s\overline{p_2p_1} + t\overline{p_2p_3} + \mathbf{p}_2 - \mathbf{p}}{\mathbf{n}_p}. \tag{5.7}$$

Since the vectors that described the triangle t in equation 5.1 are used to describe the intersecting plane, we can check if the point $\mathbf{p}_i$ is within the triangle t, by

simply looking on the values s, and t. If values $s, t \in [0, 1]$ then the point $\mathbf{p}_i$ lays in the triangle.

$$d_{normal}(\mathbf{p}, t) = \begin{cases} \frac{s\overline{p_2p_1} + t\overline{p_2p_3} + \mathbf{p}_2 - \mathbf{p}}{\mathbf{n}_p} |\mathbf{n}_p| & \text{if } s, t \in [0, 1] \\ \infty \quad \text{else} \end{cases} \tag{5.8}$$

Solution 3 can be interpreted as: the vertex $\mathbf{p}$ in the set X is in the region X' if there are a point $\mathbf{p}_i$ on the data surface P that lies on the line described by point $\mathbf{p}$ and $\mathbf{n}_p$, within a given distance.

5.2 Deleting Duplicated Region

When the duplicated region X is identified it can be removed from model mesh. The practical question arise, since the triangle information is often stored as triplet of indexes in which vertices are stored in, when deleting points the triangle information is no longer correct as order of vertices has changes due the deletion. To preserve triangular information the order of vertices must be preserved, or indexing information changed accordingly due deletion operation.

Keeping the deleted vertices and by that preserve indexing information will result in un-packed data set, with some un-used garbage values. Repeating the merging operation will then result even more garbage, putting some un-needed workload on a system. Updating the index information will result in better compressed mesh, with no garbage values, but brings more complexity to the implementation. More on re-indexing of triangular data can be found in section 8.3.2.

5.3 Data Merging

Merging of data raises the same problem with triangular information as deletion. Ether the mesh is represented by two separate vertex and index arrays, or they are combining into one. If the mesh-structure contains two vertex arrays it is difficult and impractical to have an index array that will contain information about triangles that stretch between vertices from one vertex array to another. Merging the arrays improves reading speed and simplifies the displaying and saving to disk process.

5.4 Triangulation

Surface triangulation is a process of dividing a given shape into triangular regions that approximate shape's surface. Triangulation is often used in computer graphics as triangle rasterization is heavily optimised in graphics hardware. When working with scanned data the output is often a point-cloud in 3D. Point-clouds do not give a good approximation of the surfaces, due the discrete nature of a cloud. When representing data visually the point-clouds are often triangulated in advance. Triangulation is a complex task, which has been researched for decades. In the resent years due the advance in scanning technologies point-cloud triangulation was in the spot light. Works like (Jian-Ming et al., 1990; Golias and Dutton, 1997; Qi et al., 2013), are just a fraction of the research that have been made in the field.

5.4.1 Greedy Triangulation

An example of greedy triangulation can be found in the paper (Marton et al., 2009). This triangulation algorithm was designed to aid robot navigation in real-time and there for produce a fast triangular mesh approximation to a surface.

To create a triangle set that approximates the underlying surface represented by the vertices, a surface must be estimated in advance. This is usually done by calculating a plane D that approximate the surface and is tangent to the current vertex in triangulation process. The previously mentioned paper solves this problem by creating a weighed least square plane, the normal of which approximates the true surface normal. This normal estimation is done by searching for k-nearest-neighbours in the proximity of the current vertex within a given distance, creating a weighted covariance matrix $\mathbf{C}$, from the points p_i in the neighbourhood

$$\mathbf{C} = \sum_{i=1}^{k} \xi i * (pi - p)^t * (pi - p), p = \frac{1}{k} * \sum_{i=1}^{k} pi, \tag{5.9}$$

ξ_i represents the weight for the point p to neighbour $p_i : \xi_i = exp(\frac{-d_i^2}{\mu^2})$, where μ is the mean distance from point p to all its neighbours p_i, and d_i is the distance from point p to a neighbour p_i. The process is followed by the eigenvector V and eigenvalue λ computation $C \cdot V = \lambda \cdot V$.

This weighting of neighbouring points gives better resistance to noise as it increases the contribution of points in close proximity, making normal variation smoother over the approximated surface.

With the help of the plane D that approximates the surface in a given neighbourhood a 3D surface triangulation problem can be transformed into 2D problem. This complexity reduction is achieved by projecting points to the plane D.

When all points in the neighbourhood are on the same plane 2D triangulation can be started. 2D triangulation is a well researched subject.

Delaunay triangulation a process of triangulating a set of points P in a plane so that there is no point p_i that is inside a triangle. This triangulation process is named after Boris Delaunay for his work in this topic from 1934.

5.4.2 Greedy Triangulation for MeshCombine

Open source project PCL (PCL, 2013), has a library that implements greedy triangulation. Unfortunately the library is written in C++ and operates on much more complicated classes/structures than simple arrays used in this project. This triangulation library is implementing a paper by Zoltan Csaba Marton, Radu Bogdan Rusu, and Michael Beetz (Marton et al., 2009). We based your implementation on the same paper.

Using neighbouring weighted points to calculate a plane that approximates the surface normal is computationally expensive, and since normal information about vertices is already available, we propose a normal weighting; to calculate weighted normal of the current point in triangulation process, based on the same approach as in (Marton et al., 2009), the contribution of the neighbouring vertices decreases as distance from current vertex increases. The weighting equation is presented in section 10.3.1

In (Marton et al., 2009) the authors propose to reseed the process form other random vertex that is not jet triangulated, when triangulation process is stuck. Our case can be interpreted as incomplete triangulation, as we have two triangulated parts of the mesh, model and data, and we have an un-triangulated part, the gap between model and data, created by the deletion of duplicated region. The process can be reseeded by using the boundary vertices that are defined as points that are on the edge of the gap. This triangulation technique does not alter the existing mesh and there by preserving the given quality of both data and model meshes. The only region where new error is introduced is the gap between the meshes.

Part III

Implementation

Chapter 6

Graphical User Interface

MeshCombine requires a Graphical User Interface (GUI) mainly because of two reasons, human input to simplify registration, and output of combined meshes to verify the positive results. GUI is useful to a user as it provides initiative controls and simplifies communication between human and the machine. This chapter will go through the implementation of *MeshCombine*'s graphical user interface with OpenGL and freeGLUT.

6.1 Graphics API

MeshCombine is designed to work on meshes obtained from scanning real world objects. Those meshes can contain hundred thousands of polygons. To draw a triangle meshes this size an immense amount of calculations is requires. With the support of modern graphics cards rendering of large models can be done in real time. DirectX and OpenGL are two graphics API that are used to interact with graphics dedicated hardware. The major drawback of DirectX is that it is for windows OS only. While OpenGL libraries are harder to access on Windows, as Microsoft stopped supporting OpenGL in 2003, OpenGL is still a cross platform API and works as good on Windows as on any other OS. This is the major factor in the choice of graphics API.

The *MeshCombine* uses OpenGL to render graphics on the screen, and freeGLUT to initiate OpenGL contents and create a window. FreeGLUT provides main functionality as input/output listeners, draw/update loop, and double buffer rendering. In section 3.3 freeGLUT is described in more details. The use of C programming

language, OpenGL, and freeGLUT gives possibility to run on multiple platforms, using only open source specifications and libraries.

6.2 Viewport

Since *MeshCombine* is a simple prototype we choose to use only one section where the meshes are displayed, it can be called mesh-viewport and is always the size of the current window. At the current moment the interface lack visible clues and functionality, as menus where the adjustments to the algorithms parameters are possible. The functionality is limited to orientation of the camera as, rotation and translation, selection of the sub-mesh, and start of the combining algorithm. Detailed algorithm about input functions can be found in section 3.4.4 and 3.4.5.

6.3 GL Programs

OpenGL pipeline is easy modifiable through the use of vertex and fragment shaders, see section 3.2.4. In *MeshCombine* project two vertex shaders are implemented *GouraudVertShader* and *toClipSpace*.

ToClipSpace shader is a simple shader to draw in 2D. Though this shader can draw any complex figure in 2D, in this program it is used to draw the rectangular that marks the rectangular selection for sub-mesh. For an input toClipSpace takes 2D coordinates of a vertex that are already in clip-space, $\in [-1, 1]$, and sets z-value to 0, this ensures that the figure gets lowers possible z-value in the z-buffer, section 3.1.6, and the figure is always drawn.

GouraudVertShader is a 3D shader implementing Gouraud shading (Gouraud, 1971). Gouraud shading uses normal of each vertex in a triangle to calculate the colour value at each vertex, rest of the triangle is coloured by linear interpolation between values at vertices. In contrast to *toClipSpace* shader *gouraudVertShader* takes in three parameters per vertex and six uniform values. Per vertex values are position colour and normal. The uniform parameters are constant due the draw call and contain information about lighting, orientation of the "camera" and perspective-transform-matrix. The outputs of *GouraudVertShader* are position of vertex in clip-space and colour of the vertex based on, original colour, normal, and direction to the light.

FragmentShader is a simple fragment shader that is used by both *toClipSpace* and

gouraudVertShader, as it simply redirects the input as output, without altering it.

Out of this three shaders two programs are linked, *gouraudShading* and *simpleProgram*. *GouraudShading*, which contains *gouraudVertShader* and *fragmentShader*, is used to draw every mesh displayed by the program, while *simpleProgram*, *toClipSpace* and *fragmentShader*, is used to draw the selection rectangle.

6.4 Mouse and Keboard Functions

Scene orientation

The user can orient in the scene by using a mouse, the rotation of the scene is archived by user holding down the left-mouse-button and moving the mouse, while to apply translation the user needs to hold down the right-mouse-button. Finally the scene can be "moved" closer and father from the viewing-point to give more intuitive control to the user, by using the mouse-wheel.

Sub-mesh Selection

The shift key is used to create a sub-mesh selection, the user presses down shift then clicks on the screen to set a first vertex of the selection, then while holding down the button and the key drags the mouse to the desired location. From the moment when left mouse button is clicked a rectangular, representing the selection is drawn to provide a feedback to the user. As the mouse pointer moves so does appositive corner to the start vertex moves to update the size and shape of the selection. This form of visual feedback is frequently used in software and can be considered as intuitive.

Starting Mesh Merging Algorithm

To start the procedure to merge model and data mesh is done by pressing the right-arrow-button on the keyboard. This initiates the procedure described in chapter 8.

Chapter 7

Mesh formats

There are many formats in which you can store triangular meshes on disk. Some formats save more information about the mesh, some go for less storage space, and others go for readability. In this chapter we introduce two formats, which the MeshCombine software supports, wavefront .obj and Stanford .ply formats. Later we present the way meshes are stored and interpreted in *MeshCombine* in run-time. We present function implemented to draw, update, and create meshes in *MeshCombine*.

7.1 Wavefront, .obj

OBJ format was developed by Wavefront Technologies for their Advanced Visualizer animation package in late 80s. OBJ format was adopted by many 3D graphics vendors and is now highly recognised format.

One of the reasons the OBJ format is so popular is because of its simplicity. Is only supports ASCII encoding, this adds readability as an OBJ file can be read as simple text. The format itself only supports geometry and texture-coordinates, additional information is stored in MTL format files, where texture, material, and lighting information is stored.

In OBJ format every line begins with a key word, for example vertices coordinates in a homogenous 3D space would be written as "v 1.001 42.0 -0.12 1.0" following $v = [x, y, z, w]$, w is optional. A comment in the OBJ file will begin with '#' sign. Other attributes follow the same principle.

```
# this is an example OBJ file
v -1.0 -1.0 0.0
v 1.0 -1.0 0.0
v 0.0 2.0 0.0
vn -0.7 -0.5 -0.8
vn 0.4 0.3 0.4
f 1//1 2//1 3//2
```

Figure 7.1: Example of an OBJ file representing a triangle with no textures and 2 different normal

Since OBJ format allows reusing vertexes information, the faces have a special syntax. Every face element starts with key word 'f' then follows by three or more groups of characters divided by space. Each group follows specific syntax. A group can consist of one, two or three values. First integer represent the index of a vertex, second index of a texture coordinates and third index represent a normal. Each value is ether ended with space, to indicate end of group, of a slash separating the values. A special case is when texture coordinates are skipped, this is simply done by writing two slashes after first value. All indexes are defined by the order which they were written in the file. An example of a simple OBJ file representing a triangle with no textures and 2 different normal is can bee seen in figure 7.1.

Because of simple readability and simple setup this format is very popular. It is wildly used in computer games, and visualization applications. Some of the draw backs of OBJ are speed and space. Because of no addition information about how many elements of each type will be described in a file, when reading an OBJ file, ether dynamic allocation of arrays must be done or two pass reading. Choice of ASCII also slows down as the processing unit must convert from binary value, to ASCII, and then to float or integer. Using ASCII to store data also increases size of the written file.

7.2 Polygon File Format, .ply

Polygon File Format is wildly spread in computer visualization. It was developed by Stanford University in mid 90s and is supported by most of today's 3D modeling software. The format is inspired by Wavefronts OBJ format but offers more freedom.

A PLY file consists of a header and a body part. A header part is always written

in ASCII letters and start with a magic number ply. It contains information about the rest of the file, the body part. This is a slight improvement over OBJ format in the terms that after parsing a header the size of the body is known and appropriate block of memory can be allocated. The header also specifies in what format the rest of the body is written, ASCII, big-endian, and little-endian. The ability to write and read files written in binary format is a great improvement to the ASCII format, as binary data takes less memory and is faster to read/write.

The body of a PLY file is a big block of data that stored in a specific way described in the header of the file.

An over view of the format can be found at the "The Stanford 3D Scanning Repository" (StanfordUniversity, 2012).

There are many open source libraries to read and write .ply files. In this project we use library written by Diego Nehab (Nehab).

7.3 GeometryMesh

Geometry mesh is a data structure written in C to group together geometry information of a 3D mesh. It contains pointers to the start of the arrays, and integers for vertex and element count.

VertexArray is a floating point array representing vertices' coordinates. Each vertex is written in a format $\mathbf{v} = [x, y, z]$. The array is tightly packed, meaning that after z-coordinate of the first vector follows x-coordinate of second vector. This gives 3 to 1 relationship between the size of an array and number of vertices.

NormalsArray contain normal information about each vertex, the first vertex in vertexArray has first normal in normalsArray, and second vertex has second normal and so on forth. This order is required of OpenGL to draw the object by index list. Because of this particular order the size of normalsArray is same as of vertexArray, three times number of vertices.

ElementArray is a list of triangles written in the format triangle $t = [p1, p2, p3]$ where $p1$, $p2$, and $p3$ are indexes in vertexArray, representing vertex with position given by the unsigned integer $p1$, $p2$, and $p3$. This array is also tightly packed. *VertexCount* is length of the VertexArray.

ElementCount is length of the ElementArray

7.4 MeshObject

MeshObject is a data structure to contain all information needed to draw a mesh on the screen.

7.4.1 Buffers and Arrays

If for every draw call the software needs to send data to the GPU through the buss between CPU and GPU, then this transfer is easily becomes a bottleneck for the performance. For a complex polygon mesh to be drawn efficiently on the screen in real time, the required information needs stored in GPUs local memory, before the draw call is called. This functionality is hardware based, but is luckily encapsulated by OpenGL. For this purpose OpenGL uses buffers.

Buffers are generated by function call $glGenBuffers$. This receives the number of buffers to generate and a pointer to the return value, and returns buffer id/ids as an unsigned integer. The user can fill the buffer with arbitrary data, by using *glBufferData* function.

Every *meshObject* has four buffers: *vertexBuffer*, *normalsBuffer*, *colorBuffer*, and *elementBuffer*. The vertex buffer stores vertices' coordinates information, normal buffer normal of each vertex and colour buffer stores colour per vertex, while element buffer stores triangle information.

Mesh object also stores mesh information in arrays, this way it is possible to extract and edit data without contacting GPUs memory.

7.4.2 VOA

There is one Vertex Object Array per *meshObject*. It describes how the OpenGL will draw an object on the screen, what buffers to use and how to interpret the data. Data that has been extracted from the buffers is used by user-defined GLSL program. Because OpenGL Shading Langue provides software developer with great amount of freedom, the naming and positioning of input attributes can differ from program to program. This makes VOA highly dependent on implementation of OpenGL program that is used to render objects.

7.4.3 Creation

The creation of a *meshObject* is simple; a string containing file name/address and a pointer to a GLSL program are send as an attribute to the function *createMeshObject* in *meshObject.h*. The function follows this procedure

- Interpret if a file name has an .obj or .ply ending and uses correct library to load vertex and element array.
- Based on triangle information the normals are calculated for each vertex and stored in *normalsArray*.
- Colour array is created and filled with define colour absorption factor.
- The buffers are then created and filled with appropriate data from the arrays
- VOA is generated and initiated based on predetermined syntax of shaders
- Program pointer is the stored within a *meshObject* structure

7.4.4 Drawing

With all elements in place a *meshObject* can be drawn on a screen with one call *drawMeshObject()* that takes a pointer to the object as an argument.

7.4.5 Updating

Since the *meshObject* not only stores id of the buffers on GPU where the vertex data is saved but also stores the original arrays, updating a *meshObject* is quite simple. The procedure to updating an object is as follows:

- Update the information in vertex, element, colour or normal array
- When the update is complete call the *updateBuffers* function

UpdateBuffers function simply re loads all of the information from the arrays, CPU memory, to the buffers, GPU. This is not optimal way to perform an update, since for example if only a small part of vertex array was updated, the way *updateBuffers* currently implemented, the OpenGL will discard all of the information in the buffer, and reload all of the information contained in vertex array to the buffer, even is only a small portion of the vertex array is changed.

Chapter 8

Mesh Combine Implementation

Mesh combination implementation can be divided into three major files: *meshCombine*, *greedyTriangulation*, and *icp*. *meshCombine.c* contain function a *MeshCombine*, which merges data mesh with model mesh using *greedyTriangulation.c/h*, and aligns data mesh to user selected sub-mesh of model shape by running ICP. This chapter will go through the implementation of the *meshCombine.c/h* file.

The responsibilities of *meshCombine* file are,

1. run the ICP algorithm on user selected potion of the model mesh
2. apply registration matrix returned by the ICP algorithm
3. estimate and delete the region that is represented by both data and model mesh
4. identify the boundary around deleted region
5. initiate greedy triangulation algorithm with boundary points from model as initiate start points
6. return newGeometryMesh received from greedy triangulation algorithm

8.1 Running ICP

Both running the ICP algorithm and applying registration matrix is done in function *aligneMeshes*. The minimum and maximum bounds of selected sub mesh are

found, and all the required information is send to *icp.h*'s function *globalICPRegistration*, the information about ICP implementation is found in chapter 9.

8.2 Region Estimation

Region estimation is the most technical part of *meshCombine* file. To estimate a region in a model represented by the data shape, two alternatives were implemented, radius to triangle distance, and distance in normal direction. Which method the algorithm will use is predefined by USE_RADIUS constant. By setting this constant to 0 the program will use directional distance in normal direction, if USE_RADIUS is set to 1, spherical distance will be used.

To determent if vertex x is in the region X' that is represented by the data shape, we use the formulas described in section 5.1. The complexity of this kind of matching is $O(n^2)$ in worst case run, to reduce this complexity we created an oct-tree representation of the model. Oct-trees can speed up matching process as only mesh information within oct-tree nodes, that are with is a specific distance, have to be considered.

More information about use and creation of oct-tree can be found in appendix B

Identification of region X' is summarized in algorithm 8.1 and implemented in function *getDuplicatedVertexList*. Since the process is implemented linear and not parallel, the resulting index list of vertices that are in region X', will be ordered from smallest to greatest index. This order is a product of the implementation. When program iterates through the vertex array V_m of a model shape with k_m elements and finds that vertex v_i is in the region X' it saves the index, $i \in [0, n]$ in a index array E. Let n, m be indexes in the array E. Because the iteration of the array V_m is linear, for each loop we increment index by one, we are guaranteed that if $n < m \iff e_n < e_m$. We take advantage of this order later when we are deleting vertices that are in region X'.

Algorithm 8.1 Algorithm for detecting duplicated region

Require: shape M with vertex set V_m of k_m vertices and normal set N_m with k_m normals **and** shape D with vertex set V_d of k_{dv} vertices and triangle set T_d with k_{dt} triangles **and** mimimum distance d_{min}

```
X' = ∅
create oct-Tree Ot representing D: Ot = OctTree(V_d, k_dv, T_d, k_dt)
for all v_i ∈ Vm do
  find oct-tree leaf nodes that are within a minimum distance: S = C(Ot, v_i)
  for all s_j ∈ S do
    check if there are tringle with in s_j
    if USE_RADIUS then
      b = vertexInRadius(s_j, v_i)
    else
      b = rayIntersectsData(s_j, v_i, n_i)
    end if
    if b then
      break
    end if
  end for
  if b then
    add v_i to region X'
  end if
end for
return X'
```

8.3 Deletion of Duplicated Region

8.3.1 Deletion of Vertices

In previous section it was shown that by following algorithm 8.1 the index array that contains indexes to the vertices that a found to be in the region X', is ordered from low to high. This opens for the use of standards library function memcpy, appendix E. The algorithm 8.2 describes the procedure of rewriting the vertex array. The function $restructureVertexArray$ in $meshCombine.c$ implements the algorithm.

Algorithm 8.2 Algorithm for detecting vertices from vertex array

Require: vertex set V of n vertices **and** ordered index list X with m elements
  $s = 0$;
  $k = n - m$
  for all $x_i \in X$ **do**
    copy elements from s to x_i : $memcpy(V_{new}, v_s, x_i)$
    $s = x_i + 1$
  end for
  return $[V_{new}, k]$

8.3.2 Triangle Information Update and Identification of Duplicated Region Boundary

When deleting vertices from a model one must also update the triangle information to avoid future errors, see section 5.2. The triangle update can be divided into two tasks, deleting triangles that are not complete because of vertex deletion, and update index value of vertices of remaining triangles.

The return of binary search is either -1 or position in which the same values as p_n was found. For our means we have altered the implementation of binary search so it returns a duplet, a Boolean and a position. The Boolean contains information of the item is present in the set, and the position tells how many elements have values less than the queried item.

Our implementation solves both tasks of updating the triangle information, and identifies the boundary of the region X' in one pass through element array. The binary search can only operate on ordered list, from section 8.2 we know that

the list of deleted vertices V is ordered from low to high values. The process of updating triangle information and creating triangle-to-delete index list X_t is given by algorithm 8.3. The removal of deleted triangles from element array is archived by following the algorithm 8.2

Algorithm 8.3 Algorithm for updating triangle information and region boundary detection

---

Require: Triangle set T of n triangles **and** ordered index list X with m elements

for all $t_i \in T$ **do**
  for all $p_j \in t_i$ **and** $j \in [1, 3]$ **do**
    search in list X for value p_j: $[d_j, s_j] = S(X, p_j)$
    update index of the point previosly indexed as p_j: $p_j = p_j - s_j$
  end for
  if any of points p_j are found in delete list X
  the triangle t_i is deleted.
  if d_1 **or** d_2 **or** d_3 **then**
    add i to X_t
    for $k \in [1, 3]$ **do**
      if any of the vertices of the deleted triangle are not to be deleted,
      then the remaning vertices are part of boundry B
      if $!d_k$ **then**
        add p_k to B
      end if
    end for
  end if
end for
return $[X_t, B]$

For each triangle $t = [p_1, p_2, p_3]$ in element array we search for each vertex p_n in V by using a binary search. The triangle is deleted if any of the triangle's vertices are in vertex list V. If one or two vertices in a triangle are deleted but not all, the remaining vertices are belonging to a border between region X' and rest of the model mesh.

If a triangle t is not deleted the index information of the triangle must be updated. To update the triangle information we must reduce the index value p_n by the number of deleted vertices with the index lesser than n. The position k, values returned by binary search, is exactly the amount of vertices that have been deleted with the index value less than n. If the vertex p_n is not deleted than in a new

vertex array the index of the same vertex v_m is given by

$$m = n - k$$

8.4 Running Greedy Triangulation

Greedy triangulation implementation is explained in chapter 10.

The identification of the boundary C between deleted region X' and remaining model mesh is introduced in the previous section. This boundary is used to start the triangulation process and used to grow the mesh between data and model mesh.

Chapter 9

ICP Implementation

The implementation of the iterative closest point algorithm can be found in the *icp.c/h* files. This chapter will go through stages of the implementation, and describe the work flow, data, and functions. The algorithm is divided in two main parts local and global registration. The global part is mainly a loop where a given number of local registration procedures are run and the best result is stored. The *localICPregistration* function is an implementation of an ICP algorithm, where after given amount of iteration or if given threshold is met the algorithm produces the registration matrix and error metric value.

9.1 Global Registration

As mentioned in section 4.1.4 the ICP algorithm has the possibility to get stuck in local minima resulting in a non optimal registration, to avoid this unsuitable situation, an easy solution is to restart the local ICP with different starting locations and save the best result as optimal registration. The function *globalICPregistration* does exactly that.

The function takes as arguments model and data meshes, their size, maximum and minimum relocation bounds, and maximum number of local registration restarts. The purpose of *globalICPregistration* function can be divided in three tasks, firstly, structure data in a form suitable to localICPregistration function, secondly, run the local registration, and thirdly, keep track of the results.

The data required by local registration function is packed in a structured called as

icpStruct. The *icpStruct* contain: data vertices information in a form of flat array, length of data array, pointer to a kd-tree containing model information, pointers to point pair arrays, pointer to registration matrix, centre of mass values for model and data shapes, and error value.

To increase the speed of the algorithm model data is represented as a kd-tree, see section 3.5.1 this improves the search for closest point from $O(N^2)$ to $O(Nlog(N))$. To restart local algorithm from different starting points we use the *randomHelper.h* file that provides functionality to reseed the pseudo-random generator and get random values in given format and given range.

The implementation of *globalICPregistration* function is summarised in algorithm 9.1

Algorithm 9.1 Implementation of global registration with ICP algorithm

Require: point set P with N_p points **and** shape X **and** maximum number of local ICP to run N
  $e_{best} = \infty$
  $k = 0$
  for $k < N$ **and** $e_k > \tau$ **do**
    $\mathbf{q}_T = random(P_{min}, P_{max})$
    Compute local ICP: $[\mathbf{Q}_k, e_k]$ =localICPRegistration$(P, N_p, X, \mathbf{q}_T)$
    if $e_k < e_{best}$ **then**
      save registration matrix: $\mathbf{Q}_{best} = \mathbf{Q}_k$
    end if
  end for
  return Q_{best}

9.2 Local Registration

The local registration procedure is implemented in *localICPRegistration* function, which takes *icpStruct*, threshold, delta threshold, and maximum number of iterations. The implementation of the ICP algorithm follows the algorithm 9.2

Algorithm 9.2 Local implementation of iterative closest point

Require: point set P with N_p points **and** shape X **and** initial registration offset $\mathbf{q}_T$
  $\mathbf{q}_0 = [1, 0, 0, 0|\mathbf{q}_T]^t$
  $P_0 = P\mathbf{Q}_0$
  $k = 0$
  for $k < maxIterations$ **do**
    Compute random point set: P'_k
    Compute closest points: $[Y_k, d_k = C(P'_k, X)$
    if $d_k < \tau$ **or** $d_{k+1} - d_k < \Delta$ **then**
      return $[\mathbf{Q}_{k-1}, d_k]$
    end if
    Compute the registration: $\mathbf{Q}_k = Q(P_0, Y_k)$
    Apply the registration $P_{k+1} = \mathbf{Q}_k(P_0)$
  end for
  return $[\mathbf{Q}_k, d_k]$

9.2.1 Selecting Points

In meshCombine project we have implemented the random point selection scheme introduced in (Masuda et al., 1996) as it requires little additional calculations, and is easy to implement. Random points are selected in *createRandomIndexPair* function where *dataPair* array, is filled by selecting a random point from 0 to *dataSize*.

9.2.2 Pair Matching

Finding a corresponding point in appositive mesh, to the points selected in previous section, is achieved by using same principle in (Besl and McKay, 1992), by selecting closest point. For distance metric distance from point to point is used. The model mesh is stored as a kd-tree to accelerate the matching process.

The function *findClosestPair* iterates through on the *dataPair* array, generation of which is described in previous section. For each point in *dataPair findClosestPair* function finds a pointer to the closest point in model kd-tree, stores it in *modelPair* array.

The distance between each point pair are added together to find the mean square

distance between meshes after a previous iteration, which is used as an error metric.

9.2.3 Registration Calculation

The function *createRegistrationMatrix* follows the paper (Besl and McKay, 1992) to calculate the registration. Firstly the covariance matrix is created, secondly the $\mathbf{Q}_{(}\sum_{px})$ matrix is filled, and finally the rotation vector is found by calculated the eigenvectors and eigenvalues of the $\mathbf{Q}_{(}\sum_{px})$ matrix.

For calculation of eigenvalues and eigenvectors we used open source software CloudCompare (CloudCompare, 2012). The function *computeJacobianEigenValuesAndVectors* is inspired by the a function *computeJacobianEigenValues* in CloudCompare program, the difference are that by inheriting the function we need to translate it to C programming language from C++, as well as to adopt data input you used standard in *MeshMobine*. Any how the mathematics behind the function is based on method proposed by Carl Gustav Jacob Jacobi in 1846, and is described in modern time in papers like (Golub and van der Vorst, 2000/11/01).

Finaly the translation vector is found by the equation 9.1 from (Besl and McKay, 1992)

$$\mathbf{q}_T = \mathbf{u}_x - \mathbf{Q}_R\mathbf{u}_p. \tag{9.1}$$

After the registration matrix is applied on the data vertex set and the "centre of mass" of data shape, with the function *applyRegistration* the loop restarts from the top.

Chapter 10

Greedy Triangulation Implementation

This chapter goes through the implementation of the triangulation part of the *MeshCombine* project.Triangulation itself is a major subject, and plays a vital role in recreation process. The quality of combined mesh, is highly dependent on the accuracy of the triangulation process.

The key task of the triangulation procedure in *MeshCombine* project is to fill the gap or connect the data to model mesh. This task can also be formulated as hole-filling. An optimal result of this sort of triangulation would be the mesh that preserve as much of the topology information as possible, avoid thin long triangles, creates a water tight transition from model to data, and does not add any triangles that are not approximating the surface between model and data set.

Files that contain the functions that support and execute triangulation in *MeshCombine* are *greedyTriangulation.c/h* The functions that are written in *greedyTriangulation.c* can be divided into three categories:

- Surface triangle graph representation and support functions
- Greedy triangulation functions
- Mesh combination function

10.1 Surface Graph

Previously in section 7.3 a triangular mesh was described as a list of vertices and triangles. This way of representing a triangular mesh contains explicit triangle information. This explicit information is useful when rendering mesh to a screen, as every triangle is an independent shape making the total process easy to adapt for parallel computing. On the other hand this representation method does not provide an optimal structure to extract connectivity information about a specific vertex.

A surface graph contains information about connectivity of the mesh. Connectivity between vertices can be described as, if there is a triangle $t = [a, b, c]$ in a mesh T, there are an edges in the symmetric surface graph G_T between vertices a, b, and c. Surface graph contain implicit triangle information, and explicit information about vertex connectivity.

The explicit edge information of a vertex graph is useful under triangulation process as it can be used to avoid over lapping triangles.

10.2 Surface Graph Representation and Support Functions

In mesh combine we extract connectivity information about model and data set from their triangular-index-array, to create a surface graph. This is done by *copyTriangleInformation* function, in which for every triangle $t = [i_1, i_2, i_3]$ in *elementIndexArray* of a *meshObject* where i_1, i_2, and i_3 are indexes of the points in vertex array, we try to add new edges between the vertices with given indexes.

To keep the creation of the surface graph as fast as possible we try to avoid numerous memory allocation calls, which are time consuming. Instead we use an aggressive memory allocation scheme, where we allocate a memory in larger blocks. To support this aggressive allocation we have implemented a set of stack manipulation functions that keep control over the allocated memory. as well as create/add edge functionality.

10.3 Greedy Triangulation Functions

There are three functions that can be categorised as triangulation functions: *edgeValid*, *toZrotMatrix*, and *greedyTriangulation*.

10.3.1 Greedy Triangulation Algorithm

Greedy algorithms usually produce a result faster that their non-greedy alternatives but the results are not always optimal. Our greedy triangulation implementation can be summarised in algorithm 10.1

Algorithm 10.1 Greedy triangulation algorithm implementation

Require: shape M **and** shape D **and** List P of points to triangulate **and** maximum edge dinstance d_{max}
  create surface graph from shape M and D: $S = G(M, D)$
  create kd-tree from M's and D's vertex array Vm, Vd: $T_{kd} = U(Vm, Vd)$
  $T_{new} = \emptyset$
  while $P! = \emptyset$ **do**
    get a point p from triangulation list P, $p \in S$
    create a neibourhood V by searching for k nearest neibours in T_{kd}: $V = knn(Q, p, d_{max})$
    create weighted normal n_p: $n_p = W(V)$
    create rotation matrix R: $R = toZ(n_p)$
    check if edge from p to v_i is valid and store any new triangle created in T_i: $[valid, T_i] = edgeValid(p, v_i, R,)$
    if *valid* **then**
      add v_i to P
      for all $t_j \in T_i$ **do**
        add t_j to T_{new}
      end for
    end if
    remove p from P.
  end while
  return T_{new}

In section 8.3.2 we have explained how we determent the boundary B between the deleted region X' and rest of the model mesh. We start our triangulation by copying every vertex in B to the vertex triangulation list P. For each vertex p_m in P we collect the neighbourhood K of k-points in a given distance. By using

similar weighting technique from (Marton et al., 2009) shown in equation 10.1, we create a weighted normal $\overline{n}$ given by

$$\overline{n} = \frac{1}{k}\sum_{i=1}^{k}[n_i * \xi_i], \xi = \frac{d_{max}}{d_i}, \tag{10.1}$$

where ξ is weight of the normal, $d_i \neq 0$ is distance between points and p_i and p_m, and d_{max} is the maximum distance for the neighbourhood.

This weighted normal can be used to approximate the surface normal at the point p_m.

In section 5.4 in was mentioned that a 3D surface triangulation problem can be reduced to 2D if the point to be triangulated are to be brought to the plane approximating the surface at a given vertex.

To reduce our problem to two dimensions we rotate every point so the planes normal, $\overline{n}$, is pointing in z-axis direction. Projecting a point to this rotated plane is as simple as ignoring z-coordinates. The rotation is archived by multiplying vertex coordinates by $\mathbf{M}_{rot}$ which is given by

$$M_{rot} = M_x * M_y \tag{10.2}$$

.

The required angle θ is found by

$$\theta = \mathbf{z} \cdot \overline{n}, \tag{10.3}$$

where $\mathbf{z}$ is z-unit vector $\mathbf{z} = [0, 0, 1]^t$.

Because of the unique from of z-unit vector the equation 10.3 can be simplified to

$$\theta = \frac{z_{\overline{n}}}{|\overline{n}|} \tag{10.4}$$

When the rotation matrix and weighted normal is calculated edge creation can be started. For each point p_j in the neighbourhood P we try to add an edge between point p_m and p_j. If the edge is valid than it is added to the surface graph, and point p_j is added to T, if the edge is not valid it is then ignored.

10.3.2 Edge Validation

An edge is structure in a graph between nodes. The intersection between edges is a vague question as it depends on the interpretation of the graph. In our case of a surface graph each node represents a point in 3D Cartesian coordinate system, where two points create a line segment, and lines can intersect each other. If we look at the edges in a surface graph as line segments we can determent if they intersect. A valid edge e is defined as there are no other edges in a surface graph that intercept the edge e. The intersection of the edges can be defined as, edges $e_1$and e_2 intersect if line segments represented by edges projected to the plane approximating the surface, intersect.

10.3.3 2D line segment intersection

Let l_1 be a line segment described by two points a and b on a line L_1, then any point lp_1 on the line segment l_1 is given by

$$lp_1(t) = a + t\overline{ab} \tag{10.5}$$

where $t \in [0, 1]$

Given that line segment l_2, described by points c and d, intersects with l_1 then point of intersection is given by

$$a + t\overline{ab} = c + v\overline{cd} \tag{10.6}$$

From equation 10.6 this we can determent that line segments l_1 and l_2 intersect if and only if $t, v \in [0, 1]$

10.4 Edge Validation Procedure

The Edge validation procedure in mesh combine is given by algorithm 10.2. Let the edge ab stretch from point a to point b. Let a' be an existing point that is connected to the point a with an edge aa', and point a'' be the point connected to a' by the edge $a'a''$. To determent if ab is valid we check for each point a' if there an edge $a'a''$ that intersects with edge ab. The figure 10.1 illustrates a two dimensional example.

Algorithm 10.2 Edge validation algorithm

Require: current point a **and** to point b
  create tringluar list $t = \emptyset$
  if $a = b$ **then**
    return edge not valid: $[0, NULL]$
  end if
  for all edges aa' in a **do**
    if $a' = b$ **then**
      return edge not valid: $[0, NULL]$
    end if
    for all edges $a'a''$ in a' **do**
      if $a'' = b$ **then**
        add tringle to t
      end if
      if $!edgeIntersect(ab, a'a'')$ **then**
        return edge not valid: $[0, NULL]$
      end if
    end for
  end for
  return edge is valid : $[1, t]$

Each new added edge can create new triangles. Our implementation keeps the track of triangle, in edge validation procedure. Following the previous example, if there is a point a'' that is equals to point b that the triangle is created.

10.5 Greedy Triangulation Procedure

Final function *combimeMeshes* is a combination of previous steps. Firstly, the triangular information is transformed from triangle index representation to a surface graph. Secondly, the greedy-triangulation algorithm is started. Finally, the model and data meshes are combined and new triangles are added to combined *meshobject* structure.

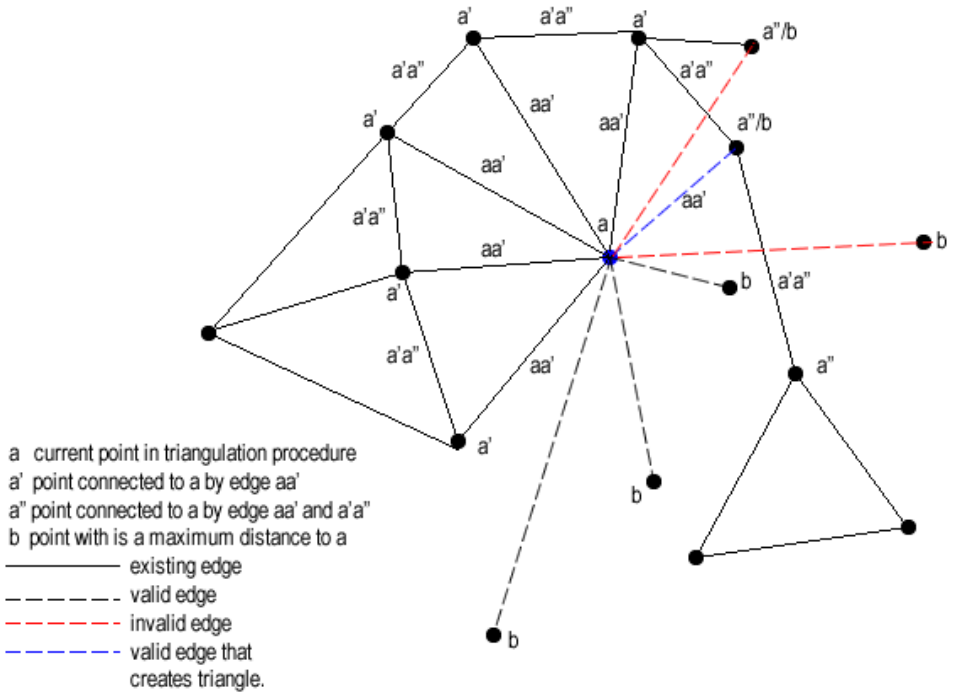

Figure 10.1: Illustration of edge validation procedure.

10.6 Mesh Merging

When all necessary information is gathered to produce the final product, the resulting mesh can be created. The composition of the combined mesh is as follows, the vertex array is simply a model vertex array followed by data vertex array, indubitably the vertex count is a sum of model and data vertex count, the same is applied for normal arrays. Element index list of combined mesh is compound from model and data element arrays, and triangles created in triangulation process, in this order, while the element count is a sum of model and data count, and number of triangles added due triangulation procedure.

When construction of resulting mesh is finished and memory that have been allocated and only used within a function call is freed, the pointer to the structure *geometryMesh* is returned.

Part IV

Tests and Procedures

Chapter 11

Test Objects and Scenes

In this chapter we will introduce the test sets and procedures that were executed to test the *MeshCombine* software.

11.1 Test Sets

The *MeshCombine* program was tested on two sets of meshes, the scan of an archaeological fragment and a scene of an office desk. Models meshes of both sets are created by using photogrammetric software photoModeler, while data is captured by Artec's 3D scanner EVA. By using two different scanning technologies we not only create testing sets of different resolutions, but also ensure that noise characteristics and vertex quality would vary from data to model mesh. This different origin of scans gives a unique challenge for *MeshCombine* software.

11.1.1 Nidaros Stone

Nidaros stone is a good testing object because it contains different types of surfaces, with different grade of smoothens and concavity. In the illustration of the stone, figure 11.1, we can observe a concave surface, and two types of roughness.

Concave surfaces are difficult to scan with optical sensors, due the natural occlusion they create, as well as low angles, between scanned surface normal and camera direction, results in noisy values. When triangulating concave surfaces other difficulties arise. In regions where a surface bends and forms a loop like pattern

Figure 11.1: Nidaros stone. A fragment of axiological artefact from Nidaros Cathedral.

it is easy for triangulation algorithms to connect the appositive sides of a loop, preserving the continuity but wrongly approximating the surface. In the terms of identifying the duplicated region the problem with the concave surfaces arise in the same loop shape situation, where one side of the loop can be misinterpret to represent the other part of the loop.

The recreation of the stone done in photoModeler is treated as a low resolution model, while the concave frontal part, scanned in high resolution with EVA 3D scanner, is referred as data. This division of the same object in different parts is a realistic scenario. To capture the concave part of the Nidaros stone, extra care due the scanning procedure is necessary to archive a high level of derails.

11.1.2 Deck Scene

This test set illustrates the purpose of the software: to replace individual parts of the mesh with a high resolution representation. In the centre of the deck scene we

can observe porcelain figurine in a shape of a pig. This figurine is considered as region of interests, and is scanned with high precision scanner, while the rest of the scene is captured with much lower resolution. The purpose a this lower resolution scan is to create reference for orientation.

Figure 11.2: Desk scene. A scene with different objects to test *MeshCombine* software.

11.2 Minimum Square Distance

The quality of a mesh representation can be assessed by evaluating the minimum distance, from each vertex in a mesh to the actual surface. The minimum distance between mesh and the surface gives us a good estimation of how well the existing mesh represents the actual surface.

Unfortunately this does not give any indication of how well the mesh covers the surface. Deleting parts of the mesh that have distance to the mesh higher then

certain value will improve overall statistic of a mesh but will also reduce the percentage of the aria covering the surface.

By reversing the distance metric and calculating the distance from every point on a surface to the mesh will incorporate the coverage of the mesh, but on the other hand this metric will not penalize noisy values that are far away from the original surface.

In our testing we will use mesh to surface distance to assess the quality of mesh merging. As our model, data, and ground truth mesh are not guaranteed to be watertight (without holes) checking meshes for topological continuity is not effective way to determent the quality of representation. The coverage of the recreated models is thereby only examined visually.

11.3 Tests Description

To create an objective error measurement both test sets were scanned in high resolution. This complete high resolution scan is treated as "ground truth". Testing procedure for mesh combine is as follows, firstly, the minimal square distance between ground truth and the model is measured to capture the difference between them, secondly, the data and the model is combined in *MeshCombine* software, thirdly, the square distance between the ground truth and combined mesh is calculated, and finally, the difference between the error of the model and combined mesh is calculated to present the result.

This procedure will show if overall error have increased or decreased by introducing a higher resolution region in model mesh. Unfortunately this error metric will not provide the information about quality of the triangulation.

There are several criteria that describe a good triangulation, a good triangulation has no overlapping triangles, no holes, and maximise the internal minimal angle of all triangles. Greedy triangulation does not guaranty that any of the triangulation criteria will hold. Running precise tests on triangulation quality is unnecessary, on the other hand we can observe if triangulation produce unwanted visible artefacts, as noisy triangles, holes, and etc.

11.4 Colouring Meshes Based on Distance in CloudCombine

CloudCombine software provides functionality to colour each vertex based on the previous minimal distance calculation, this colouring scheme gives a good vitalisation of noisy regions. The colouring process is as follows, each vertex in compared mesh is given the colour based on the distance between the vertex and closest point on the reference mesh. Let the colour red symbolise maximum distance d_{max} from compared to reference mesh, while colour blue is minimal distance d_{min}. Then the colour $\mathbf{c}_i$ of a vertex $\mathbf{v}_i$ that have minimal distance $d_i \in [d_{min}, d_{max}]$ to the reference mesh, is given by linear interpolation

$$\mathbf{c}_i = \mathbf{c}_{blue} + \frac{d_i}{d_{max}}(\mathbf{c}_{red} - \mathbf{c}_{blue}). \tag{11.1}$$

11.4.1 Nidaros Stone

The scans and the required photographs of the Nidaros stone were captured by the author in the fall semester during the Specialization Project Thesis (Kongevold, 2012).

Ground Truth

The complete recreation of Nidaros stone were done with Artec Group technology, by using structured light scanner EVA for capturing scans and Artec Studio 8.1 for post-processing. This resulted in a high resolution mesh, Stone.ply, of 1 million vertices and 2 million triangles.

Model Mesh

Model shape was acquired by taking photographs of the Nidaros Stone and then processing them in photogrammetric software photoModeller Scanner. This resulted in a mesh StoneNC.obj with 60 thousand vertices and 130 thousands triangles.

The distance between the model and ground truth was calculated by using cloudCombare software and is summarised in the table 11.1. The colouring of the

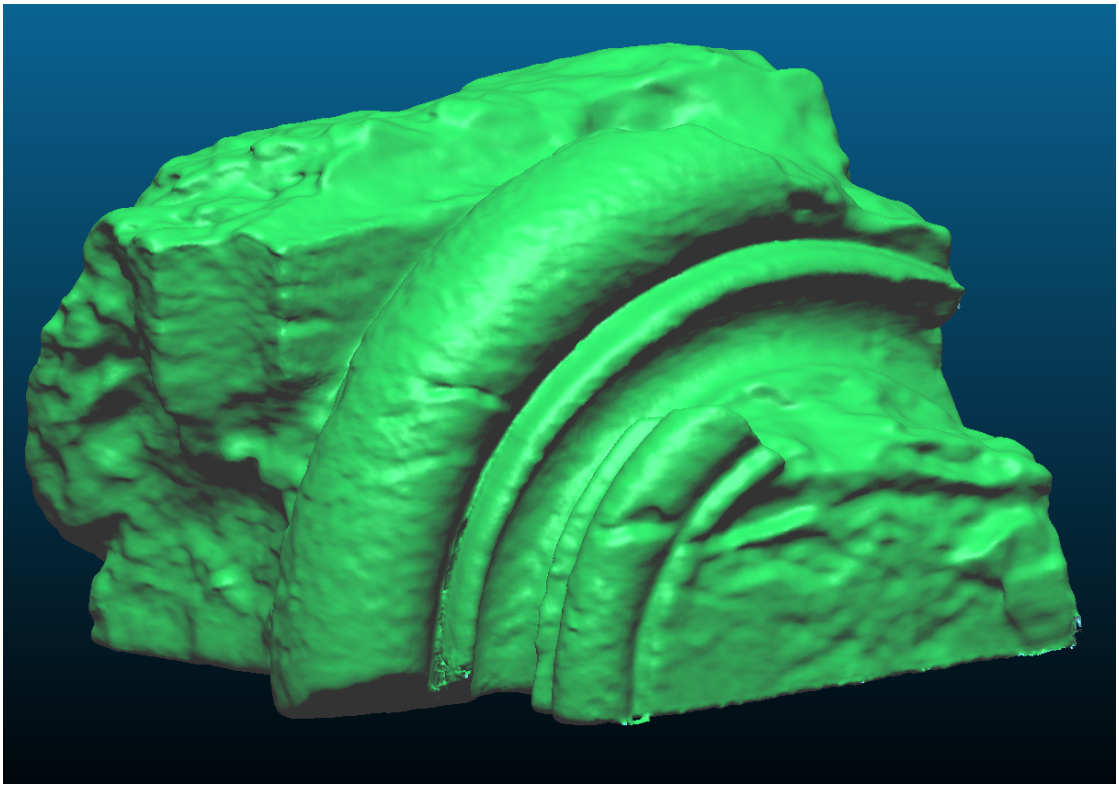

Figure 11.3: High precision recreation of Nidaros stone using Artec 3D scanner EVA.

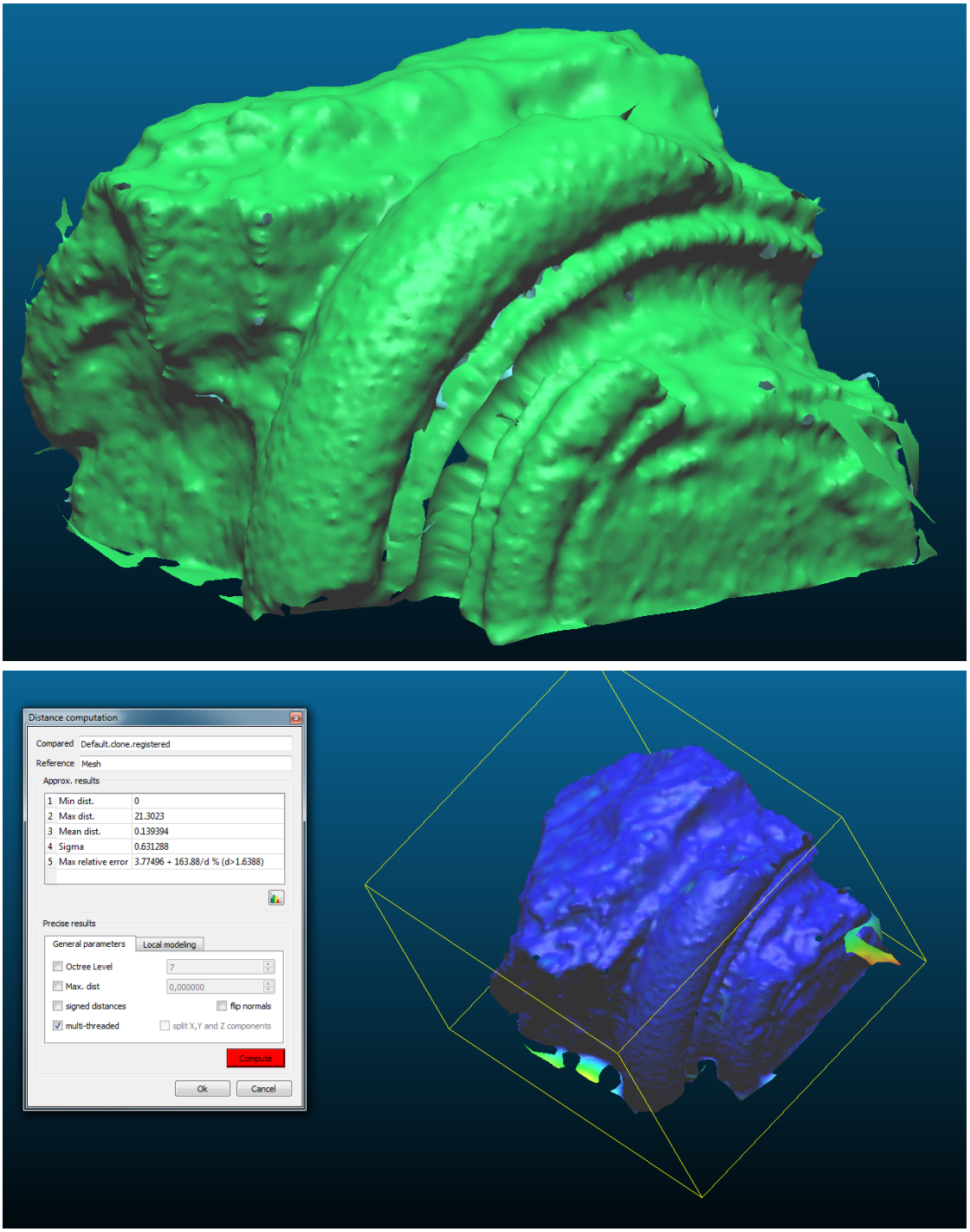

Figure 11.4: Nidaros stone mesh representation by using PhotoModdeler Scanner(top), model to ground truth distance visualisation (bottom).

Minimum distance	0
Maximum distance	21.3023
Mean distance	0.139394
Sigma	0.631288

Table 11.1: Results of minimum square distance calculation between model and ground truth of a Nidaros Stone set.

model in the figure 11.4 is based on the distance towards the ground truth, colour blue represent the smallest error when colour red is given to the vertex with largest distance to ground truth.

Data Mesh

Data mesh was produced by only selecting the frontal part of the ground truth shape, figure 11.5.

11.4.2 Deck Scene

Desk scene was set up in Visualisation lab at IDI (IDI), see figure 11.2 for illustration.

Ground Truth

Ground truth was acquired by scanning the scene with Artec 3D scanner EVA to produce a high resolution mesh, figure 11.6.

Model Mesh

To digitalize the scene images were captured with pre calibrated Minolta 7 digital camera and processed in photoModeler Scanner program. For camera calibration the procedure shown in videos on photoModeler web site (EosSystems, 2012b), ware followed with constant focus length and zoom of the camera. The settings of the camera were constant due camera calibration and image acquisition. Unfortunately the resulting mesh was too noisy and could not be use, as reconstruction of the scene provided barely recognizable results, figure 11.7.

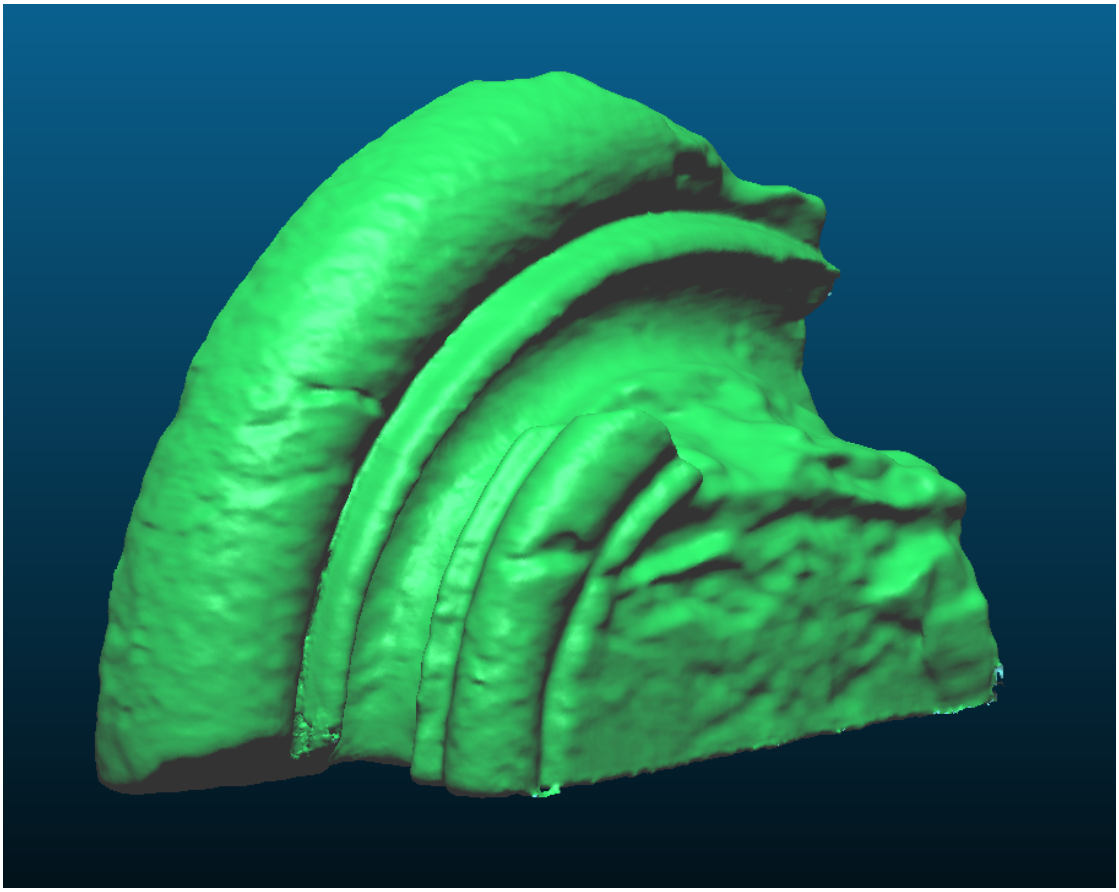

Figure 11.5: A part of high resolution scan serving as data mesh.

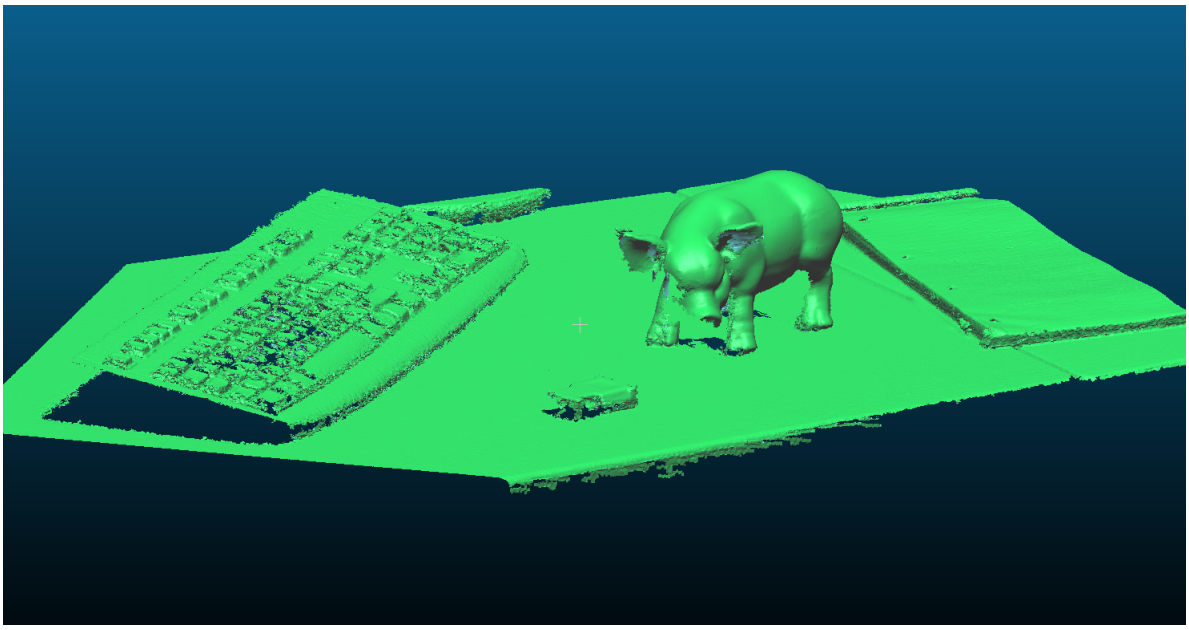

Figure 11.6: Mesh representation of the desk scene by using Artec 3D scanner EVA.

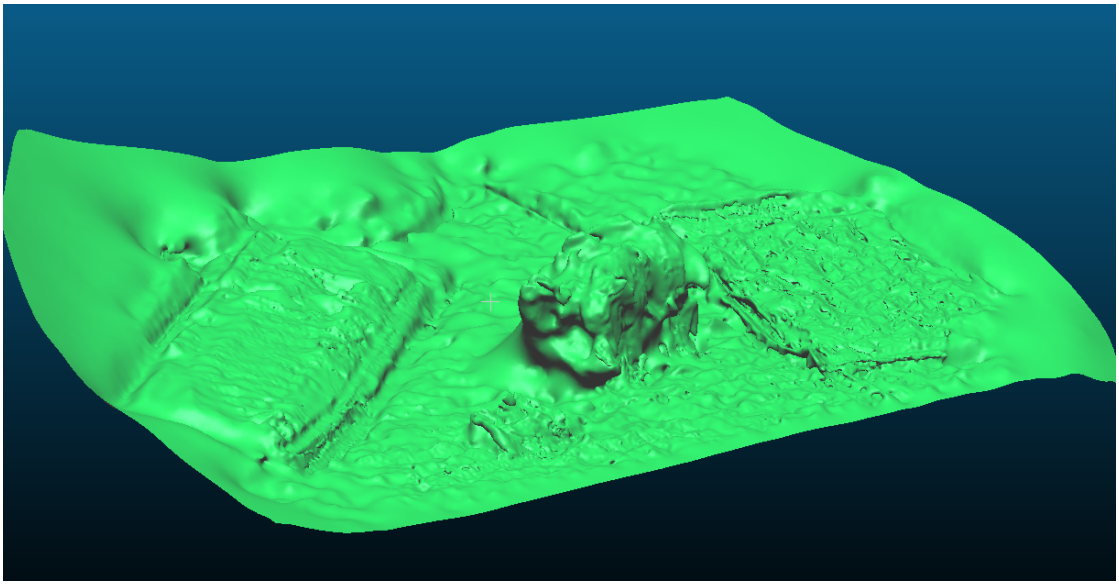

Figure 11.7: Unsuccessful recreation of the desk scene by using PhotoModdeler Scanner.

Minimum distance	0
Maximum distance	7.00054
Mean distance	0.00982883
Sigma	0.262127

Table 11.2: Results of minimum square distance calculation between model and ground truth of a desk sceene.

Because of this failure to reconstruct the scene with photoModeller Scanner, we used a down sampled mesh of the ground truth as a model, figure 11.8. The minimum square distance is shown in the table 11.2.

Data Mesh

Data mesh is created by scanning the figuring of the pig placed on a turn table with 3D scanner EVA. The result was a high resolution mesh, with no holes and representing 100% of visible surface, figure 11.9. The data mesh in this case is better representation that the ground truth model, as it was hard to obtain 360 degree scan of the desk scene with artec 3D scanner EVA.

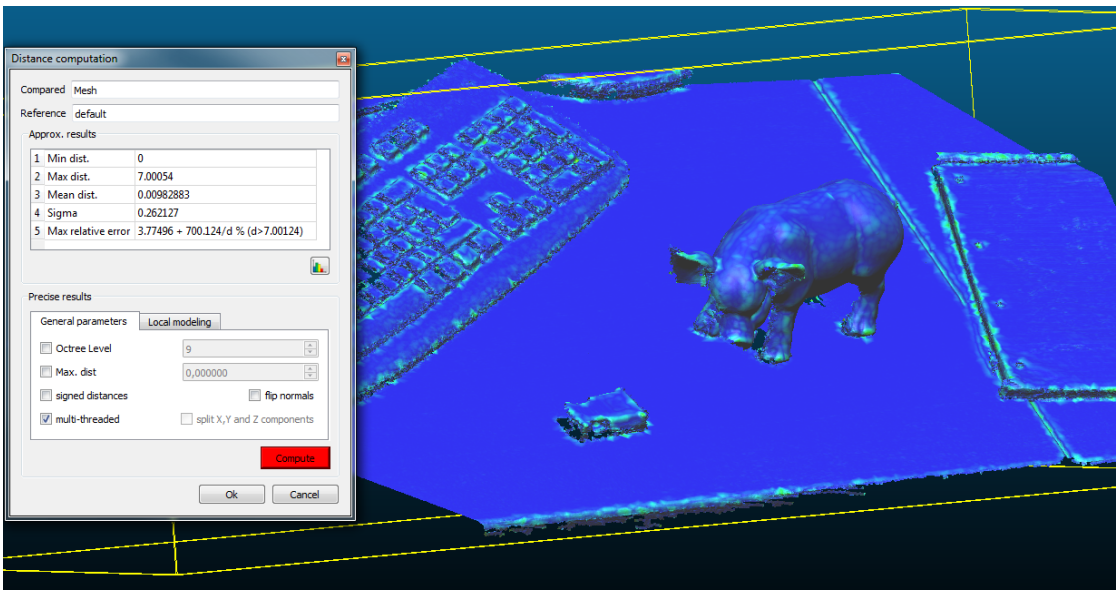

Figure 11.8: Desk sceene, model to data distance visualisation.

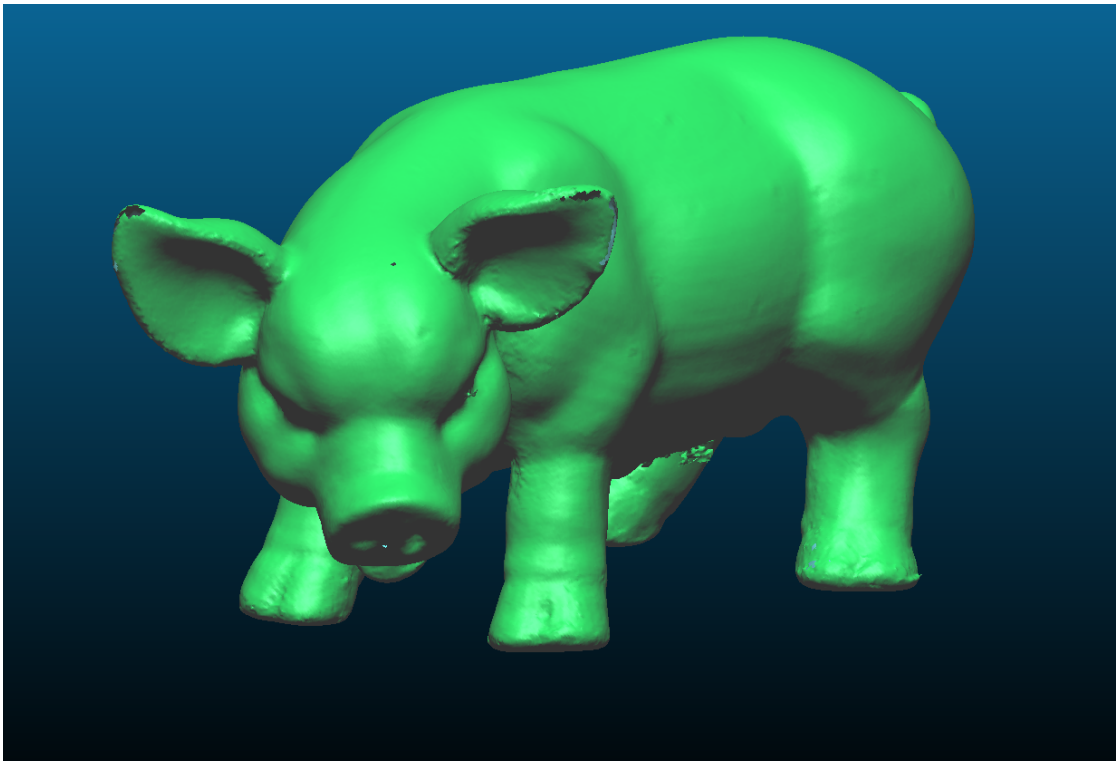

Figure 11.9: High precision scan of the porcelain pig figurine.

Chapter 12

Results

Within this chapter we present merged results of *MeshCombine* tool. There are four key features we will focus on, alignment of meshes, duplicated region estimation, triangulation, and overall change in minimal square distance between meshes.

12.1 Alignment

The result alignment of the data mesh to the user selected region produced unsatisfactory results, as ICP resulted in local minima, which is common for this sort of algorithm. To continue testing, the data and model mesh were aligned in a third party software, Artec Atudio v8.1 (ArtecGroup, 2012) and saved as ply files. The rest of the combination procedure was tested on manually aligned meshes. Error between aligned model and ground truth for each set mesh is the same, as alignment is required to calculate the minimal square distance.

The examples of bad alignment are represented in figure 12.1

12.2 Region Identification

Region identification is a vital task. If the region estimated is bigger than the actual region then deleting vertices deletes information, if smaller, then remaining model vertices corrupt data region. In correct region estimation provides bad

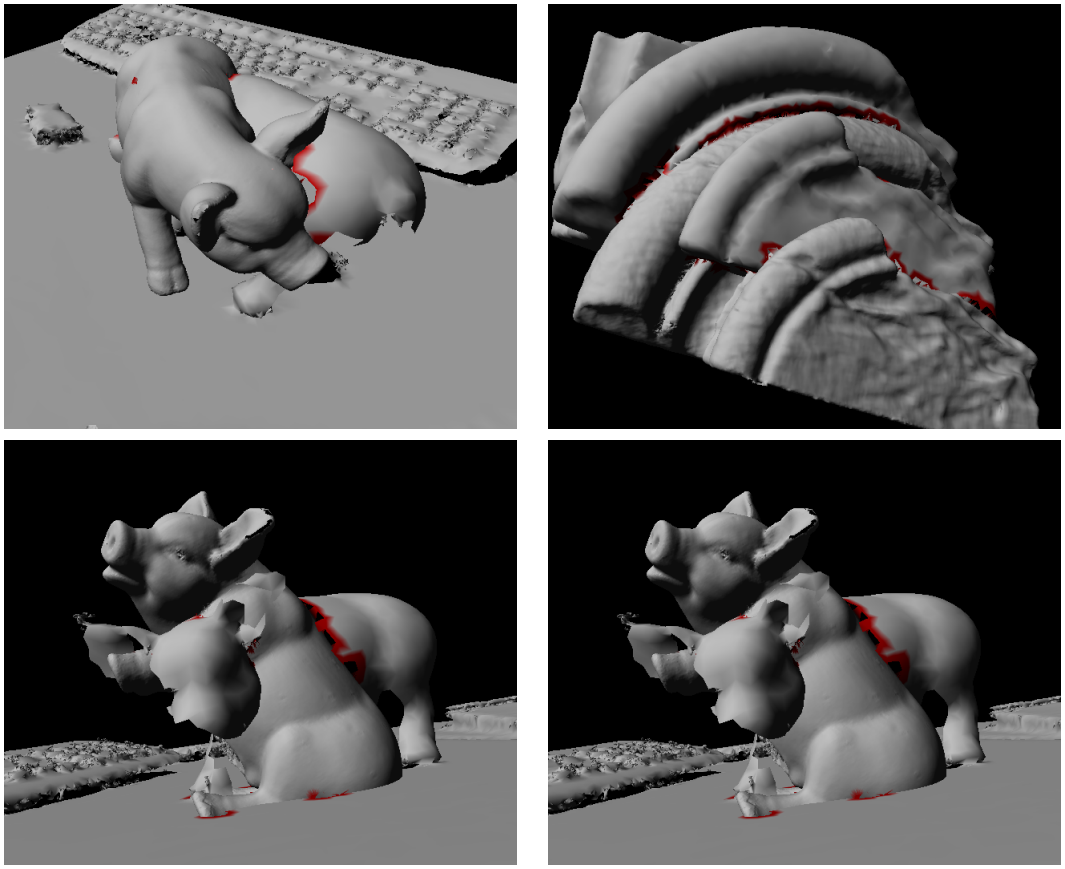

Figure 12.1: Bad alignment from ICP algorithm, resulting in local minimum.

boundary information that seeds the triangulation process.

There are two procedures to estimate the duplicated region in *MeshCombine* project, by using directional distance or spherical. Results are hard to test the region estimation objectively on a data set obtaining by scanning a physical object, without proceeding to the next phase of mesh combination. For a more subjective feeling of each approach we altered the program to stop after region approximation and to colour the estimated region with red colour. The figure 12.2 displays the regions estimated with the same meshes and same maximum distance, but different distance metric, section 5.1.

We would like to point out the boundary regions presented in figure 12.3. The spherical distance, image to the left, marks more of the model surface that distance in normals direction, on the other hand in concave and more noisy regions the spherical distance produces more grouped markings, resulting in more precise

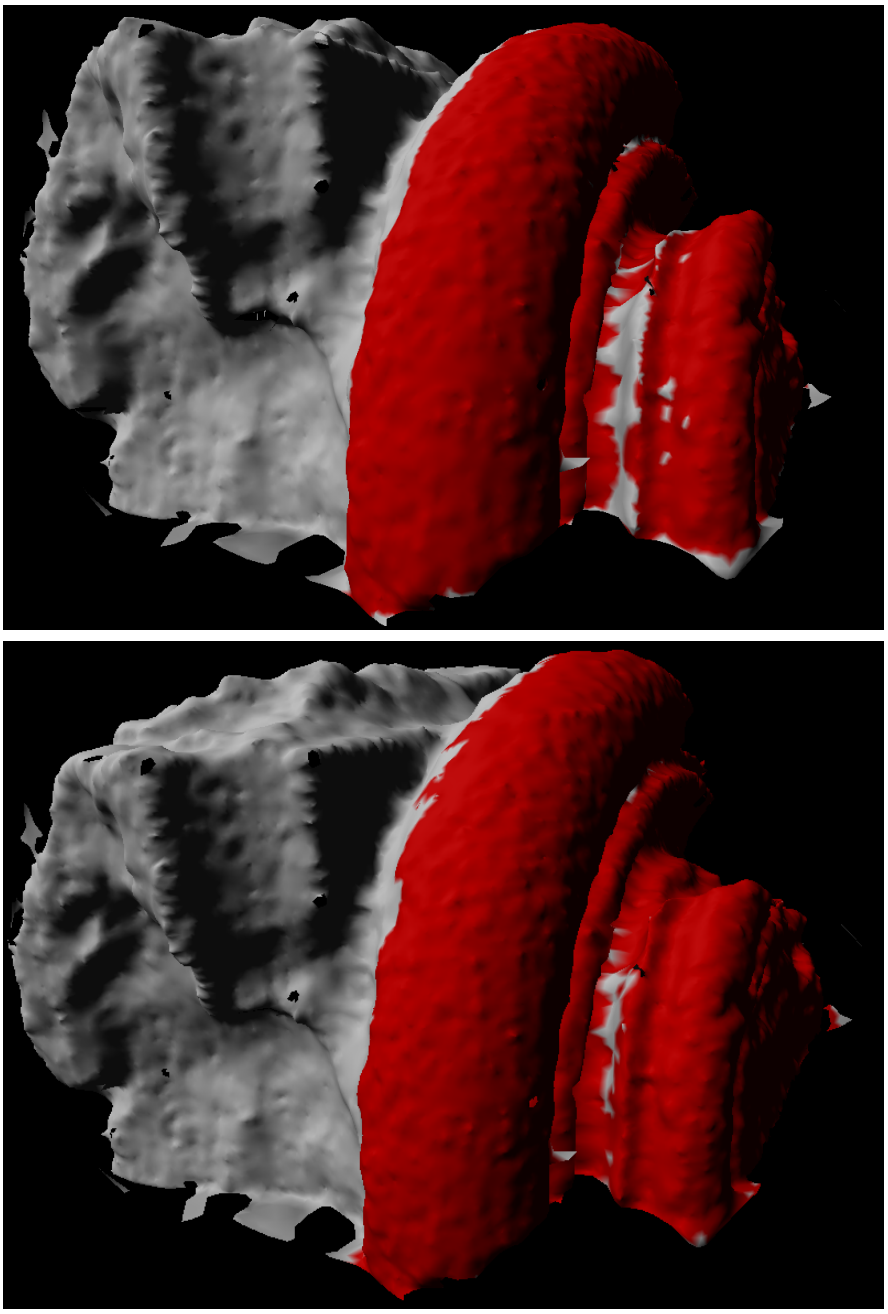

Figure 12.2: Region estimation by using directional distance in the direction of approximating surface normal, above, and using Euclidian distance, below.

boundary.

12.3 Gap Triangulation

Triangulating the gap or stitching as is sometimes called is a final step in mesh combining process. The triangulation does not add new points but it preserves continuity over the transition between surfaces.

For each testing set, we produced two combined meshes: one was created by using spherical distance to identify duplicated region and the other one by using distance in the direction of the surface normal. Combined meshes are illustrated in figure 12.4, and figure 12.5. The red line indicates the vertices that start greedy triangulation process, also referred as the boundary between duplicated region X' and the rest of the model mesh. We can observe that the triangulation mostly succeeds on the true boundary region, but noisy seeds, that are not a part of the gap between meshes, tend to produce falls triangles. Those falls triangles are easy to spot in the region of concave surface, figure 12.6.

12.4 Minimum Square Distance

After manual alignment of model and data mesh, the rest of the algorithm could be restarted and alignment step could be ignored. The resulted mesh has been written to disk as a wavefront .obj format. At the end we used open source software CloudCompare, to calculate the minimum square distance between combined mesh and the high precision scan treated as a ground truth of a set. These distance values give us an objective measure of the quality of the final result.

To visualise the local improvement of the mesh we have the resulting meshes has been coloured based on their distance. If a maximum distance of the mesh is much larger that the mean error between regions, that it is hard to observe the colour change, to avoid this we have limited the colouring scheme to only display and colour vertices in the region of $[0, 3.3]$ this way resulted in more illustrative colouring.

	normal	radius
Minimum distance	0	0
Maximum distance	21.2962	21.3093
Mean distance	0.0489126	0.0505309
Sigma	0.32576	0.329806

Table 12.1: Minimum square distance between combined mesh and ground truth of a Nidaros stone test set.

	normal	radius
Minimum distance	0	0
Maximum distance	30.3357	30.3357
Mean distance	2.25392	2.52957
Sigma	5.3374	5.34203

Table 12.2: Minimum square distance between combined mesh and ground truth of a desk scene test set.

12.4.1 Nidaros Stone

The result of distance metric from combined mesh to the ground truth mesh is summarized in the table 12.1, the second column displays values for the mesh created by using radius, or Euclidian distance, third column represents by using distance in direction of the normal. Combined meshes coloured based on distance to ground truth for the Nidaros Stone set are given in the figure 12.7. The figure 12.8, shows same combined mesh of the Nidaros stone but colouring is limited to the previously discussed range.

12.4.2 Desk Scene

The Euclidian distance from combined desk scene mesh and to high resolution scan, ground truth, is given in the table 12.2 and figure 12.9. Figure 12.10 shows the visualisation of the results with limited to the range of $[0, 3.3]$ colouring.

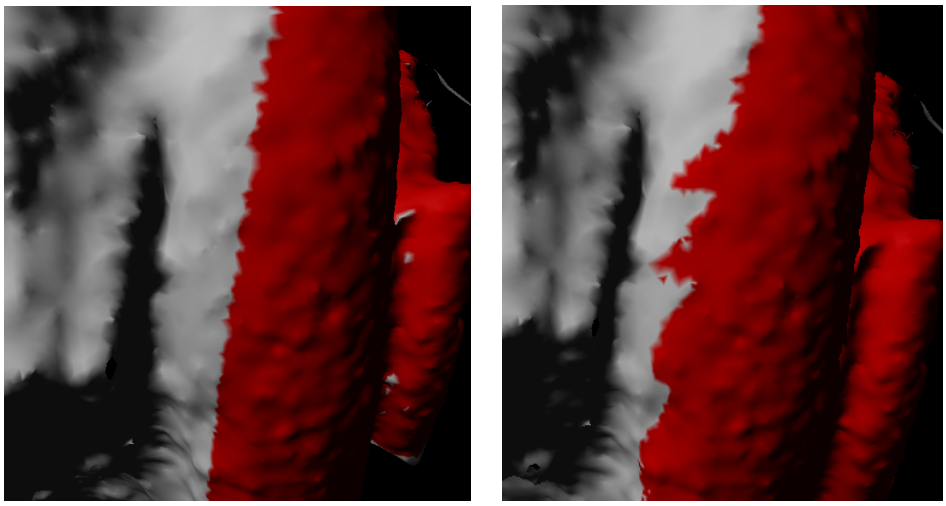

Figure 12.3: Different boundary of the duplicated region, cause by different identification metric, directional - left, spherical - right

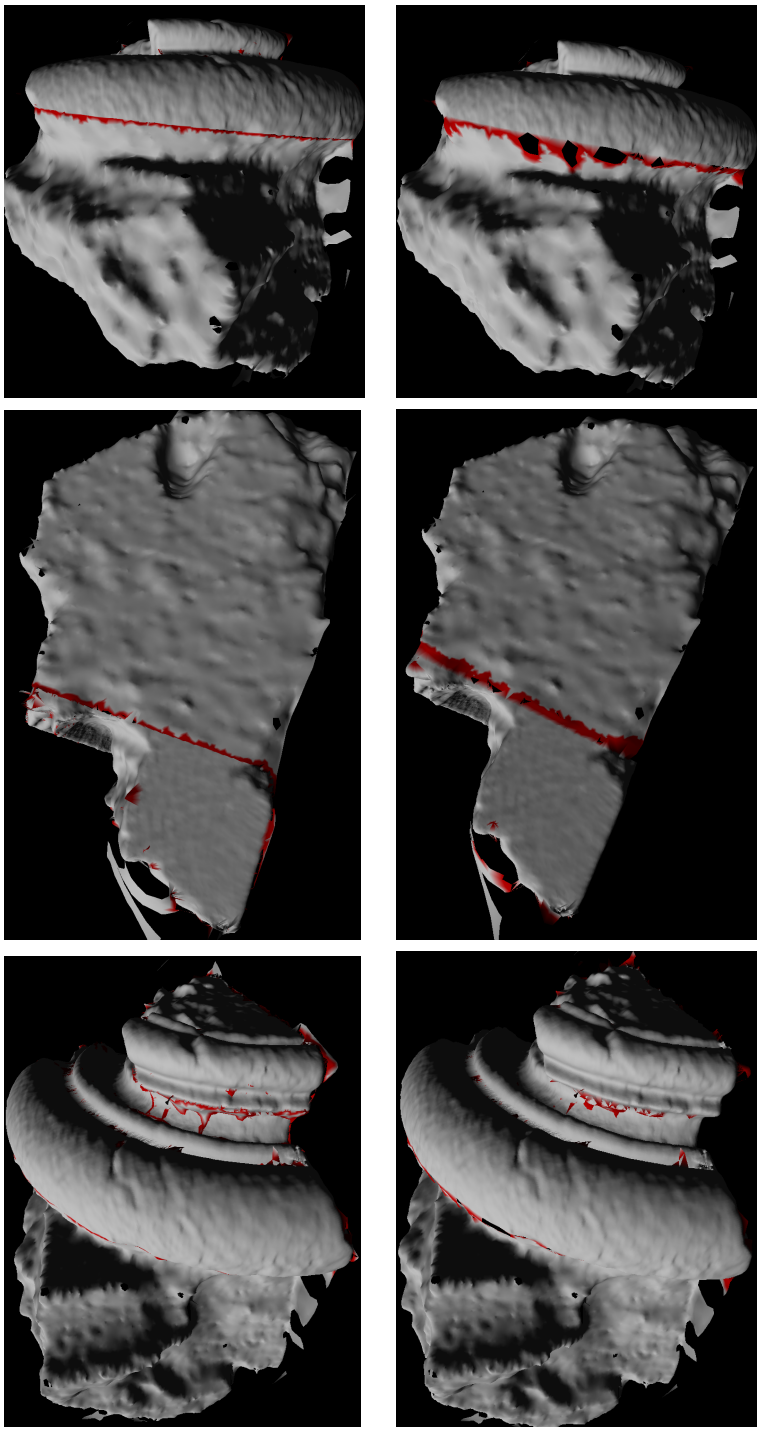

Figure 12.4: Triangulation results on Nidaros stone test set, directional distance - left, spherical distance - right.

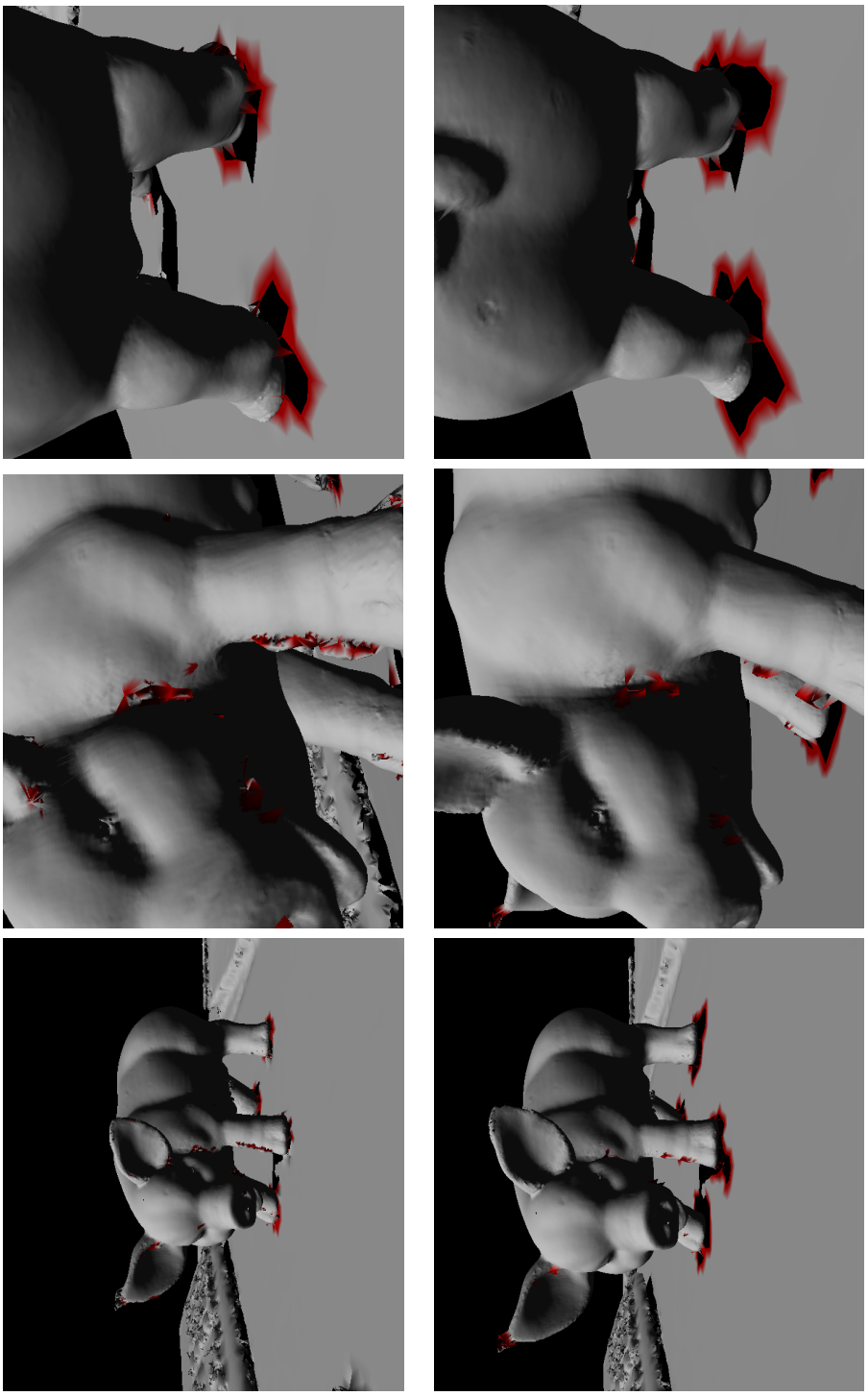

Figure 12.5: Triangulation results on desk sceene test set, directional distance - left, spherical distance - right.

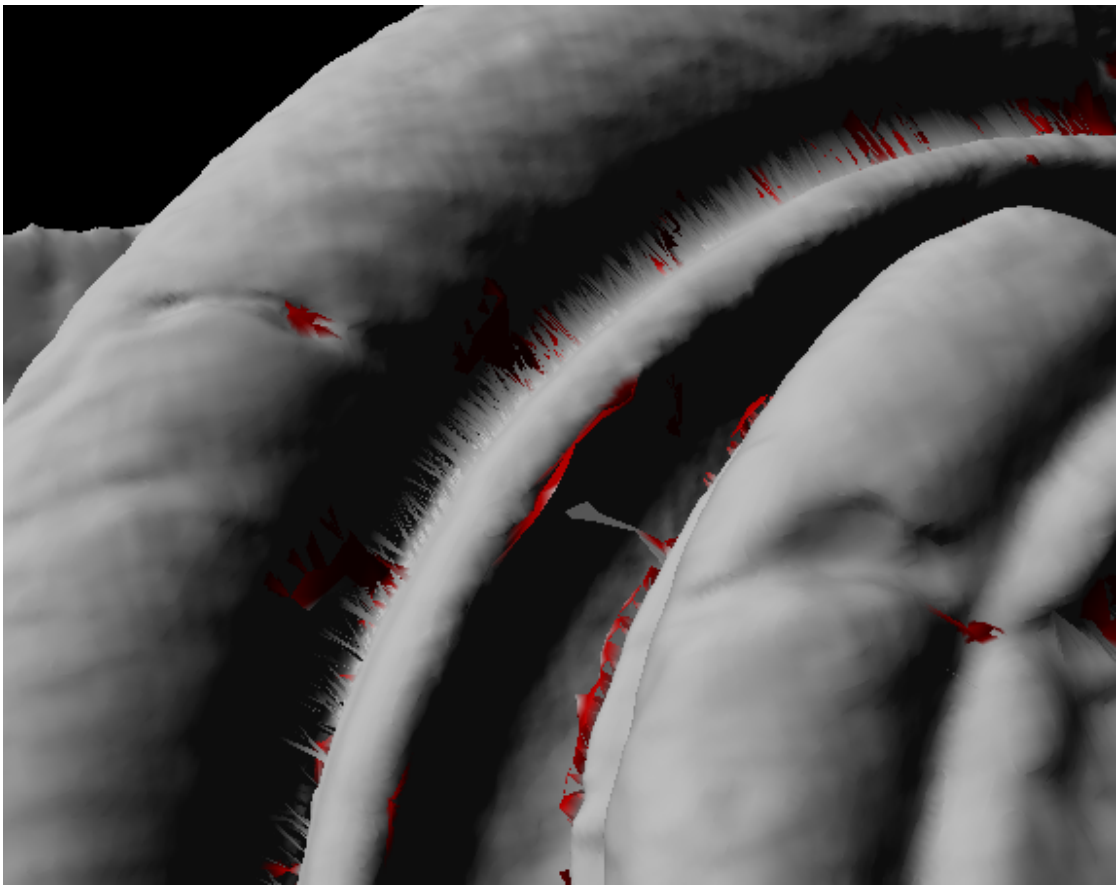

Figure 12.6: Wrongly triangulated surface in the upper concave region of the Nidaros stone set, using directional distance.

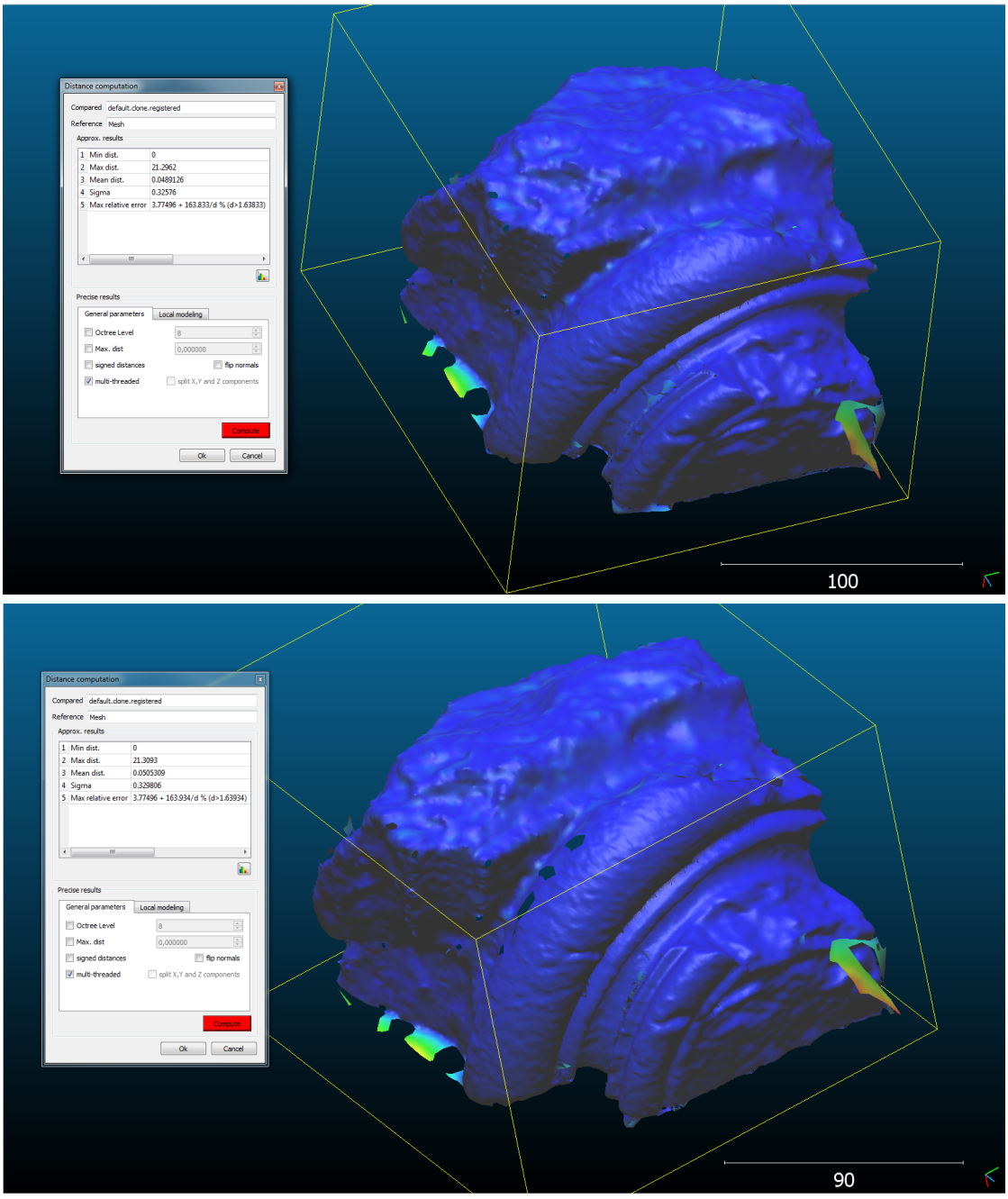

Figure 12.7: Visualisation of distances between merged and ground truth mesh of the Nidaros stone test set, directional distance - above, spherical distance - below.

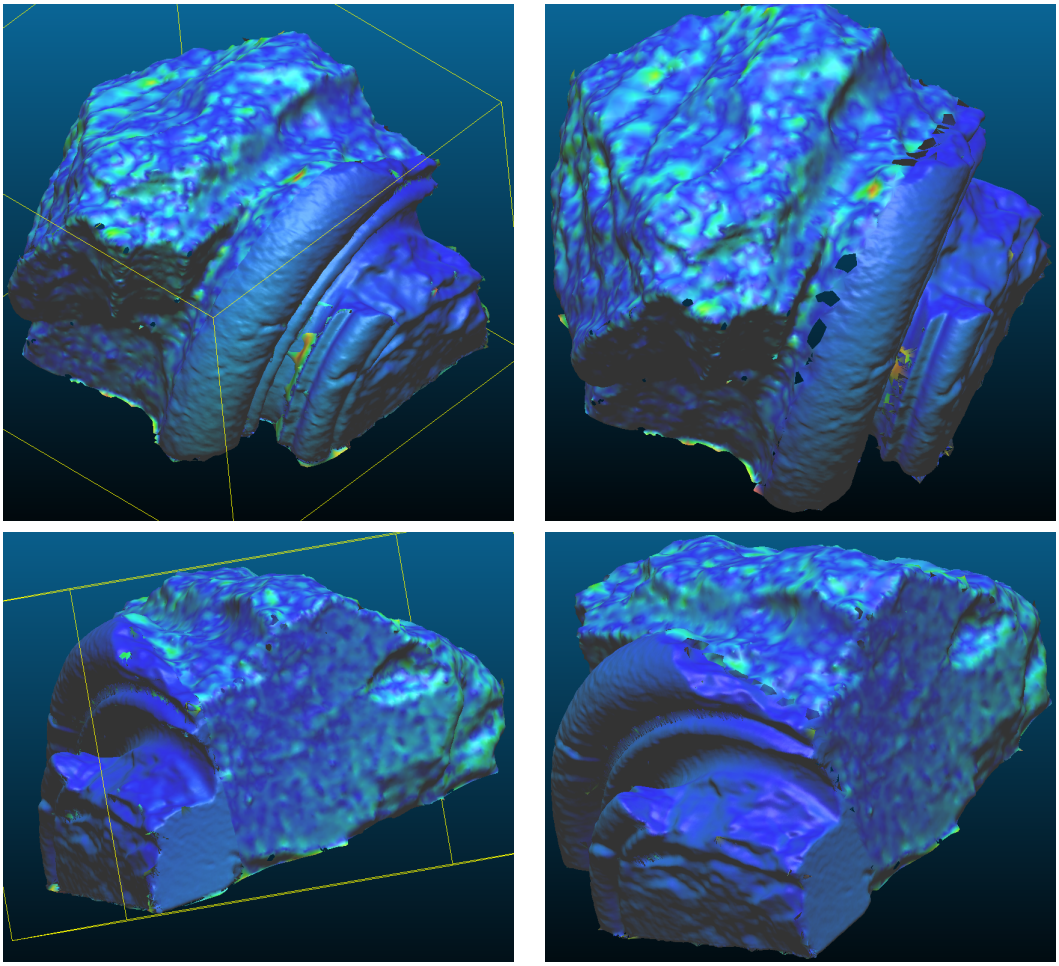

Figure 12.8: Visualisation of distances between merged and ground truth mesh of the Nidaros stone test set, bounded within the region of [0,3.3], directional distance - left, spherical distance - right.

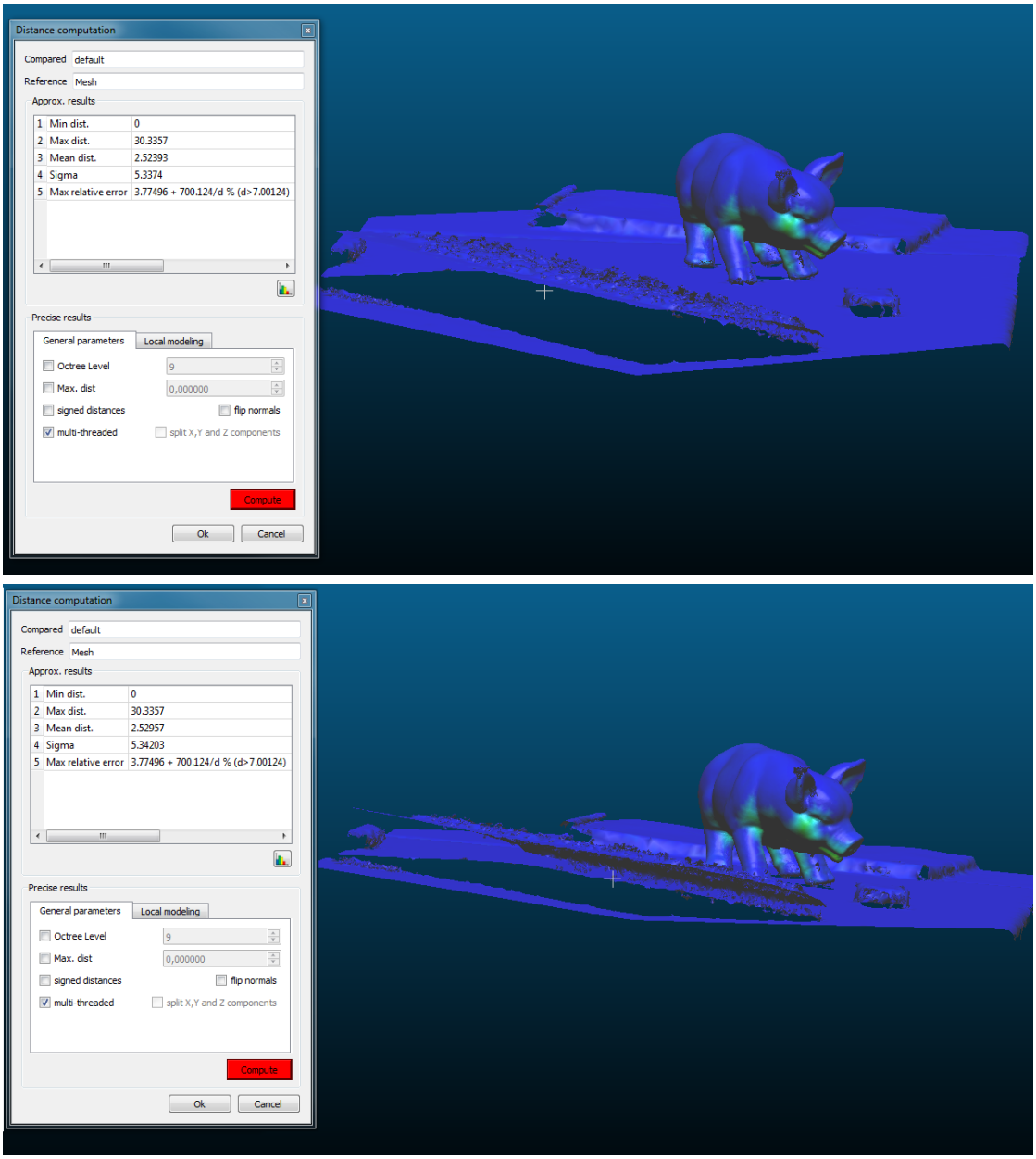

Figure 12.9: Visualisation of distances between merged and ground truth mesh of the desk scene test set, directional distance - above, spherical distance - below.

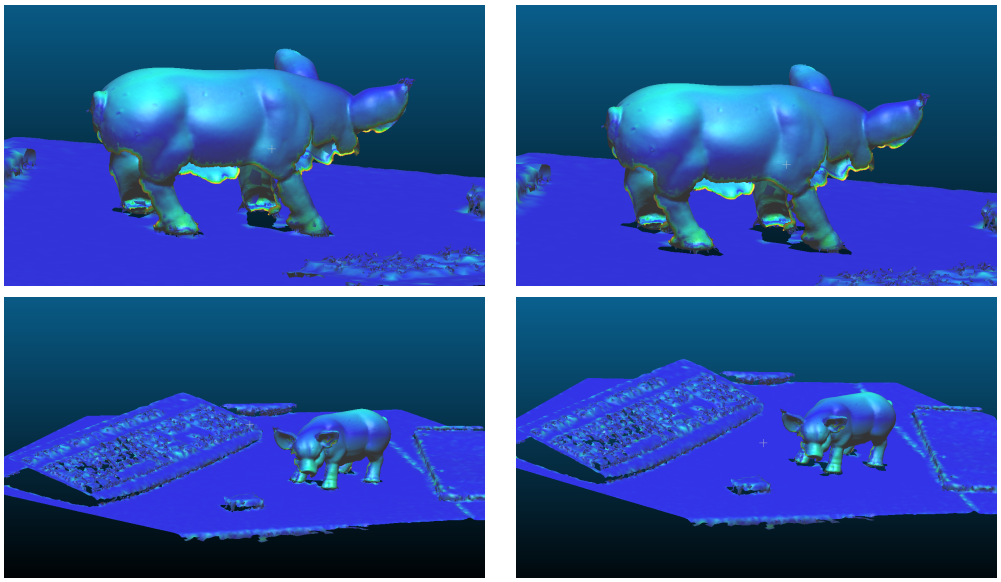

Figure 12.10: Visualisation of distances between merged and ground truth mesh of the desk scene test set, bounded within the region of [0,3.3], directional distance - left, spherical distance - right.

Part V

Closing Remarks

Chapter 13

Discussions

In this chapter will analyse the results acquired from testing, and discuss the possible reasons for deviation from expected values. At the end we will propose-posibilitys future work, and ways to improve *MeshCombine* program.

13.1 Interpriting the Results

Understanding the results can be as vital as performing tests. Both negative and positive results can provide insight that can be useful to improve future research. This section will focus on four main aspects of our tests, mesh registration, duplicated region estimation, gap triangulation, and square distance between combined mesh and ground truth.

13.1.1 Iterative Closest Point

Local ICP algorithms resulted in global minima, and could not archive an optimal registration. This problem was addressed already by the inventors of the algorithm, Besl and McKay. The random translation scheme did not resulted in positive results. This failure can be due a complicated shape of the mesh, where a global minimum is surrounded by local minima, so if initial registration does not start on the slope leading to global minimum, the optimal registration is not possible, in this scenario only random translation is not enough, random rotation of the model is also necessary. This lack of random initial rotation can be the reason why global registration scheme failed.

13.1.2 Duplicated Region Estimation

We have tested two different region estimation approaches, using Euclidian distance and normal directed distance. Both approaches produced usable results with expected differences. We can take a closer look at the results of region identification, in the figure 12.2 we can observe the difference between using different distance metrics.

The image at the bottom displaying region estimation by using distance from vertex to mesh, Euclidian distance, we can see that the region is larger and more continues than the region estimated by using distance in direction of the vertex's normal. We can also observe that spherical distance is more resistant to noisy vertices than directional distance. The removal of noisy values is crucial for quality of the resulting mesh, noisy point not only increases error metric value, but also results in false boundary, with corrupts the triangulation algorithm later in the combination process.

Using the normal of the current vertex as the direction for the distance metric is proven to be more sensible to noise. On the other hand it resulted in a more precise boundary between duplicated region X' and the rest of the model mesh, which later on resulted in better triangulation. In the figure 12.3 we can observe a clear difference between approximated duplicated regions, while the normal distance resulted in a smooth line, the spherical distance resulted in a jagged curve. This difference propagates to the next step, triangulation, and results in holes, which can be observed in figure 12.4, the image at upper-right.

13.1.3 Triangulation

Greedy methods do not guarantee optimal results. In our case the triangulation process resulted in mixed results. In the case of Nidaros stone, the gap was well triangulated, with only few holes remaining, but in the case of the desk scene the hooves of the porcelain pig were badly triangulated to the surface of the table.

This different triangulation quality can come from normal weighting procedure we discussed in section 10.3.1, we can observe from images, figure 12.4 and figure 12.5 that in the case of Nidaros stone the surface of data mesh and model mesh are close to parallel, but in the case of desk scene the angle between normal are close to 90 degree. This can explain the different behaviour of the algorithm.

	Nidaros stone		Desk scene	
	Normal	Radius	Normal	Radius
Minimum distance	0	0	0	0
Maximum distance	-0.0061	0.007	22.99946	22.99946
Mean distance	-0.0904814	-0.0888631	2.24409177	2.51974117
Sigma	-0.305528	-0.301482	5.075273	5.079903

Table 13.1: Difference between distance between model and combined mesh

False Triangles

False triangles are those that were created during triangulation process that do not represent the actual surface.

By visually examining results we can observe false triangles in the frontal part of the recreated mesh of the stone mesh, figure 12.6. On the same image we can also observe wrongly determent boundary, as each vertex that was used as seed in triangulation process is coloured red.

This gives reason to believe that those triangles are created because the triangulation process was wrongly seeded. On the other hand concave regions have proven to be a challenge for triangulation algorithms.

13.1.4 Minimum Distance Error Metric

The change in distance after introducing high resolution data region into the model mesh is presented in table 13.1.

As we can see in column four and five of the table 13.1 the maximum distance has increased dramatically, this increase is caused by the fact that the ground truth scan was not able to capture the region of interest as completely as our data scans. In figure 13.1 our assumption is supported by visualisation of minimal distances between the combined mesh and groud truth scan. There by the minimal distance from combine mesh to ground truth is not representative, as mesh used as ground truth was not detailed enough.

Yet, the desk scene test set is not completely useless, as through this test we demonstrated a possibility to integrate a complete representation of the region of interest into a scene that was not possible to scan entirely. By using this kind of tool a user can improve a chosen scene incrementally, by introducing a more

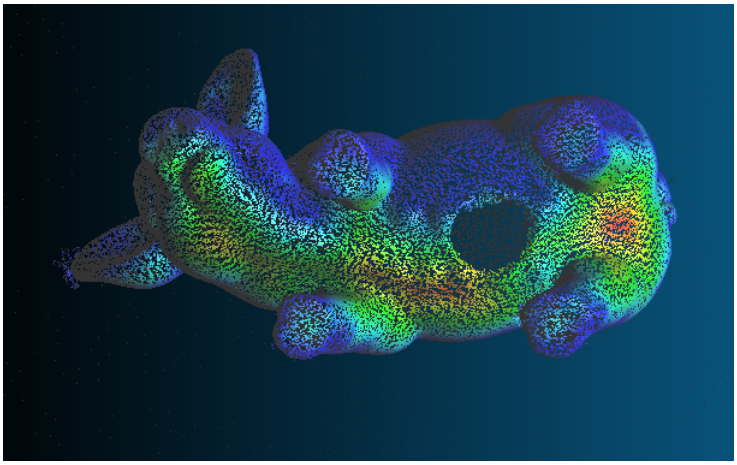

Figure 13.1: Insufficient ground truth representation. The regions that were not present in ground truth mesh were treated as noise.

complete or detailed representations of specific regions in the scene. Never the less this test has also shown that a new measure of quality is required to evaluate the results, as simple square distance between meshes is not sufficient.

On the other hand in the case of Nidaros stone test set, where the "ground truth" scan did covered the whole test object, a reduction in deviation from ground truth was noticed. A change in mean distance and sigma of the distribution is a sign of overall improvement. We can observe a two time decrease in mean value and sigma. By analysing the colouring information in figure 12.8 we can see that the data mesh region contain less variation in colour, then region originated from the model mesh. This distribution of distances is especially noticeable in Nidaros stone test set, as the model and ground truth meshes were originated from different scanners. In desk scene test set the model mesh is originated from down sampling the high precision scan, there by the error is mostly visible on sharp edges, for example keyboard in figure 11.8.

13.2 Discussion

The result of testing the *MeshCombine* product gave a vital insight to some problems and complications, in the process of combining meshes. By using the knowledge we obtain from this implementation we can precede to improve the software.

13.2.1 Alignment

It was clearly that registration by ICP resulted in unacceptable results, figure 12.1, this is not uncommon. Since 1992 much of the research have been made to guarantee the optimal registration result of ICP based algorithm, some of the research is found in literature as (Dorai et al., 1997; Chen et al., 1998; Sharp et al., 2002; Krishnan et al., 2005; Toldo et al., 2010). Simulated annealing is often used for finding a global extreme of a function, the paper by Jason P. Luck, William A. Hoff, Robert G. Underwood, and Hoff, Charles Q. Little (Luck et al., 2000) propose a method that combines ICP with simulated annealing, ensuring that the posses results in global minima.

A more practical approach to global registration, which is used in commercial software as Artec Studio and open source projects as CloudCompare, is to give user a possibility to initially align meshes manually. Given this initial alignment the ICP algorithm is set on the path to global minima.

13.2.2 Region Estimation

In this thesis we have implemented and tested two approaches to estimate the region in the model mesh that is represented by data shape. Both of the methods have proven to work, and both have features that can be considered as positive and negative. The spherical region estimation has proven to be more noise resistant by in the cost of including the neighbouring regions that are not part of the duplicated region. While the directional distance, in normal direction, has shown to estimate regions more precisely, but is more sensitive to noisy values.

A noisy vertex is a vertex that does not represent the surface of an object but was wrongly triangulated as so. Given that region estimation selects vertices V that are approximating the actual surface in the region and ignores noisy values that are not on the surface, after deletion of the vertex set V from the mesh noisy values will lose their connectivity to the surface, leaving them as individual points or as small patches of connected triangles. To increase the noise robustness of the region estimation, a small patch filtering scheme can be introduced. After the deletion of estimated region V, thereby interrupting connectivity between noisy vertices and the surface, the noisy values can be identified as patches of the mesh that are smaller than a certain threshold of connected triangles.

The threshold value needs to be selected carefully, to small and noisy regions are not deleted, to large and surface information is lost.

The small patch filter can be introduced into mesh marching pipeline after region estimation, and before boundary estimation.

13.2.3 Triangulation

When testing greedy triangulation on Nidaros stone set, the acceptable results were achieved, especially when the duplicated region X' was approximated by using directional distance. While triangulation in the desk scene test set did not succeed.

In section 13.1.3 we introduced a possible explanation for these different results of the triangulation algorithm.

To improve normal estimation function for handle rapid changes of surface normal in tightly populated neighbourhood we propose to alter the equation 10.1.

The normal weighting scheme is used to increase triangulations resistance towards noise and sudden changes in surface normal. In our implementation of greedy algorithm we use equation 10.1 to calculate the weighted normal, where ξ_i is the weight of individual normal based on distance between current point p_m and vertex p_i in the neighbourhood K, and d_{max} is the maximum allowed distance in knn-search. Let K be the neighbourhood of k nearest neighbours, with the maximum distance from p_m to a point in K denoted as d_k. If $d_{max} >> d_k$ then the weight of each neighbours normal ξ_i is less sensitive to distance variation $d_i \in \langle 0, d_k]$, than when distance $d_k = d_{max}$. Changing the equation 10.1 to

$$\xi_i = \frac{d_k}{d_i}, \tag{13.1}$$

will result in more sensitivity to individual distance in tightly populated neighbourhood.

This improvement is sensitivity will create a more adjustable weighting scheme and will be less computationally complex as the one proposed in (Marton et al., 2009).

By using the vertices to approximate a weighted plane as proposed in (Marton et al., 2009) the problem with appositive planes will be solved as normal information is ignored. This problem only occurs in specific type of situations where the object of interest could be scanned from every direction. This is not so when digitalising large objects as monuments and buildings.

Chapter 14

Conclusions and Future Work

In this chapter we draw conclusions about *MeshCombine* project, based on the experience and results from the tests. Based on the conclusions we recommend alternatives for future work and improvement of the *MeshCombine* tool.

14.1 Conclusions

In this master thesis we proposed, implemented and tested a tool for merging triangular meshes. Our approach does not rely on any information about capturing process and simply operates on meshes as they are without any additional information. All that is required is for a user to import two meshes, low precision model and high precision data of the same object, and roughly select the region where the data set is represented in the model, to start mesh combing algorithm which produces a combined mesh of model and data. To aligned meshes we use Iterative Closest Point algorithm proposed by Besl and McKay (Besl and McKay, 1992). For identify region of model mesh that is represented by the data we proposed two different approaches. We preserve continuity of the mesh by triangulating the gap, generated by removal of the duplicated region, by a greedy triangulation approach similar to the one described in (Marton et al., 2009). For rendering the information on the screen we used OpenGL as API.

To evaluate results we calculate square distance from resulting mesh to a high precision scan, which is treated as a ground truth for the testing.

ICP algorithm has proven to result in local minima, producing insufficient results,

and thereby is in need to be improved, by ether provide functionality to set initial alignment manually by user, or implement are more robust approach.

Region estimation has shown to be more precise when directional distance, in the direction of vertex's normal is used, than a spherical, Euclidian, distance. To increase resistance to noise we propose a small patch filter stage after region estimation.

Greedy triangulation results have proven to be acceptable, but not optimal, due the greedy nature of the algorithm.

14.2 Future work

As *MeshCombine* software only in the starting phase, there are many possibilities for improvement in the fields of, performance, user experience, and code readability of the program.

The results of the ICP algorithm has proven to be an acceptable, leading to necessity to ether implement possibility for manual initial registration, or use a more robust approach like the one proposed in (Luck et al., 2000). As the purpose for this software is to ease visualization/digitalization procedures, the automated approach should be considered before initial manual registration.

The region estimation approaches have proven to produce usable results, especially directional distance. To improve resistance of region estimation to noisy values in section 13.2.2 we propose a small patches filter, which will delete unconnected regions of few triangles. With increased removal of noisy regions, we recommend using directional distance in the direction of the normal as it proved to estimate the duplicated region more precisely then spherical distance, Euclidian distance.

Greedy triangulation has proven to function in certain circumstances, with few unwanted artifacts. Some improvements for weighting on normals are proposed in section 13.2.3. Greedy nature of the algorithm does not guarantee any qualities of the resulting mesh, if those qualities are required then a better algorithm should be implemented, the paper (Pito, 1996) propose a gap stitching algorithm between two triangular meshes, but this requires to identify mesh boundary in both meshes.

Appendices

Appendix A

Overview of MeshCombine Software

MeshCombine is written in C programming language, as a prototype for mesh merging tool. It consists of a small graphical user interface, and command line input/output. The graphics of *MeshCombine* are rendered by using OpenGL, with freeGLUT to contain control over rendering pipeline.

A.1 Linking and Compiling

The only external dependencies of *MeshCombine* are OpenGL API and FreeGLUT. Rest of the source code is written in C and follows with this master thesis. To compile the *MeshCombine* program the freeGLUT library must be installed on the computer.

OpenGL function access is different for different operating systems, since the gl3w.h is configured to operate on windows there no guarantee that the program will compile on LINUX or Mac OS without minor changes to OpenGL wrangler library. To achieve total compatibility the freeGLUT and OpenGL offers small OS dependent wrappers are needed.

The program is written, and compiled by in C79 dialect, which does not affect the possibility to compile the program in newer versions of C language.

All of the headers are protected by a pre-processor guard, so multiple includes stamens of the same file will not create conflicts.

A.2 Architecture

By using FreeGLUT the rendering pipeline follows MVC architectural pattern, A.1. After the program is invoked and the control-low enter main method, the freeGLUT is initiated, and main-loop is started. *GlobalVariables.h* can be looked upon as the model of the program which stores all required data for operation, meshes, orientation, and mouse and keyboard information. The draw function is view of the program, with extracts data from the model, *globalVariables.h*, and renders in appropriately. Keyboard and mouse functions serve in a role of controller, which changes data in the model and posts signal to glut for redisplay.

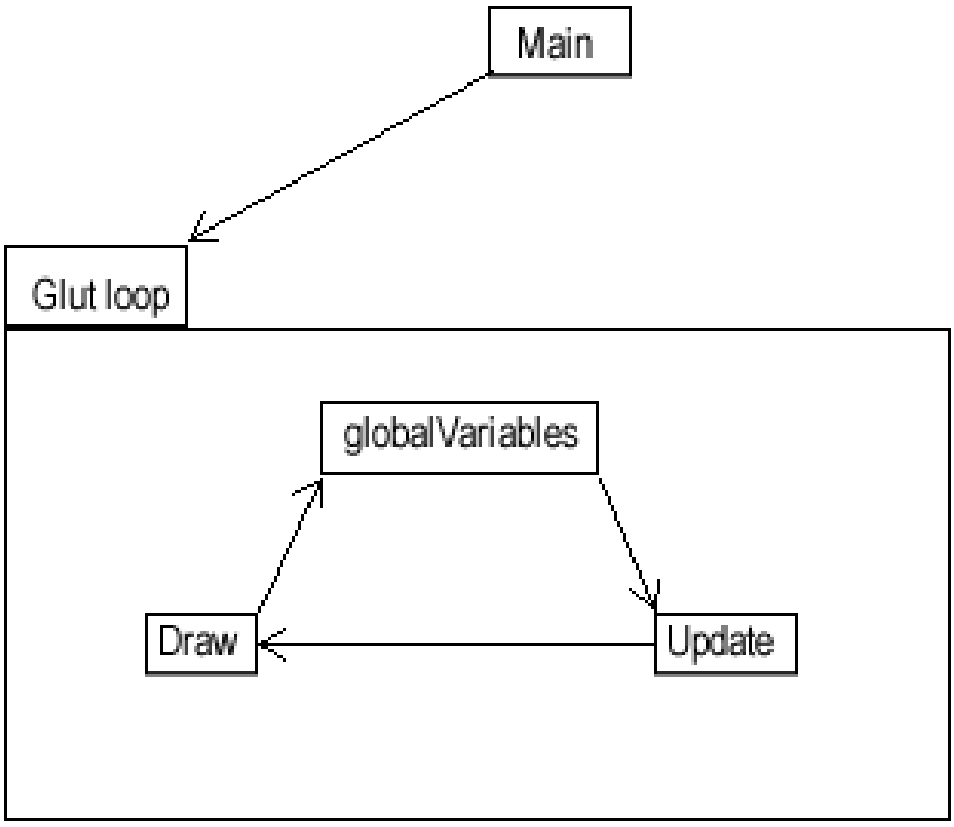

Figure A.1: Simplified model of *MeshCombine* displaying MVC architecture pattern.

A.3 Dependencies Graph

To give a better insight of how the *MeshCombine* functions we present the dependencies graph in figure A.2. The nodes in the graph represent a combination of .c/.h files. (not all .h files are accompanied by .c, to reduce amount of files needed). An arrow from node a to node b represent that node a uses (includes) node b in its source code.

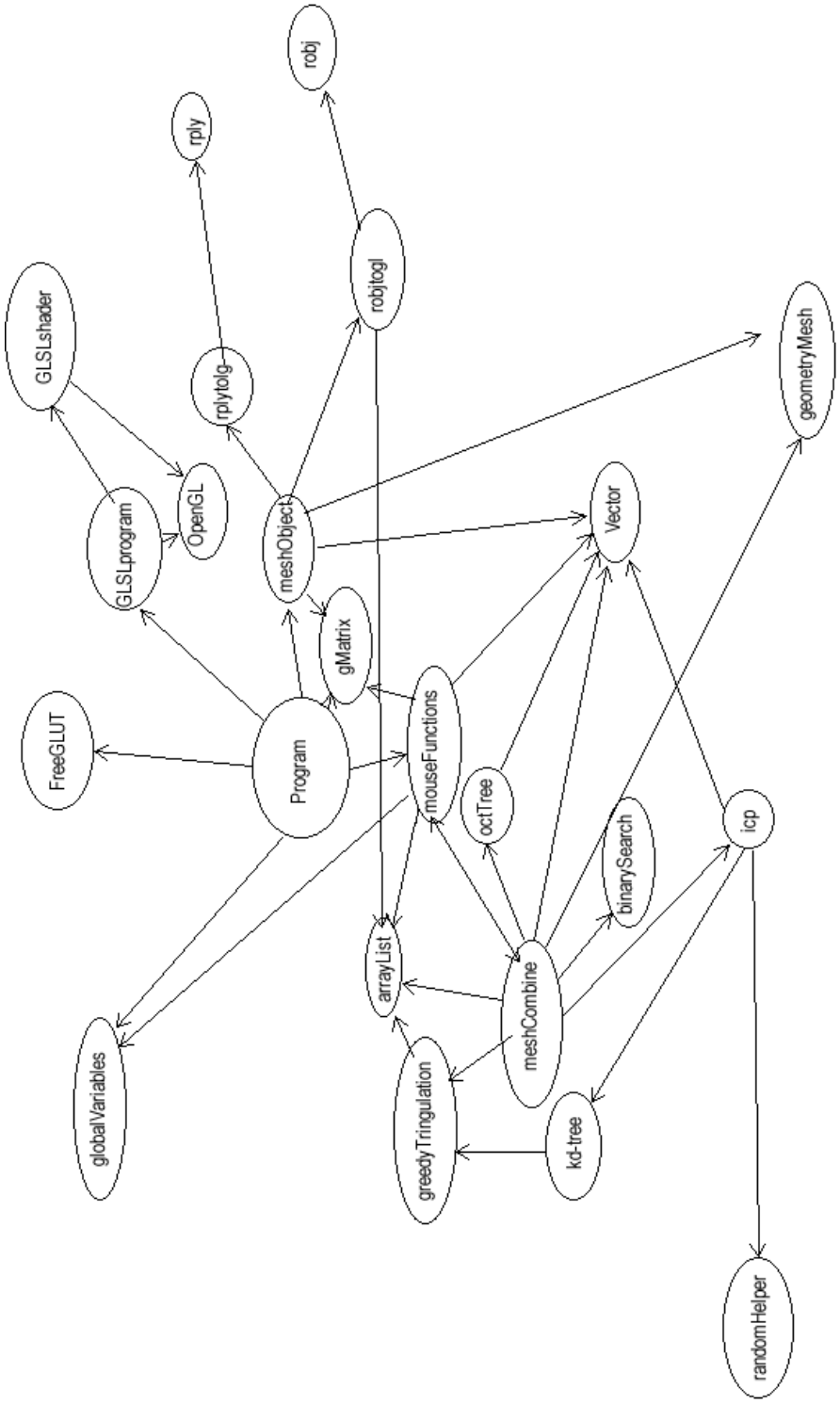

Figure A.2: Dependencies graph for *MeshCombine* implementation.

Appendix B

Octree Implementation

Octrees are used for their more structure determent structure. In comparison to balanced kd-trees there little overhead, to delete and add data to an octree. An advantage of an octree is that leaf nodes can also be looked upon as a grid system, finding the correct node for an input can be done directly.

The octree implementation in *octree.h/c* creates a regular octree, meaning that each node is divided in eight nodes with equal volume, this children creation does not guarantee that each node will contain same amount of data. For purpose of dividing a node in eight children a recursive function is used, and eight sub-functions that allocate, and fill a given children with corresponding values.

Most of the octree procedures are strait forward and does not require a special mention, except tree creation, addition of data, and trees deletion.

To create a dynamic octree that can contain any type of data the void pointer is used for data and the responsibility to create data addition function is given to the user of the library, in a form of function call-back. This is a well used strategy which is not new for most C programmers.

To elaborate on the matter, a pointer to data addition function is send as an argument when calling createOctree function. This function pointer is stored in *octTree_t* structure, which is opaque, meaning the user of the library don't have access to directly manipulate values (in object oriented programming langue this is similar to object with only private values). When it is required to add data the user specified function, stored as a function pointer is called with given data.

Similarly when deleting an octree extra precautions must be taken when deleting

data, since user can add any data to an octree node, user must also specify a deletion procedure. A pointer to octree deletion function is also send during the creation process.

An example from *MeshCombine* program can be found in *meshCombine.c*. Function *createOctTreeTrangulated* encapsulates the procedure and returns a pointer to newly created tree, functions *octTreeAddTriData* and *deleteDataFnc* are add and delete data functions.

octTreeAddTriData adds triangle information to octrees leaf nodes. Nodes are chosen based on the vertex information from vertex array. Both triangle index and vertex arrays must me send as a pointer to *octTreeAddTriData*.

deleteDataFnc simply deletes triangle information in an appropriate way.

Appendix C

KD-tree Implementation

Kd-tree is well known for logarithmic nearest neighbor search complexity. In *Mesh-Combine* software we used kd-trees to reduse complexity of ICP pair matching and k-nearest-neighbour search used in greedy triangulation.

Kd-trees are well known data structure and many libraries exist to create balanced kd-trees. Our implementation is based on pseudo code found on Wikipedia. We extended the implementation to include a knn-search function as well. The number of dimensions is specified in a header as a constant value *DIMENSIONS*.

The restricted knn-search implemented in *kdtree.c/h* is implemented in such a way that it will always return ordered list, from closest to farthest point of the length k, and is restricted by maximum distance. If there are less than k neighbours that are within the maximum distance, the return of the list will be order as follows: first part will contain pointers to all the neighbours with the distance to the target less that maximum in order previously mentioned, the rest of the list will contain *NULL* values.

Creation of the kd-tree is strait forward as this implementation only operates on float values. A tightly packed array of float is send as an argument of a function together with int value for size.

Appendix D

Dynamic List Implementation

A dynamic list or array is common in high level languages as C# or java. The lists grow as more elements are inserted into the list. Array list are useful when storing elements in unknown amount. For example when parsing an wavefront .obj file the amount of vertices in unknown, this can either be solved by "two-pass" when first time algorithm parse the file in counts the amount of objects, and second time it stores them in pre-allocated array for known size. Or dynamic arrays/lists can be used. Dynamic lists introduce a cost of reallocation, but reduce the complexity by one pass.

To create the dynamic list functionality in C we have implemented a library *arraList.h/c*. Two types of list have been implemented for our purpose, arrayListf and arrayListui for floats and unsigned integers respectably.

As overloading of function is not allowed in C the library follows a naming convention often used in libraries as OpenGL. The name of a structure of a function tell which type in operates on. For instance a function named *addToArrayListf* has an ending *'f'* for float, while *createArrayListui* has an ending *'ui'* for unsigned integer values. Suffix *'fv'* stands for float vector and *'uiv'* for unsigned integer vector, meaning that the function for example *addToArrayListuiv* takes as arguments list pointer that function operates on, array(vector) of unsigned integers values, and integer size that tell how many values the function shall add to given list.

The first argument on the function, except create function, is a pointer to the list itself, as C does not support an object oriented feature as keyword ***this***.

DeleteArrayListx function simply frees all the memory associated by provided pointer. After deletion memory allocated to the list structure have been freed

and because of that using a deleted list pointer after the deletion without changing the pointes value is there by illegal. Operating on the *arraylistx* structure that has been allocated without calling *createArrayListx* function is not recommended. Library does not guarantee correct execution, and specially *deleteArrayListx* function.

Appendix E

Using C Standard Libraries

The standard C libraries are the responsibilities of creators of the compilers. If standard libraries are available, it is recommended to use them. The standard libraries are often heavenly optimised, as macros and or using assembly commands directly.

We will not discuss all the available standard C functions as this is out of your scope, this chapter focus namely on one of them, *memcpy* from *string.h* header.

Memcpy function copies memory from one location in memory to another. This function is very effective to copy data from arrays as by using this function we allow the compiler to optimise the code based on the hardware architecture, and exploit all the power available of the hardware, in this context this would be a "perfect" word length to copy.

Instead of moving a value from one location to another, *memcpy* ignores the type of data stored in array, and simply copies bytes, as words of maximum length. Though knowing what a function in a standard library does exactly is not easy, as behaviour can change due different hardware and or compiler.

Through *MeshCombine* project we use *memcpy* function frequently to allow for maximum optimization.

Bibliography

ArtecGroup. Artec group, 2012. URL http://www.artec3d.com/.

G. Athanasios, B. Mathieu, and P.-D. Marc. An accuracy assessment of automated photogrammetric techniques for 3d modeling of complex interiors. *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, ISPRS Congress, 25 August - 01 September 2012,*, XXXIX-B3(XXII):23 – 8, 2012.

I. Autodesk. 3ds max, 2013a. URL http://www.autodesk.no/products/autodesk-3ds-max/overview.

I. Autodesk. Meshmixer, 2013b. URL http://www.meshmixer.com/.

R. Benjemaa and F. Schmitt. Fast global registration of 3d sampled surfaces using a multi-z-buffer technique. pages 113 – 20, Los Alamitos, CA, USA, 1997. URL http://dx.doi.org/10.1109/IM.1997.603856.

P. Besl and H. McKay. A method for registration of 3-d shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):239 – 56, 1992. ISSN 0162-8828. URL http://dx.doi.org/10.1109/34.121791.

H. Biermann, I. Martin, F. Bernardini, and D. Zorin. Cut-and-paste editing of multiresolution surfaces. volume 21, pages 312 – 21, USA, 2002.

F. Blais. Review of 20 years of range sensor development. *Journal of Electronic Imaging*, 13(1):231 – 43, 2004. ISSN 1017-9909. URL http://dx.doi.org/10.1117/1.1631921.

G. Blais and M. Levine. Registering multiview range data to create 3d computer objects. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17 (8):820 – 4, 1995. ISSN 0162-8828. URL http://dx.doi.org/10.1109/34.400574.

D. Bommes, B. Levy, N. Pietroni, E. Puppo, C. S. a, M. Tarini, and D. Zorin. State of the art in quad meshing. In *Eurographics STARS*, 2012.

A. Bradley and F. Stentiford. Jpeg 2000 and region of interest coding. pages 303 – 8, Clayton, Vic., Australia, 2002.

J. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25 – 30, 1965.

C.-S. Chen, Y.-P. Hung, and J.-B. Cheng. A fast automatic method for registration of partially-overlapping range images. pages 242 – 8, New Delhi, India, 1998. URL http://dx.doi.org/10.1109/ICCV.1998.710725.

Y. Chen and G. Medioni. Object modeling by registration of multiple range images. pages 2724 – 9, Los Alamitos, CA, USA, 1991. URL http://dx.doi.org/10.1109/ROBOT.1991.132043.

C. Christopoulos, J. Askelof, and M. Larsson. Efficient methods for encoding regions of interest in the upcoming jpeg2000 still image coding standard. *IEEE Signal Processing Letters*, 7(9):247 – 9, Sept. 2000. ISSN 1070-9908. URL http://dx.doi.org/10.1109/97.863146.

CloudCompare. Cloud compare, 2012. URL http://www.danielgm.net/cc/.

J. Daniels, C. Silva, J. Shepherd, and E. Cohen. Quadrilateral mesh simplification. *ACM Transactions on Graphics*, 27(5):148 (9 pp.) –, 2008. ISSN 0730-0301. URL http://dx.doi.org/10.1145/1409060.1409101.

A. Devrim, G. Armin, A. Zubeyde, D. Nusret, B. Bernd, E. Ilker, and N. Ender. 3d modeling of the weary herakles statue with a coded structured light system. *ISPRS Commission V Symposium 'Image Engineering and Vision Metrology'*, Commission V(WG V/2):14 – 19, 2006.

C. Dorai, J. Weng, and A. Jain. Optimal registration of object views using range data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(10): 1131 – 8, 1997. ISSN 0162-8828. URL http://dx.doi.org/10.1109/34.625115.

C. Dorai, G. Wang, A. Jain, and C. Mercer. Registration and integration of multiple object views for 3d model construction. *IEEE Transactions on*

Pattern Analysis and Machine Intelligence, 20(1):83 – 9, 1998. ISSN 0162-8828. URL http://dx.doi.org/10.1109/34.655652. multiple object views;3D model construction;range data;geometric models;robust estimation;registered data;unbroken surface;digital interferometric sensor;laser range scanner;.

EosSystems. Photomodeler scanner, 2012a. URL http://www.photomodeler.com/products/pm-scanner.htm.

EosSystems. Photomodeler tutorials videos, 2012b. URL http://www.photomodeler.com/tutorial-vids/online-tutorials.htm.

C. Fraser. A resume of some industrial applications of photogrammetry. *ISPRS Journal of Photogrammetry & Remote Sensing*, 48(3):12 – 23, 1993.

J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, Sept. 1977. ISSN 0098-3500. doi: 10.1145/355744.355745. URL http://doi.acm.org/10.1145/355744.355745.

A. Georgopoulos, C. Ioannidis, and A. Valanis. Assessing the performance of a structured light scanner. *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences, Commission V Symposium*, XXXVIII (part 5):250 – 55, 2010.

G. Godin, M. Rioux, and R. Baribeau. Three-dimensional registration using range and intensity information. volume 2350, pages 279 – 90, USA, 1994.

S. Gokturk, C. Tomasi, B. Girod, and C. Beaulieu. Medical image compression based on region of interest, with application to colon ct images. volume vol.3, pages 2453 – 6, Piscataway, NJ, USA, 2001. URL http://dx.doi.org/10.1109/IEMBS.2001.1017274.

N. Golias and R. Dutton. Delaunay triangulation and 3d adaptive mesh generation. *Finite Elements in Analysis and Design*, 25(3-4):331 – 41, 1997. ISSN 0168-874X. URL http://dx.doi.org/10.1016/S0168-874X(96)00054-6.

G. Golub and H. van der Vorst. Eigenvalue computation in the 20th century. *Journal of Computational and Applied Mathematics*, 123(1-2):35 – 65, 2000/11/01. ISSN 0377-0427. URL http://dx.doi.org/10.1016/S0377-0427(00)00413-1.

H. Gouraud. Continuous shading of curved surfaces. *IEEE Transactions on Computers*, C-20(6):623 – 9, 1971. ISSN 0018-9340.

D. D. Hearn, M. P. Baker, and W. R. Carithers. *Computer Graphics with OpenGL*. Prentice Hall Professional Technical Reference, 4 edition, 2010. ISBN 0132484579.

H. Hoppe. New quadric metric for simplifying meshes with appearance attributes. *Proceedings of the IEEE Visualization Conference*, pages 59 – 66, 1999.

X. Huang, H. Fu, O. K.-C. Au, and C.-L. Tai. Optimal boundaries for poisson mesh merging. pages 35 – 40, Beijing, China, 2007. URL http://dx.doi.org/10.1145/1236246.1236254.

IDI. Department of computer and information science, norwegian university of science and technology. URL http://www.idi.ntnu.no//.

Z. Jian-Ming, S. Ke-Ran, Z. Ke-Ding, and Z. Qiong-Hua. Computing constrained triangulation and delaunay triangulation: a new algorithm. volume 26, pages 694 – 7, USA, 1990. URL http://dx.doi.org/10.1109/20.106412.

A. Johnson and S. B. Kang. Registration and integration of textured 3-d data. In *IMAGE AND VISION COMPUTING*, pages 234–241, 1996.

KhronosGroup. Opengl faq, 2013a. URL http://www.opengl.org/wiki/FAQ.

KhronosGroup. Load opengl functions, 2013b. URL http://www.opengl.org/wiki/Load_OpenGL_Functions.

D. Kongevold. Comparison of structured light scanning and photogrammetric modelling techniques in preservation of historical artefacts, 2012. TDT4501 Specialization Project Thesis, Department of Computer and Information Science, Norwegian University of Science and Technology.

S. Krishnan, P. Y. Lee, J. B. Moore, and S. Venkatasubramanian. Global registration of multiple 3d point sets via optimization-on-a-manifold, 2005.

D. Lowe. Object recognition from local scale-invariant features. volume vol.2, pages 1150 – 7, Los Alamitos, CA, USA, 1999. URL http://dx.doi.org/10.1109/ICCV.1999.790410.

J. Luck, C. Little, and W. Hoff. Registration of range data using a hybrid simulated annealing and iterative closest point algorithm. volume vol.4, pages 3739 – 44, Piscataway, NJ, USA, 2000. URL http://dx.doi.org/10.1109/ROBOT.2000.845314.

Z. C. Marton, R. B. Rusu, and M. Beetz. On Fast Surface Reconstruction Methods for Large and Noisy Datasets. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Kobe, Japan, May 12-17 2009.

T. Masuda, K. Sakaue, and N. Yokoya. Registration and integration of multiple range images for 3-d model construction. volume vol.1, pages 879 – 83, Los Alamitos, CA, USA, 1996. URL `http://dx.doi.org/10.1109/ICPR.1996.546150`.

M. Minou, T. Kanade, and T. Sakai. A method of timecoded parallel planes of light for depth measurement. *Transactions of the IECE of Japan*, 64(8):521–528, 1981.

K. Morooka and H. Nagahashi. A method for integrating range images with different resolutions for 3d model construction. pages 3070 – 5, Piscataway, NJ, USA, 2006.

D. Nehab. Ansi c library for ply file format input and output. URL `http://w3.impa.br/~diego/software/rply/`.

PCL. Point cloud library, 2013. URL `http://pointclouds.org/`.

R. Pito. Mesh integration based on co-measurements. volume vol.2, pages 397 – 400, New York, NY, USA, 1996. URL `http://dx.doi.org/10.1109/ICIP.1996.560846`.

I. Pixologic. Zbrush, 2013. URL `http://pixologic.com/`.

K. Pulli, M. Cohen, T. Duchamp, H. Hoppe, J. McDonald, L. Shapiro, and W. Stuetzle. Surface modeling and display from range and color data. volume vol.1, pages 385 – 97, Berlin, Germany, 1997.

M. Qi, T.-T. Cao, and T.-S. Tan. Computing 2d constrained delaunay triangulation using the gpu. *IEEE Transactions on Visualization and Computer Graphics*, 19(5):736 – 48, 2013. ISSN 1077-2626. URL `http://dx.doi.org/10.1109/TVCG.2012.307`.

S. Rusinkiewicz and M. Levoy. Efficient variants of the icp algorithm. In *INTERNATIONAL CONFERENCE ON 3-D DIGITAL IMAGING AND MODELING*, 2001.

M. Rutishauser, M. Stricker, and M. Trobina. Merging range images of arbitrarily shaped objects. pages 573 – 80, Los Alamitos, CA, USA, 1994. URL `http://dx.doi.org/10.1109/CVPR.1994.323797`. range images;arbitrarily

shaped objects;shape descriptions;surfaces;graph surface;occlusion;depth images;sensor error model;.

J. Salvi, S. Fernandez, T. Pribanic, and X. Llado. A state of the art in structured light patterns for surface profilometry. *Pattern Recognition*, 43:713, 2010.

A. Sappa and M. Garcia. Incremental multiview integration of range images. volume vol.1, pages 546 – 9, Los Alamitos, CA, USA, 2000. URL http://dx.doi.org/10.1109/ICPR.2000.905396.

R. Schmidt and K. Singh. Drag, drop, and clone: An interactive interface for surface composition, 2010.

G. Sharp, S. Lee, and D. Wehe. Multiview registration of 3d scenes by minimizing error between coordinate frames. pages 587 – 97, Berlin, Germany, 2002.

O. Sorkine, D. Cohen-Or, Y. Lipman, M. Alexa, C. Rossl, and H.-P. Seidel. Laplacian surface editing. volume 71, pages 175 – 184, Nice, France, 2004. URL http://dx.doi.org/10.1145/1057432.1057456.

StanfordUniversity. The stanford 3d scanning repository, 2012. URL http://graphics.stanford.edu/data/3Dscanrep/.

R. Toldo, A. Beinat, and F. Crosilla. Global registration of multiple point clouds embedding the generalized procrustes analysis into an icp framework, 2010.

G. Turk and M. Levoy. Zippered polygon meshes from range images. pages 311 – 18, New York, NY, USA, 1994.

J. Wei and Y. Lou. Feature preserving mesh simplification using feature sensitive metric. *Journal of Computer Science and Technology (English Language Edition)*, 25(3):595 – 605, 2010. ISSN 1000-9000. URL http://dx.doi.org/10.1007/s11390-010-9348-7.

S. Weik. Registration of 3-d partial surface models using luminance and depth information. pages 93 – 100, Los Alamitos, CA, USA, 1997. URL http://dx.doi.org/10.1109/IM.1997.603853.

P. Will and K. Pennington. Grid coding: a preprocessing technique for robot and machine vision. *Artificial Intelligence*, 2(3/4):319 – 29, Winter 1971. ISSN 0004-3702.