



TAMPEREEN TEKNILLINEN YLIOPISTO

**ANTTI POHJOLA**  
**DENSE 3D MODELS FROM A SINGLE RGB-D VIEW**  
Master of Science thesis

Examiners: prof. Tommi Mikkonen  
& prof. Joni-Kristian Kämäräinen  
Examiners and topic approved by the  
Faculty Council of the Faculty of  
Computing and Electrical Engineering  
on June 3<sup>rd</sup>, 2015

# ABSTRACT

**ANTTI POHJOLA:** Dense 3D Models From a Single RGB-D View

Tampere University of Technology

Master of Science Thesis, 54 pages

January 2016

Master's Degree Programme in Information Technology

Major: Software Engineering

Examiners: Professor Joni-Kristian Kämäräinen and Professor Tommi Mikkonen

Keywords: 3D, Surface Reconstruction, Point Clouds, Range Imaging

The increase in processing power of modern computers and mobile devices has enabled the use of three-dimensional (3D) visual data with a nearly realistic level of detail in various applications. This development combined with an increasing interest in technologies such as Virtual Reality and 3D printing has created a need for 3D data acquired from real-life scenarios.

The aim of this work is to examine hardware and software needed to create a 3D model from a simple real-life scenario, and to produce a tool that can perform all the necessary processing steps. The goal is to use affordable hardware and open source software supported by multiple platforms. In this work the Microsoft Kinect is used to capture a Point Cloud representation of various scenarios and objects. This Point Cloud data is improved with multiple processing algorithms and reconstructed as a 3D model with different Surface Reconstruction methods. The results are evaluated by the processing time needed and the quality and usability of the virtual 3D model.

There is a lot of interest and research on the subject so finding tools and methods is not a problem, but selecting those most suitable can be difficult. We find out that there are multiple affordable hardware systems in the market and they are supported by a wide selection of open source software. The methods for Point Cloud processing and Surface Reconstruction are also numerous and well investigated. The Kinect was selected to be used with the Point Cloud Library (PCL) software which had ready implementations for all needed methods and algorithms. The results show that it is possible to use these affordable tools to generate dense 3D models from a single RGB-D view. Some environment decisions, such as using Windows instead of Linux, caused some compatibility issues with the libraries used and the selected methods do not work on all real-life surfaces. The Command Line Interface (CLI) software produced in this work requires further development to be more usable, such as better exception handling and a Graphical User Interface (GUI), but it functions well as a proof-of-concept. The results show that a good quality outcome is achievable with reasonable processing time, but getting the best outcome does require some prior knowledge of the theories behind the methods.

# TIIVISTELMÄ

**ANTTI POHJOLA:** Tiheä 3D malli yksittäisestä RGB-D näkymästä

Tampereen Teknillinen Yliopisto

Diplomityö, 54 sivua

Tammikuu 2016

Tietotekniikan diplomi-insinöörin tutkinto-ohjelma

Pääaine: Ohjelmistotuotanto

Tarkastajat: professori Joni-Kristian Kämäräinen ja professori Tommi Mikkonen

Avainsanat: 3D, Pinnan uudelleenrakennus, Pistepilvet, Etäisyydenmittaus

Kasvu nykytietokoneiden ja mobiililaitteiden prosessointitehoissa on mahdollistanut kolmiulotteisen (3D) visuaalisen tiedon käytön erilaisissa sovelluksissa lähes realistisilla yksityiskohdilla. Tämä kehitys yhdistettynä kasvavaan kiinnostukseen teknologioita, kuten Virtuaalitodellisuus ja 3D-tulostus, kohtaan on luonut tarpeen 3D tiedolle, joka kerätään todellisista skenaarioista.

Tämän työn tavoitteena on tutkia laitteita ja ohjelmia joita tarvitaan 3D mallin luomiseen yksinkertaisista todellisista skenaariosta, ja tuottaa työkalu joka voi suorittaa kaikki tähän tarvittavat vaiheet. Tarkoituksena on käyttää edullisia laitteita ja avoimen lähdekoodin ohjelmistoja jotka tukevat useita alustoja. Tässä työssä käytetään Microsoft Kinectiä pistepilven tallentamiseen erilaisista skenaarioista ja esineistä. Tätä pistepilveä parannetaan useilla prosessointialgoritmeilla, ja se rakennetaan uudelleen 3D mallina erilaisilla pinnanrakennus-metodeilla. Tuloksia arvioidaan vaaditun prosessointiajan ja tuotetun 3D mallin laadun sekä käytettävyyden perusteella.

Aiheesta on paljon tutkimustietoa ja mielenkiintoa joten työkalujen ja metodien löytäminen ei ole ongelma, mutta parhaiten sopivien valinta on vaativaa. Selvitämme että markkinoilla on olemassa monia edullisia laitteita, ja niitä tukee laaja valikoima avoimen lähdekoodin ohjelmia. Pistepilven prosessointi- sekä pinnan uudelleenrakennusmetodeja on myös paljon ja niitä on tutkittu laajasti. Kinect valittiin käytettäväksi Point Cloud Library (PCL) ohjelmiston kanssa jossa oli valmiit toteutukset kaikille vaadittaville metodeille ja algoritmeille. Tulokset näyttävät, että on mahdollista käyttää näitä edullisia työkaluja tiheän 3D mallin tuottamiseen yksittäisestä RGB-D näkymästä. Jotkut päätökset ympäristön suhteen, kuten Windowsin käyttö Linuxin sijaan, aiheuttivat yhteensopivuusongelmia käytettyjen kirjastojen kanssa ja valitut metodit eivät toimi kaikkien todellisten pintojen kanssa. Tässä työssä tuotettu komentorivikäyttöliittymällä (CLI) toimiva ohjelma vaatii jatkokehitystä kuten paremman poikkeuskäsittelyn ja graafisen käyttöliittymän (GUI) ollakseen paremmin käytettävä, mutta se toimii hyvin ratkaisun todennuksena. Tulokset näyttävät, että laadukas lopputulos on saavutettavissa järkevillä prosessointiajoilla, mutta parhaaseen tulokseen pääseminen vaatii ymmärrystä metodien taustalla olevista teorioista.

## **PREFACE**

This Master's thesis was written for the Tampere University of Technology.

First I wish to thank Prof. Joni-Kristian Kämäräinen from the Vision Group at the Department of Signal Processing at Tampere University of Technology for providing me with the subject and guidance along the way. Finding an interesting subject enabled me to finally finalize my studies after a lengthy delay.

I would also like to express my appreciation for Prof. Mikko Tiusanen and Prof. Tommi Mikkonen for swift feedback and comments on this thesis.

Tampere, 20.01.2016

Antti Pohjola

# CONTENTS

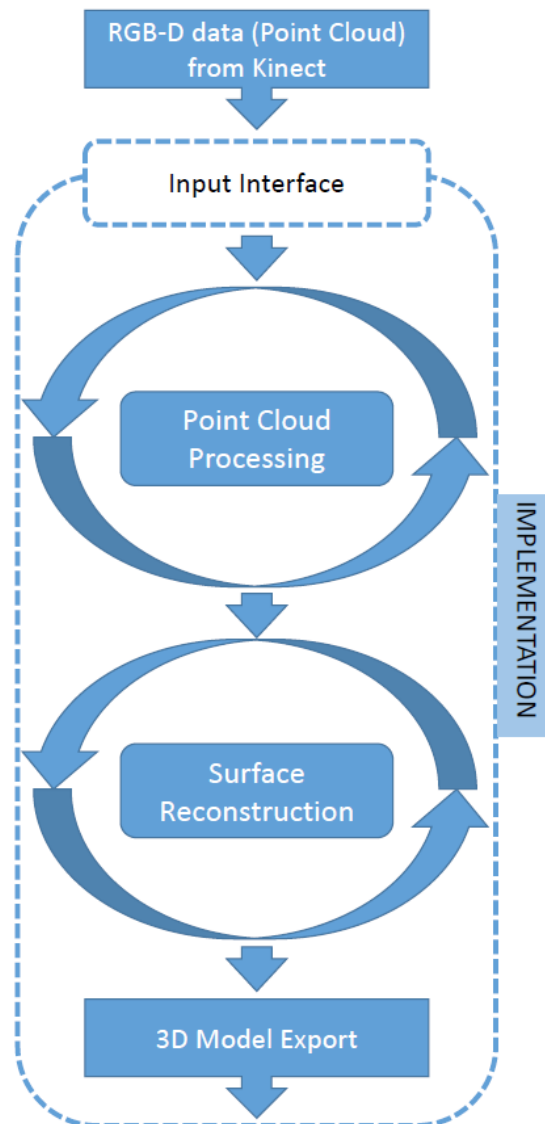
1	Introduction.....	1
2	Combined Color and Depth (RGB-D) imaging.....	4
2.1	Structured Light.....	4
2.2	Time-Of-Flight.....	7
2.3	Other Techniques.....	9
2.4	Summary.....	12
3	Reconstruction of Surfaces from Point Clouds.....	13
3.1	Auxiliary Methods.....	13
3.1.1	Delaunay triangulation .....	13
3.1.2	K-dimensional tree .....	14
3.2	Point Cloud Processing.....	15
3.2.1	Up-sampling and down-sampling .....	16
3.2.2	Moving Least Squares .....	16
3.3	Surface Reconstruction.....	17
3.3.1	Alpha shapes.....	18
3.3.2	Power Crust.....	20
3.3.3	Marching Cubes.....	21
3.3.4	Organized Fast Mesh Construction.....	21
3.3.5	Greedy Projection Triangulation.....	22
3.3.6	Grid projection.....	22
3.3.7	Poisson Surface Reconstruction.....	23
3.4	Available Software.....	23
3.4.1	Point Cloud Library (PCL).....	24
3.4.2	Open Natural Interaction (OpenNI).....	25
3.4.3	Other software.....	26
3.4.4	3D Model Export Formats.....	26
3.5	Summary.....	27
4	Implementation.....	28
4.1	Environment.....	28
4.2	Data capture setup.....	29
4.3	Software implementation.....	30
5	Evaluation.....	43
5.1	Implementation.....	43
5.2	Performance.....	44
5.3	Technical limitations.....	45
5.4	Further development.....	47
6	Conclusions.....	49
	References.....	50

## LIST OF SYMBOLS AND ABBREVIATIONS

<b>3D</b>	Three Dimensional
<b>API</b>	Application Programming Interface
<b>ASCII</b>	American Standard Code for Information Interchange
<b>CLI</b>	Command Line Interface
<b>CMOS</b>	Complementary-Metal-Oxide-Semiconductor
<b>GUI</b>	Graphical User Interface
<b>IR</b>	Infrared
<b>ISO</b>	International Organization for Standardization
<b>LED</b>	Light-emitting Diode
<b>MAT</b>	Medial Axis Transform
<b>MLS</b>	Moving Least Squares
<b>NaN</b>	Not a Number
<b>NUI</b>	Natural User Interface
<b>PLY</b>	Polygon File Format
<b>RGB-D</b>	Red Green Blue and Depth
<b>SDK</b>	Software Development Kit
<b>SL</b>	Structured Light
<b>ToF</b>	Time of Flight
<b>USB</b>	Universal Serial Bus
<b>VR</b>	Virtual Reality
<b>VRML</b>	Virtual Reality Modeling Language
<b>VTK</b>	Visualization Tool Kit
<b>X3D</b>	Extensible 3D Graphics
<b>XML</b>	Extensible Markup Language

# 1 INTRODUCTION

3D imaging means capturing and analyzing three-dimensional (3D) information from a real-life scenario, for example, measuring the distance and posture of a person waving a hand. The result of 3D imaging is RGB-D data that combines a regular red-green-blue (RGB) image with depth (D) data. The demand for 3D imaging comes from the increasing supply of new products like 3D printers, Natural User Interface (NUI) devices, and Virtual Reality (VR) to consumer markets. 3D printing as a technology has been around for decades, but with the improvements in technologies and techniques it has become affordable and more easily available for consumer and industrial use [1]. 3D printing has the potential to revolutionize manufacturing as regular consumers gain access to powerful production tools [2]. The competition on the VR market is intense with many big technology companies and newcomers introducing new devices such as the Oculus rift [3], Google Glass [4], Samsung Gear VR [5], and Apple with its own still unnamed VR product in the near future [6]. VR devices offer a seemingly endless list of potential uses from games and interactive movies to rehabilitation therapy [7][8] and visual assistance [9]. NUI devices like the Microsoft Kinect offer an interface to operate other devices or software with intuitive use of hands, movement, or voice commands without any training required. 3D imaging is also used extensively in entertainment in the form of motion capture for movies [10]. RGB-D data acquired through 3D imaging can also be combined with various processing methods to produce a 3D model from a real-life object. These 3D objects have a multitude of different uses, for example, they can be exported for use in computer gaming or 3D animation. With this rising demand and potential for 3D imaging it is interesting to examine the methods and hardware available for consumers at the moment, and see what types of results are achievable with them. Acquiring RGB-D data from a real-life scenario and processing it into a better 3D representation for further use is a complex process consisting of multiple steps illustrated in Figure 1.

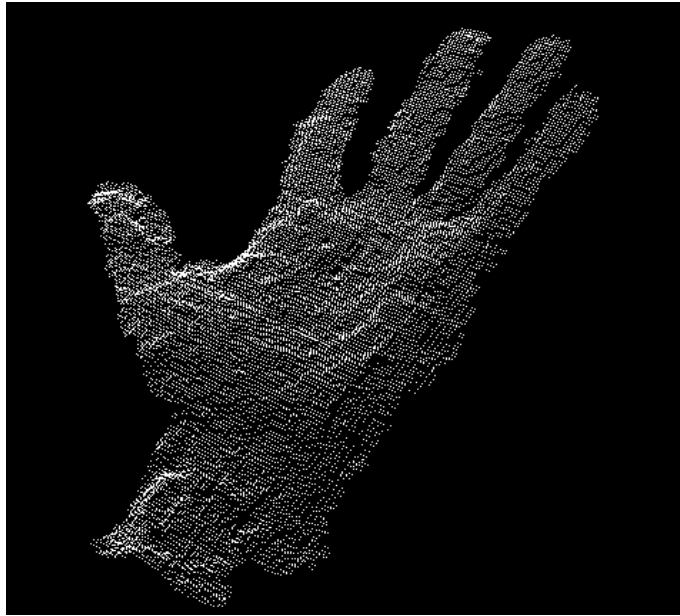


*Figure 1: Process Flow of the thesis.*

The first step is acquiring the RGB-D data through RGB-D imaging. What devices are most easily available for consumers and what level of accuracy do they provide? The hardware should be easily available and affordable. The RGB-D data will be acquired from a single image covering only one angle, which places limitations on the quality of the outcome.

The next step is figuring out how to get the data from the device to the development environment through an input interface, and what tools to use for processing it. All software used should be open source and have support for multiple platforms. In this thesis the RGB-D data will be processed as a point cloud (Figure 2), which is a data structure that represents 3D data as a collection of data points.





*Figure 2: A point cloud visualization of a hand.*

The acquired point cloud will go through different processing steps to remove unnecessary data and prepare it for the surface reconstruction phase. Surface reconstruction means building a surface that best represents the original real-life surface from the point cloud. The aim is to use surface reconstruction methods to produce a dense 3D model of the scene. What methods produce the best results? What kind of processing do the methods require? The methods should have reasonable processing times on an average desktop computer, and the results should provide a good 3D representation of the processed scene. The last step is exporting the produced model in a 3D data format for potential further usage.

The objective of this thesis work is to combine the 3D imaging with further processing methods into a tool which can perform all the steps described above. The theory will cover hardware used to capture the RGB-D data, software for importing this data, and then examine the methods used to process this raw data into a usable 3D model.

**Structure of the thesis.** The first chapter provides the motivations, goals, and restrictions for the thesis, and general view of the process of generating a dense 3D model from a single RGB-D view. In the second chapter common hardware and techniques behind capturing the RGB-D data are examined. The third chapter goes through the different methods and software used to process the captured RGB-D data into a 3D model. The fourth chapter describes the environment used to produce and use the implemented tool, and the chapter provides a step-by-step walkthrough of using the tool. The results are evaluated in chapter five. Chapter six provides a conclusion of the work done.

## 2 COMBINED COLOR AND DEPTH (RGB-D) IMAGING

RGB-D imaging is the process of capturing both the Red-Green-Blue (RGB) color and the depth (D) data from a scene. In this chapter we will cover different techniques used to capture the depth data and the hardware that uses these techniques.

### 2.1 Structured Light

Structured Light (SL) is a process of projecting a known light pattern, normally infrared (IR) or laser, on a surface and analyzing the changes in the captured projection caused by the surface. A normal setup projects the pattern to the scene from an IR source and has a sensing device detecting the projected pattern from a slightly different angle (Figure 3).

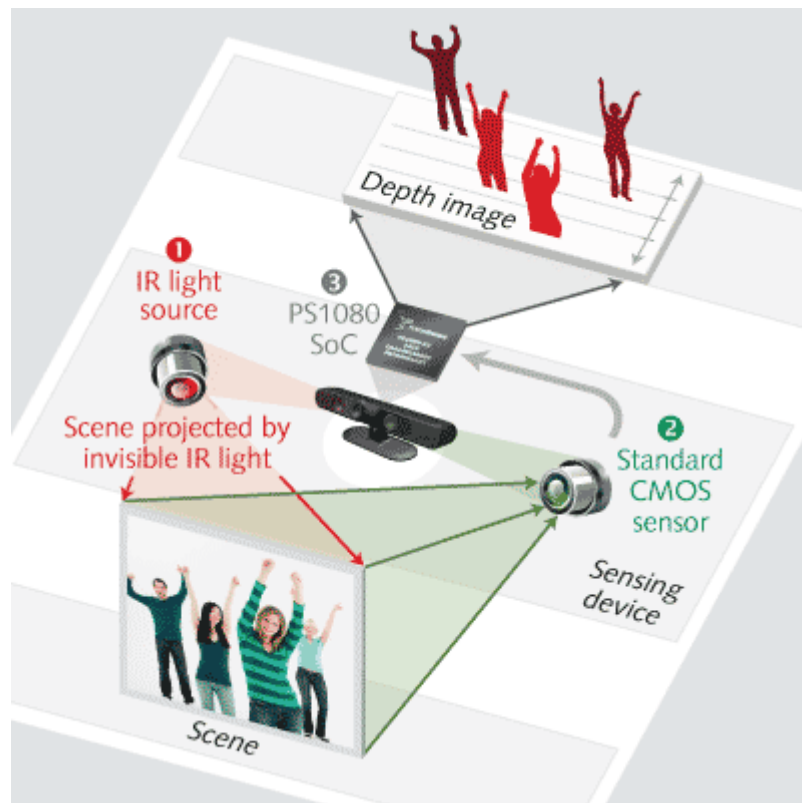


Figure 3: A structured light setup with Kinect [11].

The most common type of sensor used today is a CMOS (Complementary-Metal-Oxide-Semiconductor) sensor that is also used in many digital cameras since it is inexpensive to manufacture and uses less electrical power than other sensor types [12]. The type of the pattern projected from the light source varies. Commonly used patterns include stripes (Figure 4), grids, and speckles (Figure 5).

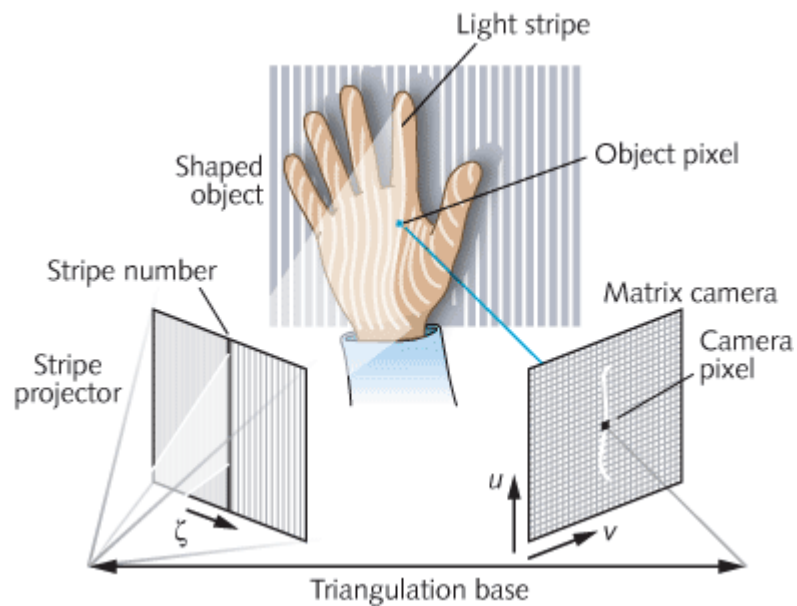


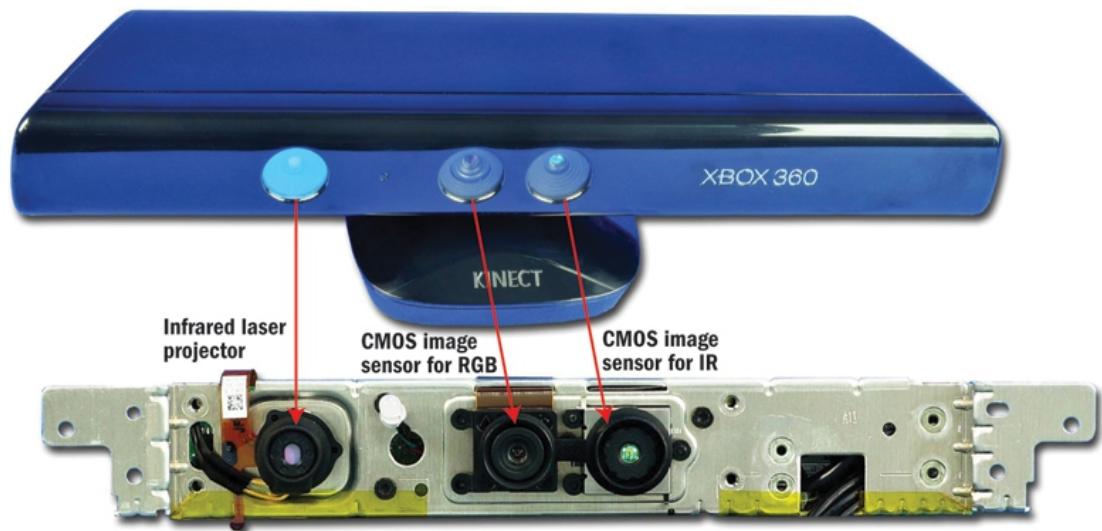
Figure 4: Illustration of a striped projection on a surface [11].



Figure 5: Speckle type projection in Kinect [13].

The next steps are to capture the projected pattern to a sensor, match the pixels from the original projected image to the captured image, and calculate the changes in geometry. Calculating the changes in geometry is done by measuring the observed changes in the projected patterns, e.g. the curvature of projected straight lines (Figure 4) or the distortion of projected speckles.

**Microsoft Kinect for Xbox 360.** Microsoft launched Kinect in 2010 for the Xbox 360 video game console with range imaging technology from an Israeli 3D sensing company PrimeSense [14]. It offers the user a Natural User Interface (NUI), which is a human-machine interface that hides complex operations behind intuitive actions. A NUI is supposed to be easy to learn to use without any specific training. The Kinect (Figure 6) offers this interface through a motion sensing device, a normal camera, and a microphone.



*Figure 6: Kinect Hardware [15].*

The motion sensing device consisting of the IR projector and the IR image sensor can track the hands of any users and use hand gestures as commands. An example is swiping left or right to turn a page, or making a circle with your finger that can be interpreted in multiple ways. The normal RGB image sensor in the device helps with the motion sensing and can be used as a web camera for human interaction or used to identify individual users. The microphone is used for voice commands, human interaction, or voice recording.

The IR image sensor developed by PrimeSense projects light close to the infrared range, so it works even without external light sources and does not distract the user since the light is not visible to the human eye. The precise process Kinect uses has not been made completely public, but it is known that Structured Light with a speckled projection (Figure 5) and a combination of other methods (such as Depth From Focus and Depth From Stereo) is used to calculate the depth [16].

**The Structure Sensor** [17] started from a crowd-funding project in 2013 with a goal to develop a consumer grade sensor and SDK for 3D sensing and surface reconstruction. The Structure Sensor is developed by Occipital. The Structure Sensor is primarily developed to be used with mobile devices and is currently available for Apple products such as the iPhone 6 and iPad Air 2 (Figure 7).

The hardware specifications are similar to the Kinect for Xbox 360 and it also uses structured light with the same resolution. Occipital has released a full SDK supporting the sensor for iOS developers, with many ready APIs, such as a regular 3D scanner, software called Room Capture for scanning bigger spaces and software called Fetch to be used with augmented reality gaming. The Structure Sensor is also fully compatible with the OpenNI 2 SDK for developing software on other platforms.



*Figure 7: The Structure Sensor attached to an Apple iPad [17].*

## 2.2 Time-Of-Flight

Time of Flight (ToF) can be used to describe many different methods that calculate the flight time of sound waves, particles, or radiowaves, but in this thesis ToF is used with light pulses. ToF using light is a relatively new technique that shows great promise in depth imaging. The process works much like a regular radar, with the exception that a light pulse is used instead of a radio wave. This process has not been used until recently since it requires advanced hardware and special LEDs or lasers. Now available for civil applications, ToF cameras are becoming much more common. These are quicker and smaller than SL systems, so they are more suitable for mobile use.

The process consist of projecting one or more light pulses to the scene, and then capturing the reflected light with a sensor (Figure 8). The time light takes to reflect back to the camera differs slightly depending on the distance of the object. Naturally, with small distances (not inter-planetary) the time differences will be minuscule.

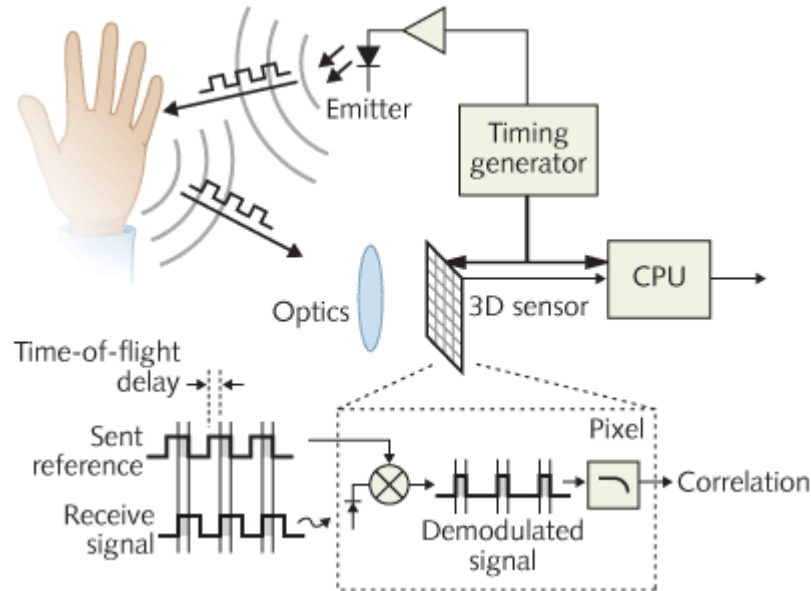


Figure 8: A Time-Of-Flight device setup [11].

With an approximate light speed of  $c = 300,000,000$  m/s, reflecting the light from an object  $D = 3$  meters away will take time calculated in Equation 1.

$$delay = 2 \frac{D}{c} = 2 \frac{3}{300\,000\,000} = 0.000\,000\,01\,s = 10\,ns \quad (1)$$

With delays this small, calculations become challenging. One way to measure the time of flight is by sending the light signals (Figure 8) at precise intervals and capturing them with synchronized intervals [18]. Calculating the time between these intervals is then used to calculate the distance.

**Microsoft Kinect for Xbox One** (Figure 9) was launched in 2014 with the new generation Microsoft Xbox One. This had 2 different models, the Xbox One and a Kinect for Windows, primary difference being the USB adapter and that the Windows version was optimized for closer range usage. Microsoft decided to discontinue the Kinect for Windows in 2015 and released a USB 3.0 adapter for the Xbox One Kinect [19].

This second generation system has major improvements over the first generation Kinect [20], including a 1080p camera, better infrared capabilities, and a greatly improved depth sensor. The sensor on the new Kinect utilizes the ToF method. The new technology allows the Kinect to track up to 6 simultaneous users, even their heart rates [21].



Figure 9: Microsoft Kinect for Xbox One.[22]

### 2.3 Other Techniques

While Structured Light and Time-of-Flight are the main techniques used to capture depth data in depth imaging devices like the Kinect, also other supporting techniques can be used to provide depth information.

**Depth from Focus** relies on the fact that objects further away from the sensor lens Object point (Figure 10), which is the point the lens is focused on, are more blurry than objects close to the point.

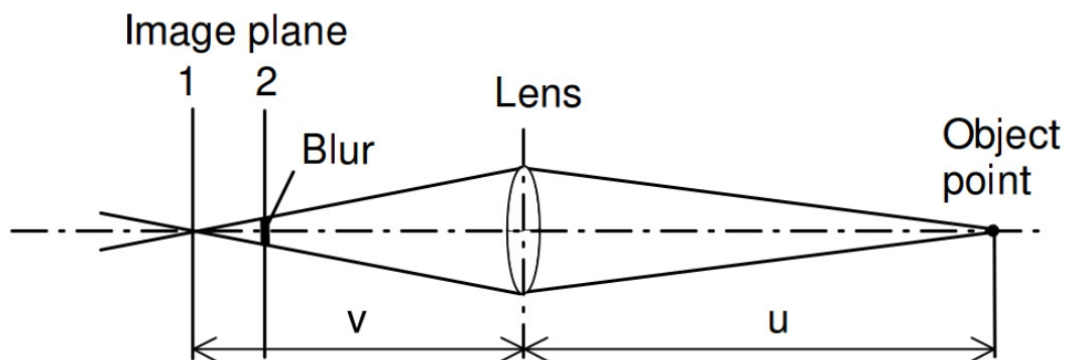


Figure 10: Measuring blur [23].

To use Depth from Focus efficiently, the camera will normally use a very small depth of field. Depth of field is the range around the lens focus point where the objects appear acceptably sharp (Figure 11).

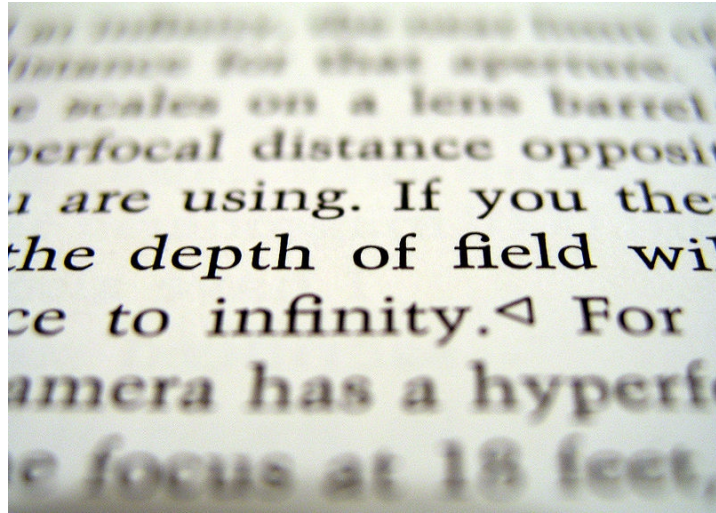


Figure 11: An image with a very small depth of field [24].

Using a relatively small depth of field the objects on the image plane in Figure 10 will have negligible focal blur, and the objects further or closer will be increasingly blurred. The distance of the object  $u$  can be easily calculated if the camera focal length  $f$  and the image plane distance from the lens  $v$  (Figure 10) are known with Equation 2.

$$\frac{1}{f} = \frac{1}{u} + \frac{1}{v} \quad (2)$$

Measuring the depth estimates for the out-of-focus distances require a model to calculate the amount of blur in relation to the distance.

**Depth from Astigmatic Difference** utilizes the properties of an astigmatic optical lens. This is like the human eye, a lens with two different focal planes on different axes with different curvatures (Figure 12). Astigmatism is also the term for a common vision defect that makes your vision blurry and difficult to focus on a single object. This defect can be corrected with glasses or contact lenses, but it can also be used to measure depth. Without astigmatism, the Tangential Focal Plane and Sagittal Focal Plane (Figure 12) would be at the same distance and changes in the pattern geometry would not occur. If a circular pattern is projected on a surface, it will deform in different ways (Figure 13) depending on the distance of the surface from the lens with an astigmatic difference. The changes in the geometry of the pattern can then be calculated and used to estimate depth.



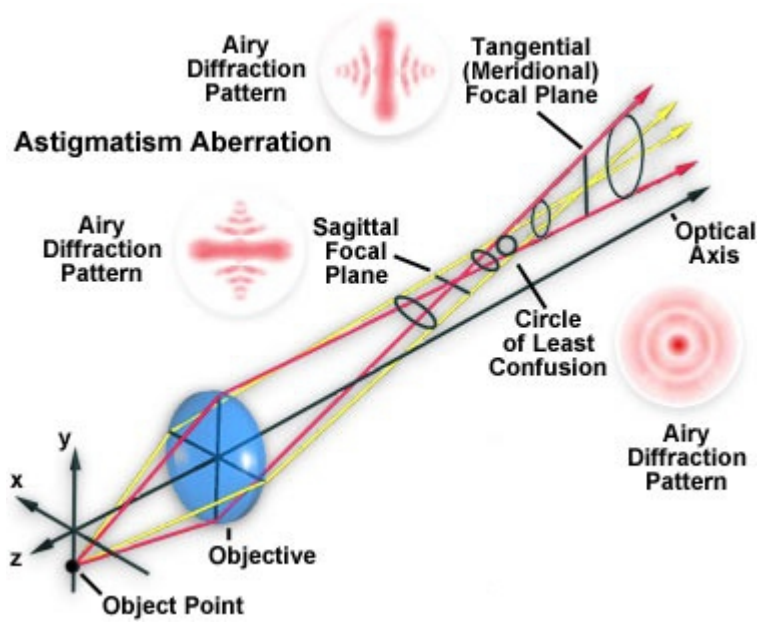


Figure 12: Astigmatism [25].

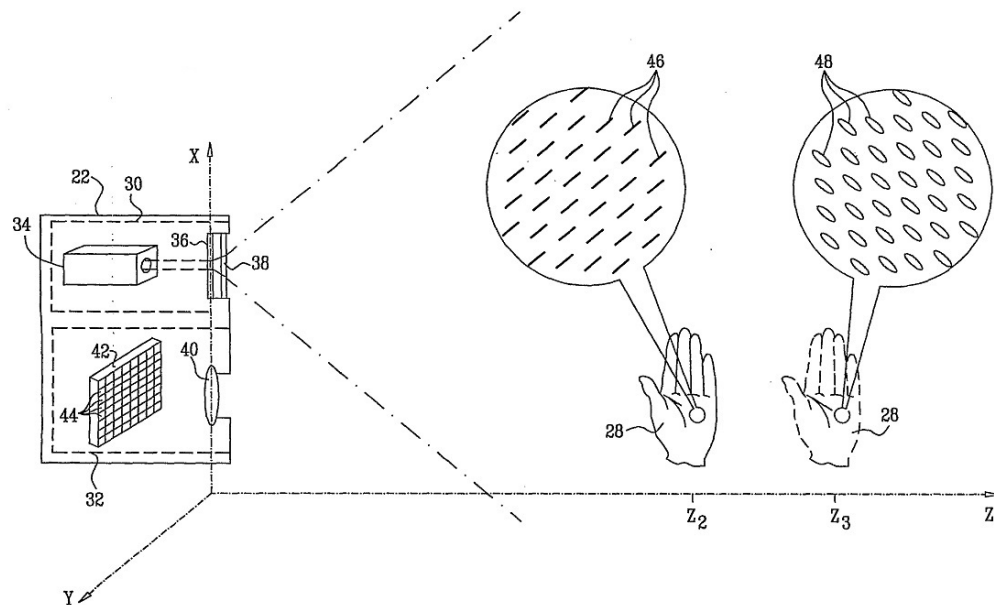


Figure 13: Ellipse distortion from an astigmatic lens with Kinect [26].

In **Depth From Stereo** two slightly different angles (Figure 14) of two viewpoints (e.g. two cameras or left and right eye) cause slight disparities between the two images and these disparities can be used to measure depth.

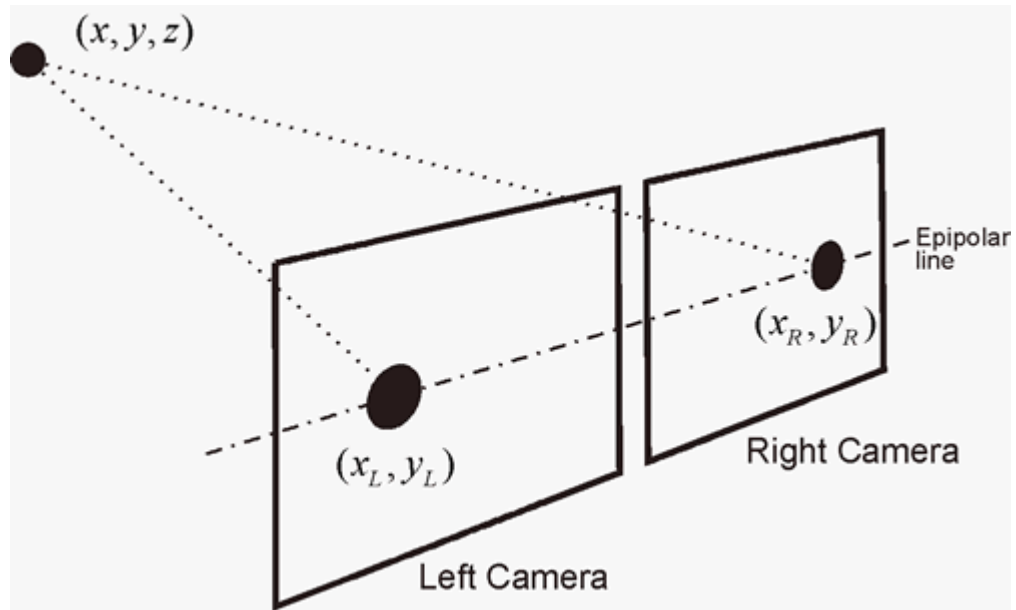


Figure 14: Stereo Vision System [27].

The first problem with using 2 slightly different images of the same scene is matching the points from each image to the other, i.e. the Correspondence Problem, and there are numerous methods to solve it [28]. The simplest methods just calculate squared differences in pixel intensities or other properties easily obtained from the images. These have problems with occlusions that hide points from one image and not the other, and repetitive patterns in the images that make it hard to differentiate similar points. The best solutions are dynamic algorithms that take into account all the different scenarios that can appear in natural environments. Another major problem with systems based simply on Depth from Stereo and regular cameras is that since only visible light is used the lighting conditions must be optimal. Low contrast situations can also prove to be difficult.

## 2.4 Summary

This chapter covered different techniques used to capture the depth data and the hardware that uses these techniques. The focus was on the Structured Light and Time of Flight techniques and the Microsoft Kinect product family. In this thesis we use the Structured Light technique with the Microsoft Kinect for Xbox 360, since it was easily available, is used a lot in similar research, and the accuracy the Kinect with SL provides is enough for the needs of this work.

## 3 RECONSTRUCTION OF SURFACES FROM POINT CLOUDS

A point cloud captured with the Kinect is raw data with anomalies caused by the limitations of the used hardware and methods, and the point cloud is difficult to read visually since it does not have any solid surfaces or shapes. In this chapter we will examine the process of transforming this point cloud to a better visual representation by improving the data with processing algorithms and building the surfaces with different surface reconstruction methods. We will also cover the software used to implement this process.

### 3.1 Auxiliary Methods

The methods of reconstructing the surfaces from point clouds utilize different algorithms to organize and restructure the data. In this section we will examine some of those methods.

#### 3.1.1 Delaunay triangulation

Delaunay triangulation [29] is used to create triangles from the points in the point cloud, which can then be used for the surface polygon mesh. A polygon mesh is a collection of polygons, which in turn are collections of points (called vertices) and line segments (called sides) connecting these points [30]. Delaunay triangulation uses circles to try connect all points of the point set by using three points at a time (Figure 15). From these circles that have three points on their circumsphere we select the ones that do not contain any other points. These spheres form the triangulation basis, and there are multiple ways to decide which spheres to use for optimal triangulation. A Voronoi diagram [31] is a plane divided to sections with a single points as seeds (Figure 16). The sections are called Voronoi Cells, and contain all points closer to the seed point than the seed point of the surrounding cells. A Voronoi diagram can be used to divide points in a point cloud to neighborhoods.

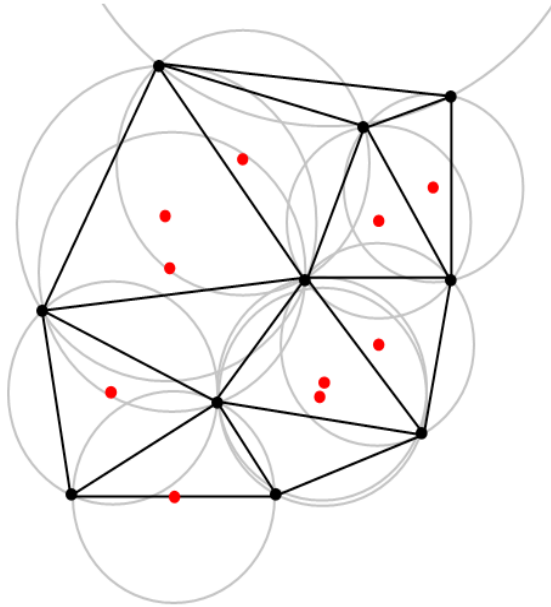


Figure 15: Delaunay triangulation. Circle centers marked with red dots, point set with black dots.

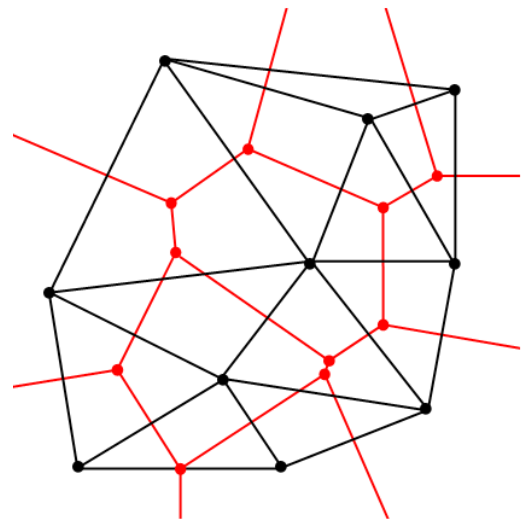


Figure 16: Voronoi diagram (red lines) produced by connecting the Delaunay triangulation circle centers.

### 3.1.2 K-dimensional tree

K-dimensional tree (kd-tree) is used in many of the point cloud processing and surface reconstruction methods to order the points to search through them quickly. Kd-tree is an extension of a binary search tree (Figure 15) to organize points in k-dimensions. A binary search tree operates by connecting a single node to 2 nodes. The nodes on the left side must always have a lesser value than the nodes on the right.

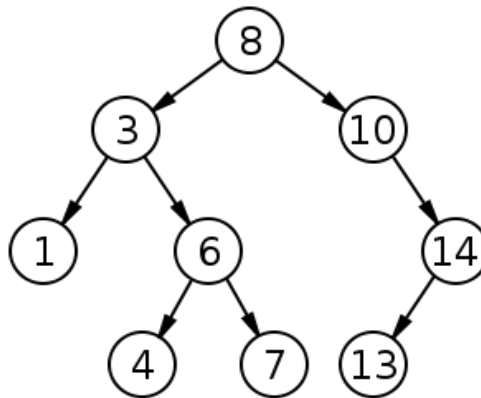


Figure 17: A binary search tree [32].

When used in 3 dimensions the tree divides the space (Figure 18) by always adding a plane to split it in two. The left and right parts are represented as the sub spaces of the original space. The process is continued until each space only contains 1 point. The 3-dimensional tree is a very efficient way of finding nearest neighbors of points and is extensively used in surface construction algorithms.

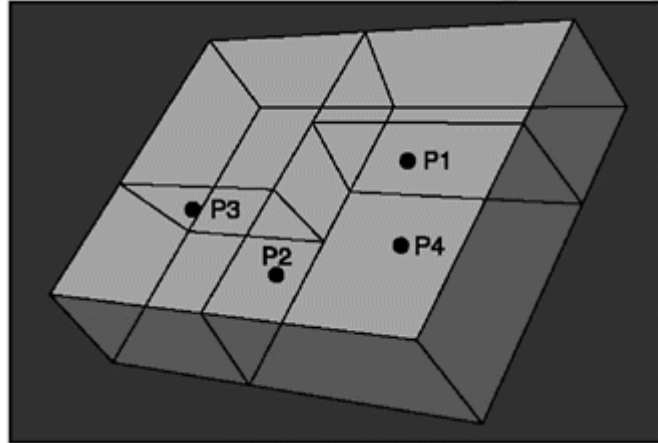


Figure 18: Visualization of a cube divided to smaller cubes by the 3-dimensional tree algorithm [33].

### 3.2 Point Cloud Processing

Capturing the point cloud will always produce data anomalies. The amount of anomalies will depend largely on the device and method used. Frequent anomalies are listed in Figure 19:

- Regular noise caused by lighting conditions, movement, or inefficient equipment.
- Misalignment and distortion caused by orientation and occlusion.
- Alignment errors in depth data retrieved from systems with multiple points-of-view or multiple scenes.
- Varying density in captured data points.
- Missing data points, caused by occlusion, lighting conditions, or technical issues.

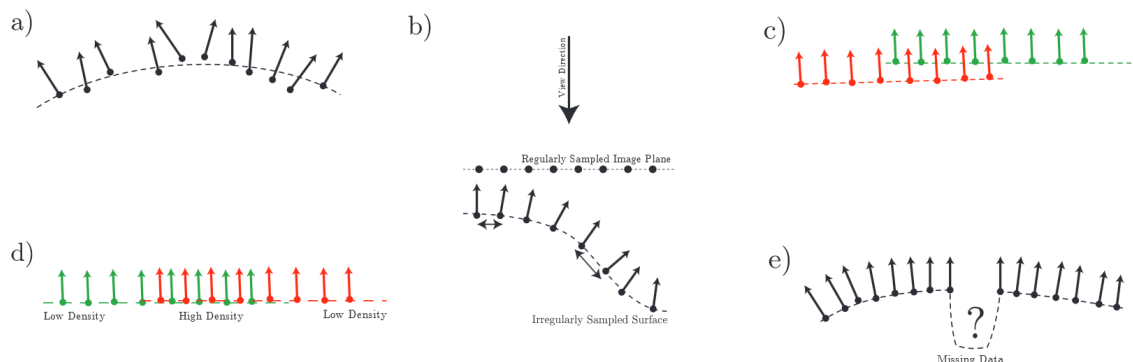


Figure 19: Common data anomalies in depth data [34].

In this section we will examine common algorithms used for point cloud processing.

### 3.2.1 Up-sampling and down-sampling

The number of points in a given point cloud is not always optimal. Too many points provide no additional information because of noise or inaccuracy of the capture; too few points prevent the use of some surface reconstruction methods. The process of artificially increasing or decreasing the amount of points in the point cloud is called up-sampling or down-sampling. Adding or removing random points is not of course an option, so there are many methods to achieve a desirable outcome. One down-sampling method uses volume pixels (voxels) to create a grid (Figure 20) over the given data points. Once the grid has been created the down-sampling algorithm approximates a point in the middle of each voxel and removes all other points inside the voxel. This way we get a point cloud with less points ins close to regular distances. The cloud will have less noise, more consistency and will be easier to process for many surface construction methods.

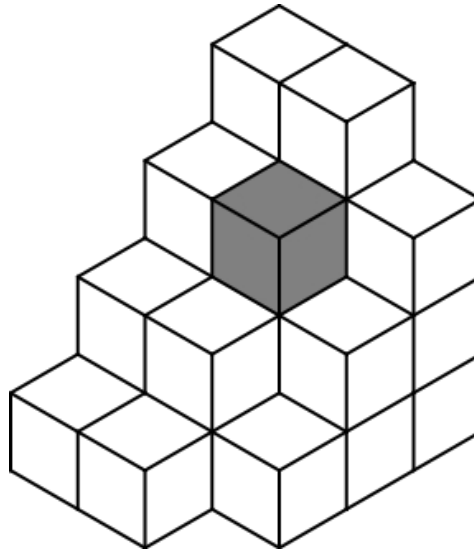


Figure 20: A voxel grid with a single voxel in gray color [35].

### 3.2.2 Moving Least Squares

The Least Squares method is one of the simplest methods for fitting a function to a set of points (Figure 21). The procedure is done by minimizing the sum of the squares of the point offsets [36]. This ordinary least squares method assumes that the offsets of the points have a constant variance, so in any practical situations the method has to be extended. Weighted Least Squares method adds a weight to each of the offset errors so you can either lessen the impact of large errors or make them more visible in the result. This way the method can adapt to different situations with varying amounts of error. Moving

Least Squares (MLS) [37] method applies the weighted least squares locally to each point that requires smoothing or re-sampling. It can be applied to every point of the set. MLS is good for re-sampling noisy data and smoothing out flat surfaces (Figure 22).

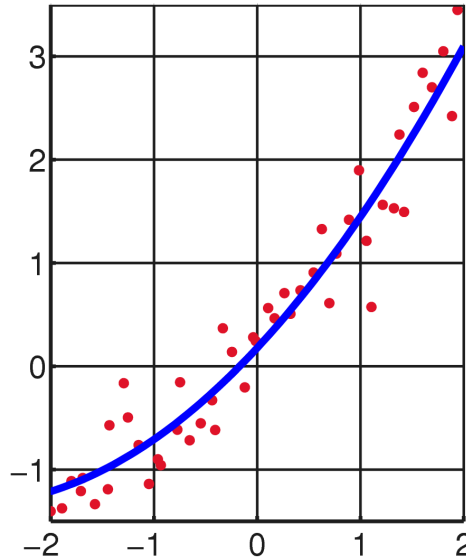


Figure 21: A quadratic function fitted on a set of data

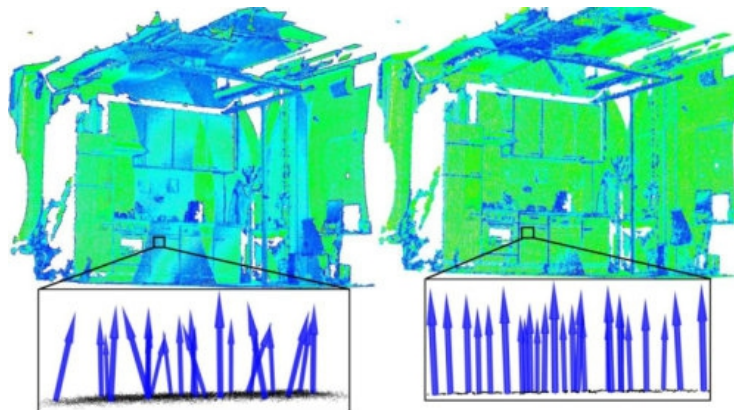


Figure 22: MLS applied to a noisy point set with a sample of the smoothed normals [38].

### 3.3 Surface Reconstruction

In this section we will examine methods for surface reconstruction.

### 3.3.1 Alpha shapes

A Convex Hull is possibly the most general shape that can be extracted from a set of three-dimensional points (Figure 23). It forms a simplified surface representation that encloses all the points of the set. A convex hull in itself is already useful for some cases, such as object collision detection or other applications that require fast calculations and only need the external surfaces. However, a hull does not create a realistic representation of a 3D object because it hides all non-convex (or concave) shapes of the object. A better model can be created by extracting the alpha shape of the point set [39]. Getting the alpha shape starts with fitting the space with spheres with a radius  $\alpha$  that touch the points on their circumference but do not enclose any (Figure 24), and then removing these spheres. The resulting object could already be called an  $\alpha$ -hull, but if we substitute straight edges for circular curves and triangles for the spherical caps (with Delaunay Triangulation) we get a more sensible shape (Figure 25). Creating the model with a fixed radius causes problems with complex shapes that have different sized cavities and sharp shapes, but some of these problems can be solved by using a weighted radius [40].

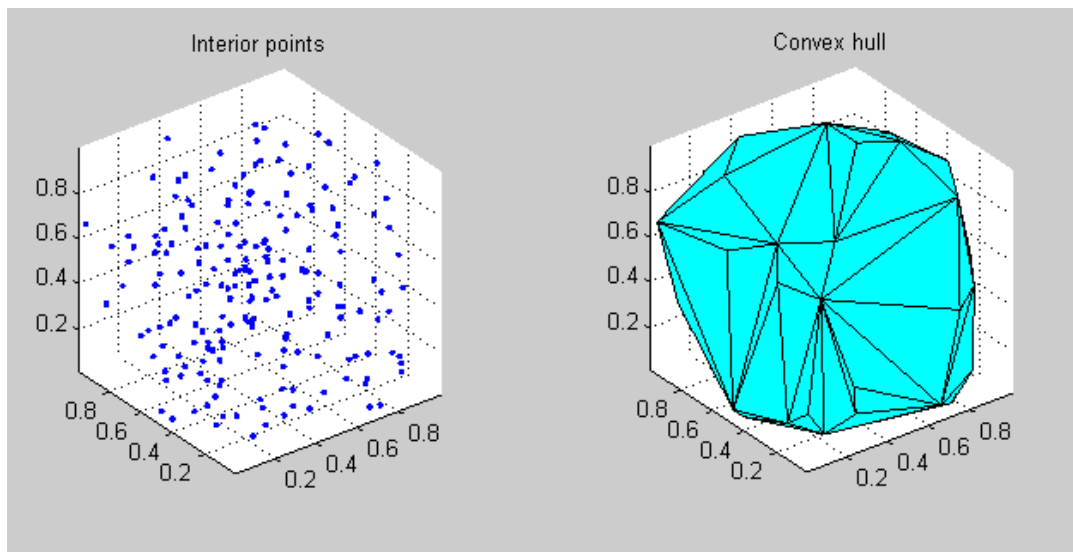


Figure 23: Convex Hull from a set of 3D points



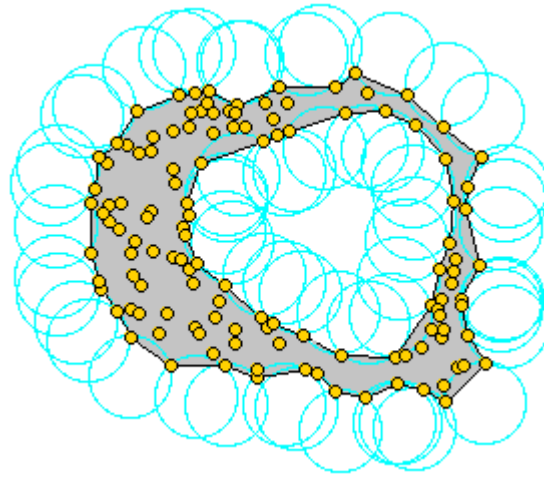


Figure 24: 2D example of an alpha shape with circles that do not enclose any points

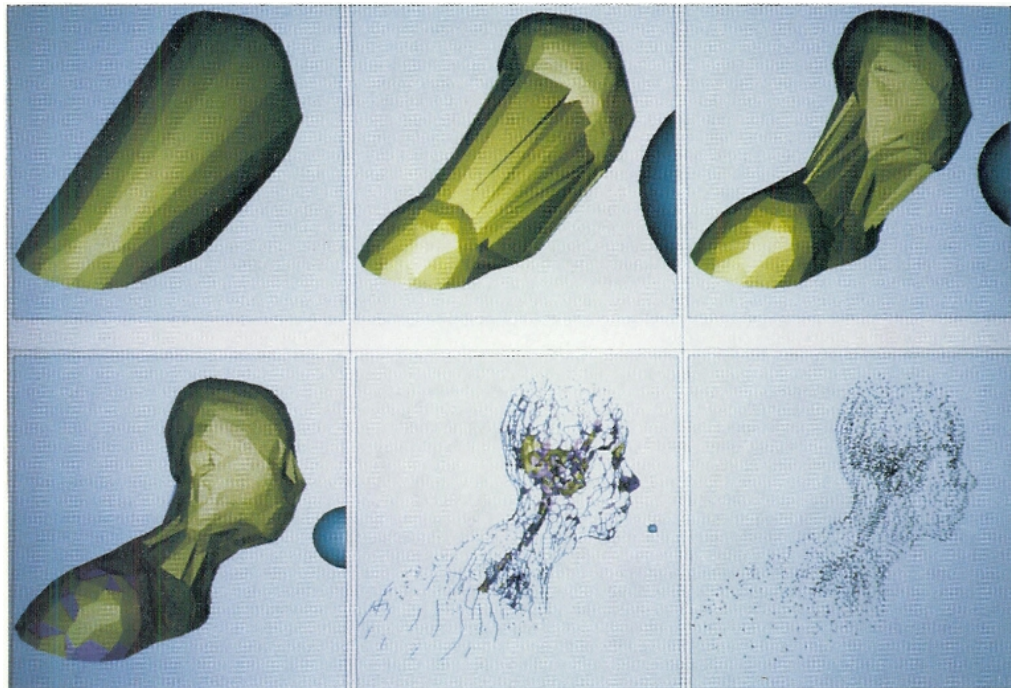


Figure 25: Alpha hull model with different sphere radius  $\alpha$ . Top left is a convex hull with  $\alpha = \infty$ , and bottom right the point set with  $\alpha = 0$  [39].

### 3.3.2 Power Crust

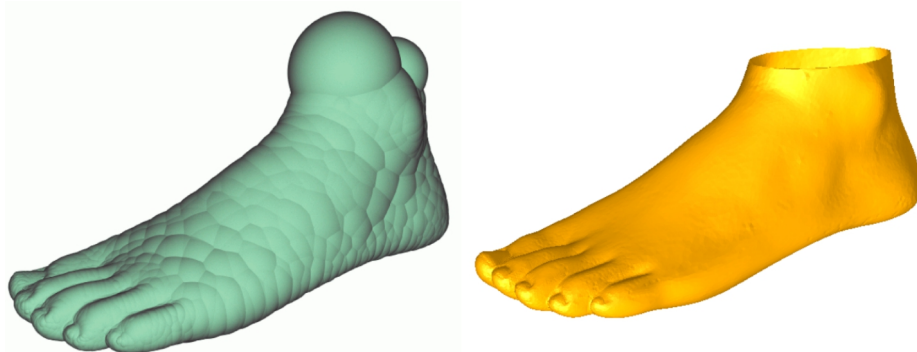
The Power Crust algorithm [41] computes the surface shape around a solid object and also calculates the Medial Axis. A Medial Axis (or a Topological skeleton) is a shape that is equally distant from all the surfaces of the object. A Medial Axis can be very useful in object recognition and identifying gestures. The process of finding the medial axis is called Medial Axis Transform (MAT).

Many of the ideas behind Power Crust are based on weighted  $\alpha$ -shapes, including the ball shapes used. Power crust is implemented by first performing the MAT. This is done by fitting the object with maximally large balls that have the surface points on their circumference. The center points of these balls form the topological skeleton (Figure 26).



*Figure 26: Point set on the left, model produced by power crust in the middle and the approximated medial axis on the right [41].*

Once the MAT is done and we have the balls inside the object, we generate analogous balls to the outer surface of the object. The surface is estimated between these outer and inner balls which a good model of the object already, but by smoothing the surface we get rid of the unnecessary round surface shapes (Figure 27). The Power Crust always produces a “watertight” object meaning that it is solid without cavities and the surface is fully connected.



*Figure 27: The model produced by the inner and outer balls and the smoothed out model. [41]*

### 3.3.3 Marching Cubes

Developed already at 1987 [42], The Marching Cubes algorithm was published in a paper that is one of all time most cited in computer graphics. It is one of the earliest 3D surface construction algorithms and was originally developed for medical 3D scans. Marching Cubes algorithm divides the 3D plane to 8 neighboring points, or cuboids, and “marches” through them one-by-one. Then we determine for each cuboid which corners (or vertices) of the cube are inside the surface and which are outside. Once we know these points, we can form the facets by connecting the surrounding edge indices (Figure 28).

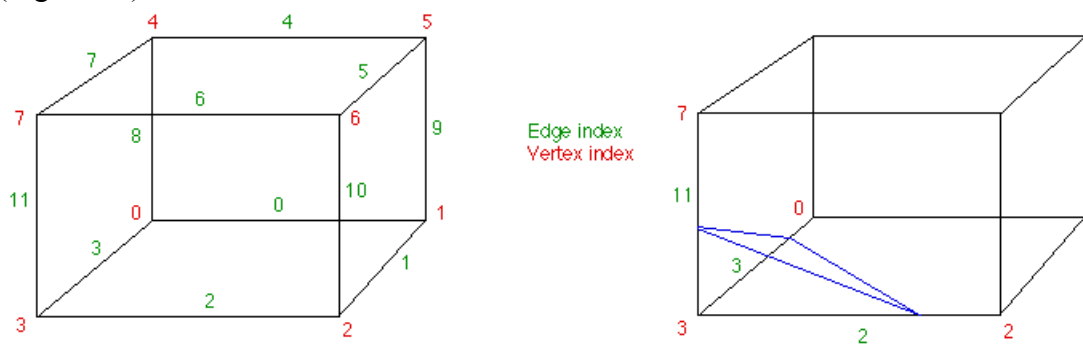


Figure 28: A cuboid on the left and the surface facet created by vertex 3 being "inside" the surface [43].

The performance of the algorithm can be adjusted by determining the appropriate cube size (Figure 29). Smaller size cubes produce more accurate models but, as always, it also takes more time to compute.

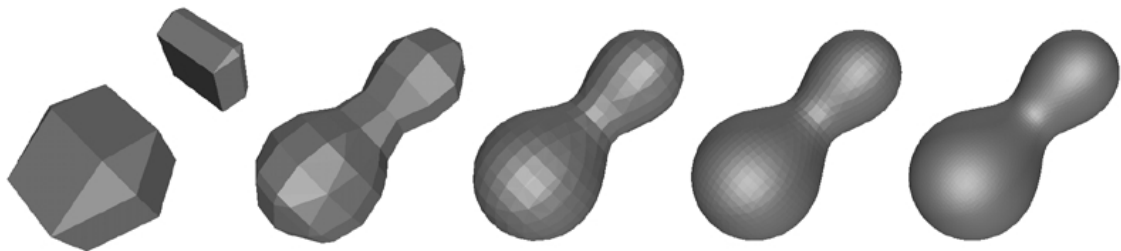


Figure 29: Marching cubes performance increasing with a decreasing cube size.[43]

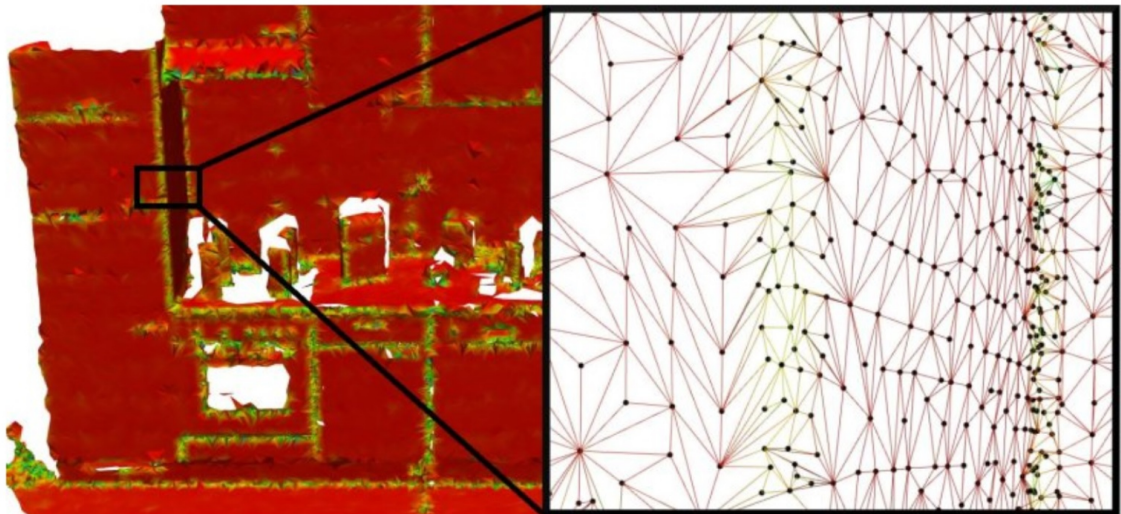
### 3.3.4 Organized Fast Mesh Construction

The Organized Fast Mesh is a fast method of constructing a surface of a point cloud [44]. It requires that the point cloud data is structured, which means that each point in the depth data has an XYZ-coordinate and all coordinates in the defined point space have a value (the value can be undefined or “NaN” as used in computing). The data can either come structured directly from the device used to capture it, or the structure can be

computed using a 3-dimensional search tree. The structured data is used to approximate local neighborhoods for the points. The approximation is done by checking the closest neighboring points of every point in the cloud, and if the all points are valid (not “NaN”) and there are no edges between the points the neighborhood is valid. The edges are detected by having certain maximum tolerances for the angles and lengths between points. The algorithm then creates a rough polygon mesh by connecting these valid neighborhoods and approximates normals and curvatures. It then uses up-sampling methods to smooth out the point surfaces and does a rough segmentation.

### 3.3.5 Greedy Projection Triangulation

The Greedy Projection Triangulation algorithm for surface reconstruction [45] uses triangulation to calculate a surface (Figure 30) for the given point cloud that can be unorganized and noisy. It starts by selecting some key points on the surface edges and using triangulation to connect the points in the points neighborhood until all points are connected. The greediness comes from the fact that once an edge is created by this algorithm it is never deleted or altered. This makes the algorithm very fast and memory-efficient, but also quite inaccurate and sensitive to anomalies.



*Figure 30: Triangulation of points around and edge. [45]*

### 3.3.6 Grid projection

The Grid Projection [46] method starts by identifying the edges of the surfaces in the point cloud that has unoriented or oriented normals. It then creates a grid from these edges with a polygonization routine using the Dual Contouring algorithm [47] until it can connect all the edges.

### 3.3.7 Poisson Surface Reconstruction

Poisson Surface Construction method creates a watertight surface from a point cloud [48]. The method uses the Poisson equation (Equation 3).

$$\nabla^2 \varphi = f \quad (3)$$

which in three-dimensional coordinates the equation takes the format in Equation 4.

$$\left( \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \right) \varphi(x, y, z) = f(x, y, z) \quad (4)$$

In the equation  $\varphi$  is the vector field formed from the point normals and  $f$  is the function that estimates the surface.

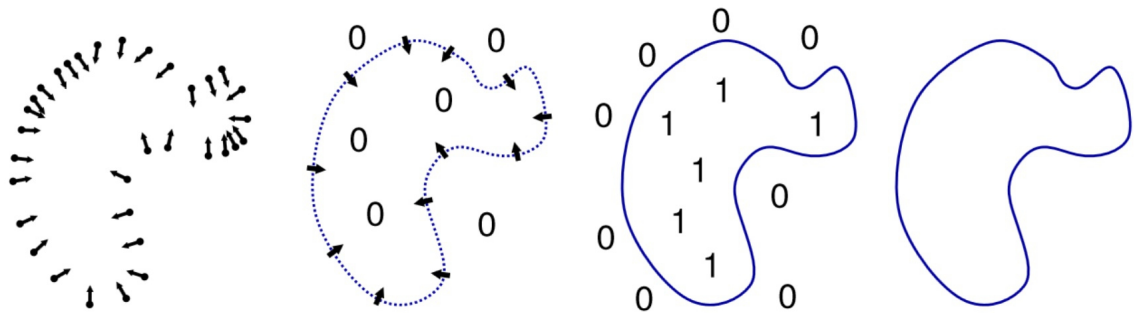


Figure 31: The poisson process of surface reconstruction in two dimensions. [48]

The method requires the points to have oriented normals that are used to approximate a vector field. The function  $f$  is then solved so that the the gradient of it is closest to the vector field. In Figure 31 we first have the points with oriented normals (thin arrows) and then the function (blue line) with gradients (thick arrows) analogous to the normals. This function is used to separate the point space to points within the surface and points outside of it.

## 3.4 Available Software

This section will cover the software libraries and drivers used to retrieve the RGB-D information from the hardware system. We will also cover the software and formats used to export the 3D model.

### 3.4.1 Point Cloud Library (PCL)

Point Cloud Library (PCL) is an open source library started in 2010 containing numerous tools for point cloud processing [49]. PCL offers perception algorithms, data visualization, filtering, and editing. It works on multiple platforms on desktop, mobile and web. The heavily modular small code libraries of PCL are linked together and can be compiled separately. PCL aims to be a stand-alone library for everything related to point cloud processing, which means that it also offers libraries such as OpenNI for acquiring the point clouds and extensive tools for exporting the processed point clouds to different formats. PCL is also the backbone for 3D processing in the Robot Operating System (ROS), which is a popular framework for writing robot software [50].

In PCL the point clouds are saved in a specifically developed Point Cloud Data (.pcd) format. The PCD offers more flexibility and speed than other suitable formats, especially when used with PCL. The .pcd format uses either standard ASCII text to store the data as plain text or a binary form where the data is a complete memory copy of the data array. The ASCII format is better if you wish to manipulate the data outside of PCL libraries or simply want to review it in plain text. Obviously the binary form has better performance.

The simplest data stored in the .pcd file are just the XYZ coordinates of the points in the point cloud. The XYZ coordinates can be complemented with different information, such as color, normals and curvatures. A normal is a vector which is perpendicular to the surface ( $\mathbf{n}$  in Figure 32) and curvature is the amount the surface deviates from being flat. The curvature of a point can be defined as the curvature of a circle touching the point from the positive or negative direction of the point normal. A circle with a smaller radius will bend more sharply and will have a higher curvature ( $\mathbf{k}$  in Figure 32).

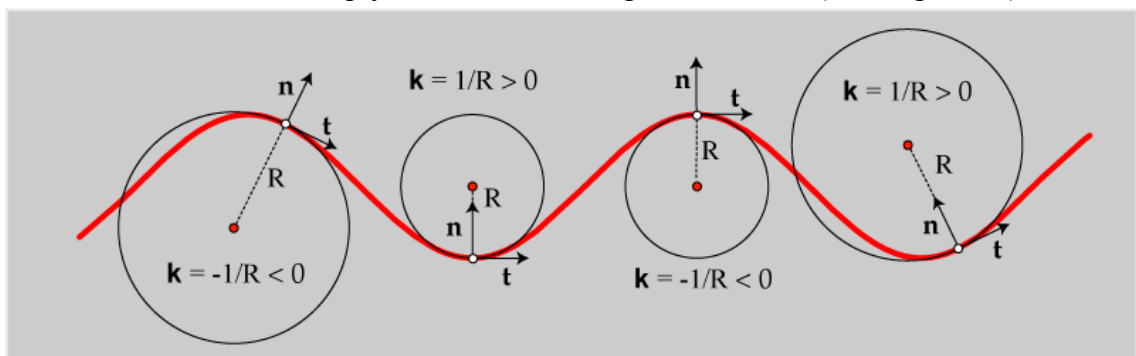


Figure 32: Normals and curvatures illustrated in 2D [51].

### 3.4.2 Open Natural Interaction (OpenNI)

OpenNI (Open Natural Interaction) is a framework that provides hardware drivers and several APIs to control a NUI device (Figure 33) with custom applications or middleware libraries. OpenNI was developed by the original Kinect depth sensor developer PrimeSense.

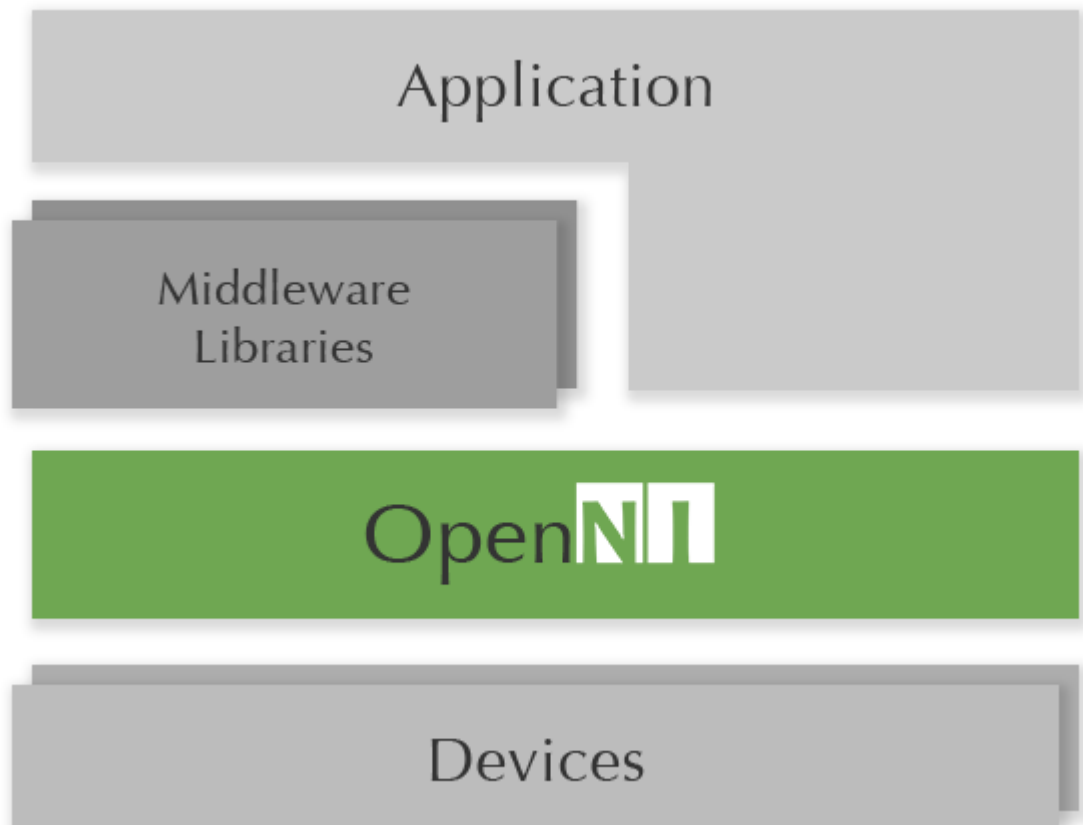


Figure 33: OpenNI architecture [52].

There has been some controversy surrounding the Kinect depth sensor developer PrimeSense. PrimeSense developed the first generation Kinect in cooperation with Microsoft and was a founding member in the OpenNI project. PrimeSense was purchased by Apple in 2013 which caused the end of the OpenNI project [53], and caused a disruption in the development. Other members of the project decided to keep developing OpenNI as a different version OpenNI 2 [52], but this event is still causing difficulties in using the OpenNI framework. OpenNI 2 is developed specifically for the Structure Sensor [53] and is not compatible with many of the other interfaces or frameworks that were used with the first version of OpenNI, including PCL. In PCL OpenNI is used to get point clouds from the device, Kinect for Xbox 360 in this case.

### 3.4.3 Other software

Many other libraries and software packages are used to support the operation of OpenNI and PCL.

**Boost** C++ libraries [54] contain over 80 different open-source and peer-reviewed libraries that extend the functionality of C++. Boost is considered a standard library for almost all C++ development and many of the libraries are considered to be included in the next C++ standard library. The libraries extend from algebra to testing and signal processing. In PCL, Boost is used mainly for shared pointers and thread operations.

**Eigen** [55] is a versatile math library that provides extensive matrix operations among other algebra. In PCL, Eigen provides matrix operations and optimized math calculations.

**Fast Library for Approximate Nearest Neighbors (FLANN)** provides fast approximate nearest neighbor searches in high dimensional spaces [56].

**Visualization Tool Kit (VTK)** is an open source software system with many different solutions in graphics, image processing and visualization [57]. PCL uses VTK for point cloud rendering and general visualization scenarios.

**Paraview** is a multi-platform data analysis and visualization application built on top of VTK. It is an easy tool for 3D model visualization and examination.

### 3.4.4 3D Model Export Formats

3D models are stored in many different formats. The best format depends on many factors such as the usage of the model, level of detail and the type of information it may contain.

The **Polygon File Format (PLY)** [58] was designed to store data from 3D scanners. It is not a general 3D description language, but a very specific way of describing a single object. It usually only contains the 3D-points for vertices and vertex indices for each face. Applications can expand the format with other information. The data can be stored either in an ASCII or a binary representation.

The **Visualization Tool Kit** also has its own file formats [59]. It offers two different type of formats: Simple serial formats similar to the PLY format and complex XML



based formats. The file formats are very extensive and offer many different options, but might not be supported by all visualization and 3D handling tools.

**Virtual Reality Modeling Language (VRML)** [60] is the ISO and industry standard for storing 3D data. It specifies the object vertices and edges with optional information such as textures, shininess, or transparency. VRML also supports adding program code to the files, so that you can have more interactive elements such as custom events after clicking a certain point of the model. The data is stored in plain text.

**Extensible 3D Graphics (X3D)** is the successor to VRML. It expands VRML by moving to the XML syntax and enhancing the APIs [61].

### 3.5 Summary

In this chapter we examined the process of transforming a point cloud to a better visual representation by improving the data with processing algorithms and reconstructing the surfaces with different methods. We also covered the software used to implement this process. In the implementation phase we use almost all of the software and methods examined in this chapter. Some surface reconstruction methods, e.g. Power Crust, were left out of the implementation due to the lack of proper open source libraries, and not all 3D model export formats were used.

## 4 IMPLEMENTATION

This chapter presents how the implementation part of the thesis was done. We describe the environment and hardware used to capture a scene, and then a walkthrough of the software implementation.

### 4.1 Environment

The first generation Kinect is used for capturing the input data in the implementation part of this thesis since it has good support for open source software, and is easily available. The surface reconstruction and export functions should work equally well with data captured with depth sensors in other current range imaging devices such as the Kinect for Xbox One or the Structure Sensor.

Microsoft Windows 7 was chosen as the platform after experiments with the Debian-based Linux operating system Ubuntu. Ubuntu would have had better support for some libraries and better supporting software for data visualization, but Windows was more familiar and already installed on the used desktop computer.

For software development we used the Microsoft Visual C++ 2010 express. As with Windows in general, the Visual Studio was not the most obvious option but was selected due to familiarity from working with it on a daily basis. Most of the libraries used have wrappers made for several programming language such as Python or C#, but best support is for C++, so this was chosen as the programming language. Hardware specifications of the computer used to run the software are

- Processor: Intel Core2 Duo CPU E6750 @ 2.66GHz
- Memory: 4.00 GB RAM DDR
- Hard Drive: Western Digital Blue WD5000AAKX, 7200 RPM 16MB Cache
- Graphics: NVIDIA GeForce GTX 550 Ti

## 4.2 Data capture setup

A setup (Figure 34) of simple and complex objects with adjustable lighting was created in order to capture the input data through the Kinect.



*Figure 34: Data capture setup with 3 objects.*

The objects (Figure 35) from the left are a blue Bluetooth speaker with intricate surface details, a black wooden sculpture with very complex shapes, and a white salt shaker with even plastic surfaces. These objects were chosen to test the Kinect depth sensing capabilities with different surfaces.

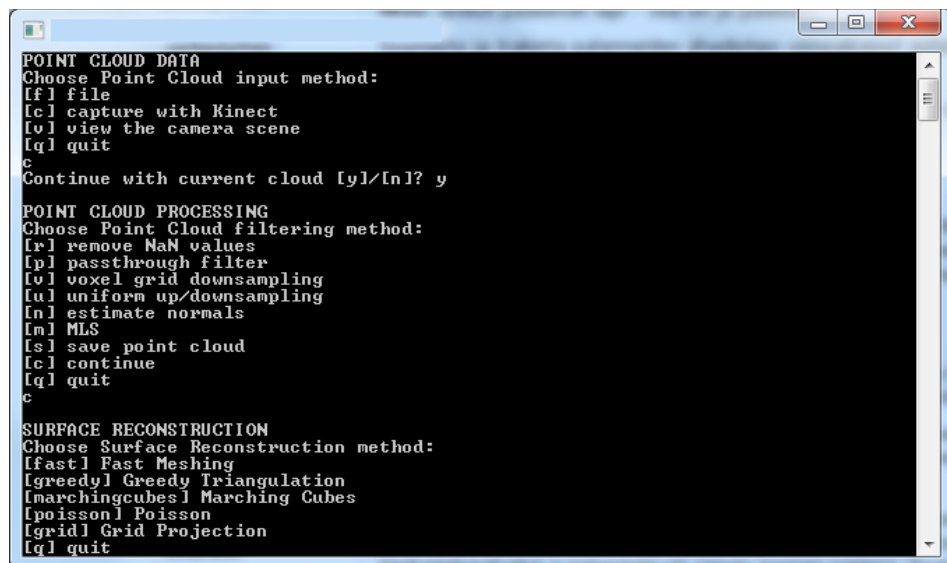


*Figure 35: Objects in the scene.*

The scope of the thesis was to capture the depth information from a single RGB-D image, so occlusion was known to become an issue. Realistically, distinguishing the entire object from depth information from only one angle is impossible, since we have no knowledge of the shape of the object on the opposite side. Any functions and methods done for depth data from a single point-of-view should be applicable for complete depth data as well. Choosing objects with clear edges and negligible concave features was necessary for good performance, since these kind of details require multiple input angles to be accurately recreated. Another goal in choosing an interesting capture setups was selecting objects with surfaces that are known to be difficult for machine vision: reflective and transparent surfaces.

### 4.3 Software implementation

The user interface (UI) of the software is a simple command line interface (CLI) (Figure 36). The actions are selected from a list of options for each phase of the implementation. The UI functions will be covered in more detail next as we walk through the implementation that follows the process described in Figure 37.



```
POINT CLOUD DATA
Choose Point Cloud input method:
[f] file
[c] capture with Kinect
[v] view the camera scene
[q] quit
c
Continue with current cloud [y]/[n]? y

POINT CLOUD PROCESSING
Choose Point Cloud filtering method:
[r] remove NaN values
[p] passthrough filter
[v] voxel grid downsampling
[u] uniform up/downsampling
[n] estimate normals
[m] MLS
[s] save point cloud
[c] continue
[q] quit
c

SURFACE RECONSTRUCTION
Choose Surface Reconstruction method:
[f] Fast Meshing
[g] Greedy Triangulation
[m] Marching Cubes
[p] Poisson
[g] Grid Projection
[q] quit
```

Figure 36: The CLI used.

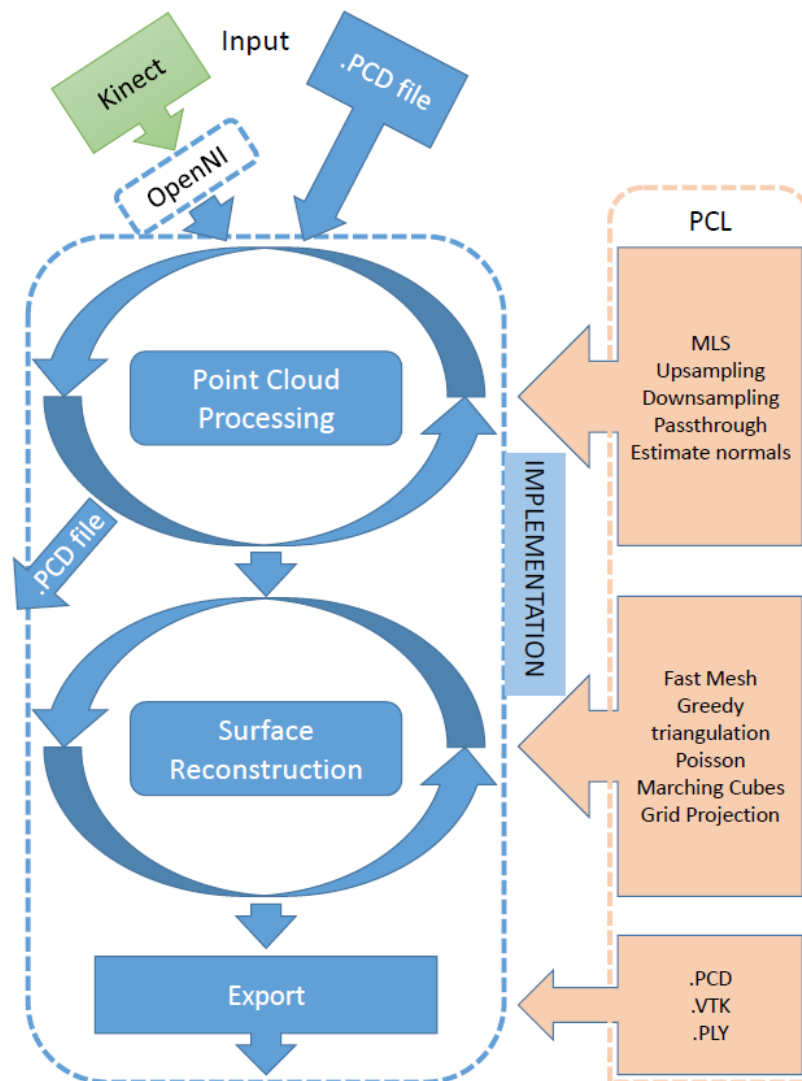


Figure 37: Process flow of the software implementation.

The first step in the implementation is the input of the depth data. The two options are to either capture it with the Kinect, or to import a PCD file. You can also simply view the scene through the Kinect camera if you wish to position the objects or otherwise modify the scene. The options are selected from the following view:

```

Choose Point Cloud input method:
[f] file
[c] capture with Kinect
[v] view the camera scene
[q] quit

```

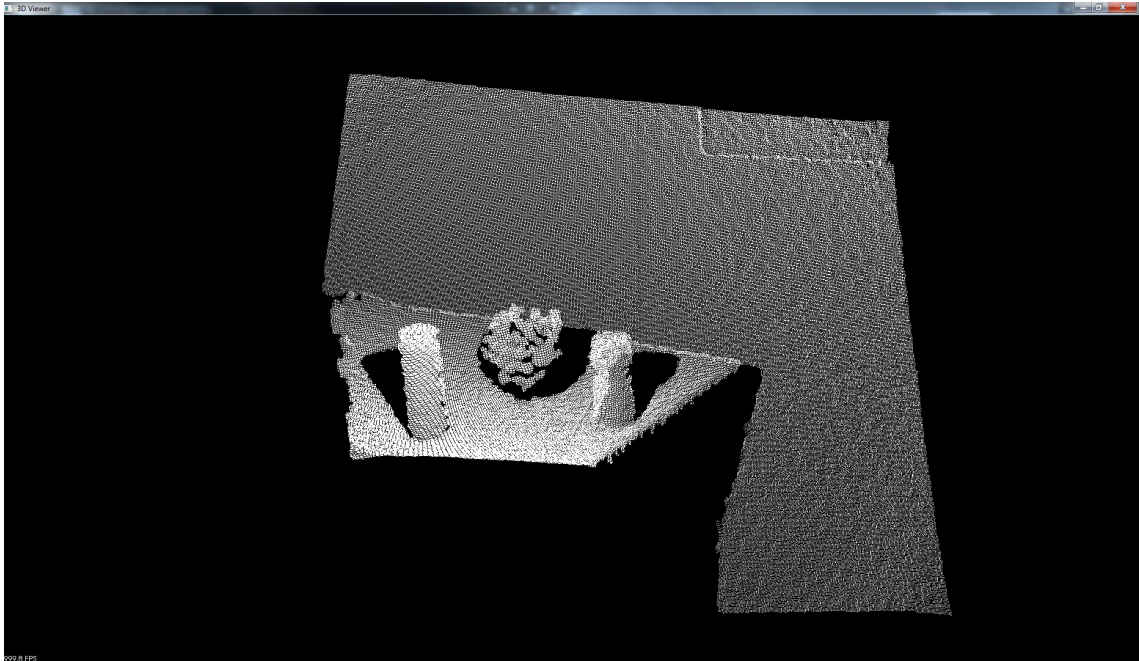
With the Kinect the image is retrieved through OpenNI drivers and methods that have been wrapped inside a PCL class. Even though the data goes through many functions and interfaces the actual code required is minimal. You only need to define the im-

age modes for both the RGB camera and the depth sensor (or leave them as default), and give the device id number if you have more than one devices in the setup:

```
pcl::OpenNIGrabber::Mode image_mode =  
    pcl::OpenNIGrabber::OpenNI_Default_Mode;  
  
pcl::OpenNIGrabber grabber ("", image_mode, image_mode);
```

Now the `grabber` variable has the interface to the Kinect. With a simple `start()` function call the device starts up and begins feeding data through the OpenNI interface. Grabbing the data is done through a callback function.

The depth data grabbed through the interface is in `pcl::PointXYZ` (Figure 38) or `pcl::PointXYZRGB` format depending on if you want to work with the color information or not. In this thesis we will not use the color information.



*Figure 38: Depth data retrieved from the Kinect in point cloud format.*

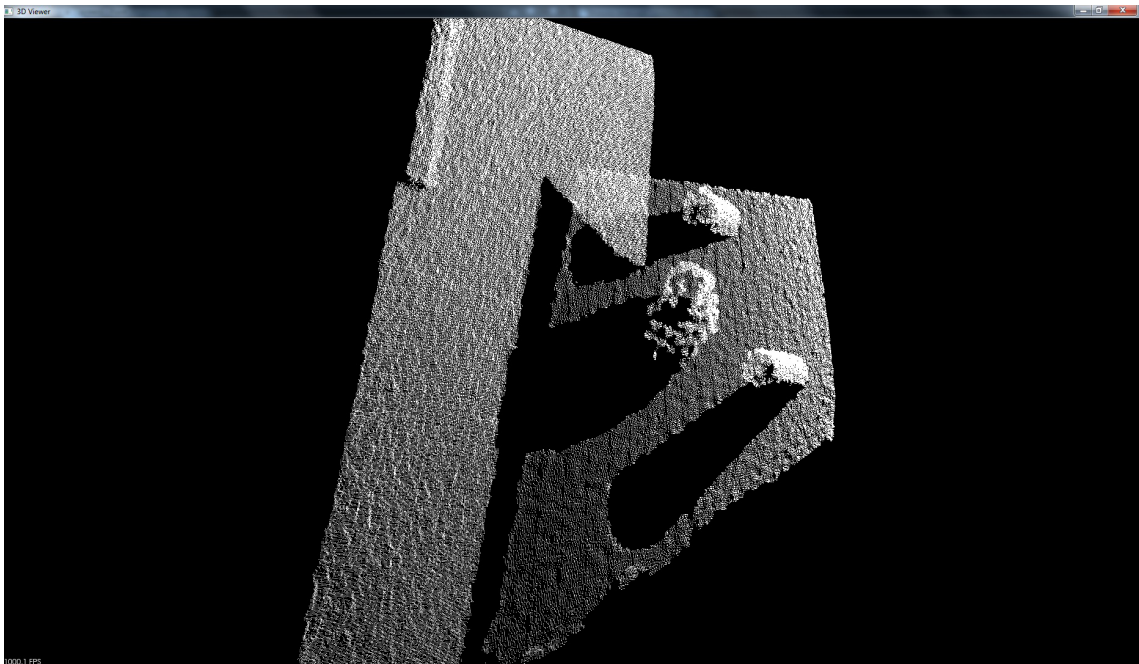
If you already have a PCD file ready you can choose to import it and skip capturing it through the Kinect. Once you have either imported the data or captured it the next phase is Point Cloud Processing. This phase offers a multitude of options that you can try, view the results and either decide to keep the changes or revert them:

```

POINT CLOUD PROCESSING
Choose Point Cloud filtering method:
[r] remove NaN values
[p] passthrough filter
[v] voxel grid downsampling
[u] uniform up/downsampling
[n] estimate normals
[m] MLS
[s] save point cloud
[c] continue
[q] quit
r
Keep changes [y]/[n]? y

```

The result of the processing is always shown in a separate window once it is done. As is clearly visible from the images (Figure 39), if we only want to model the area near the objects there is a lot of extra points on the wall and the table.



*Figure 39: Depth data from the Kinect from above.*

The first step of the processing is removing these unnecessary points with a pass-through filter. We filter the data twice: first we remove the points far behind the objects, and then we trim the data on the right side of the objects. The filtering is done with the PassThrough class in PCL. The filter is given the point cloud and some basic parameters:

```

Choose parameters:
Axis [x]/[y]/[z] (x) : z
filter range start (0.0) : 0
filter range stop (1.0) : 0.95
filter limits negative (false) : false

```

```

Choose parameters:
Axis [x]/[y]/[z] (x) : y
filter range start (0.0) : 0
filter range stop (1.0) : 0.45
filter limits negative (false) : false

```

The range is given in axis (X/Y/Z) with a default axis X and then the range in meters. You can also select if you want to keep the points within the given range, or points outside of the given range. The point cloud is filtered in both X and Y axis with best values found by trial and error. The chosen parameters are handled in code in the following way:

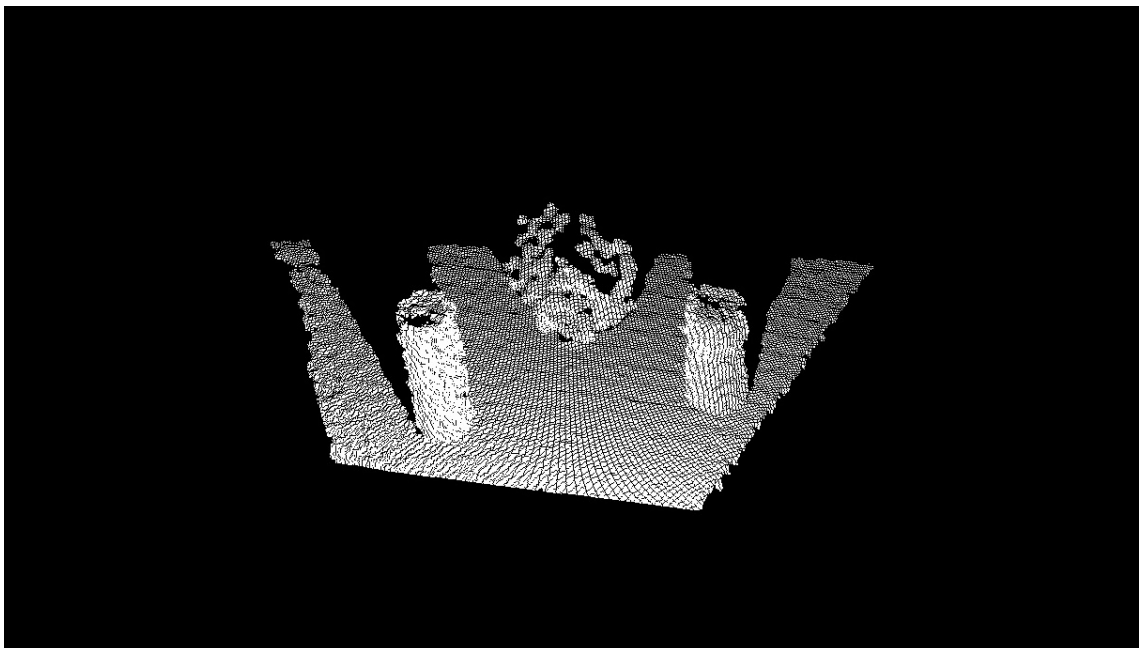
```

pcl::PassThrough<pcl::PointXYZ> passF;
passF.setInputCloud (currentCloud);
passF.setKeepOrganized(keepOrganized);
passF.setFilterFieldName (axis);
passF.setFilterLimits (filterstart, filterstop);
passF.setFilterLimitsNegative (filterLimitsNegative);
passF.filter (*currentCloud);

```

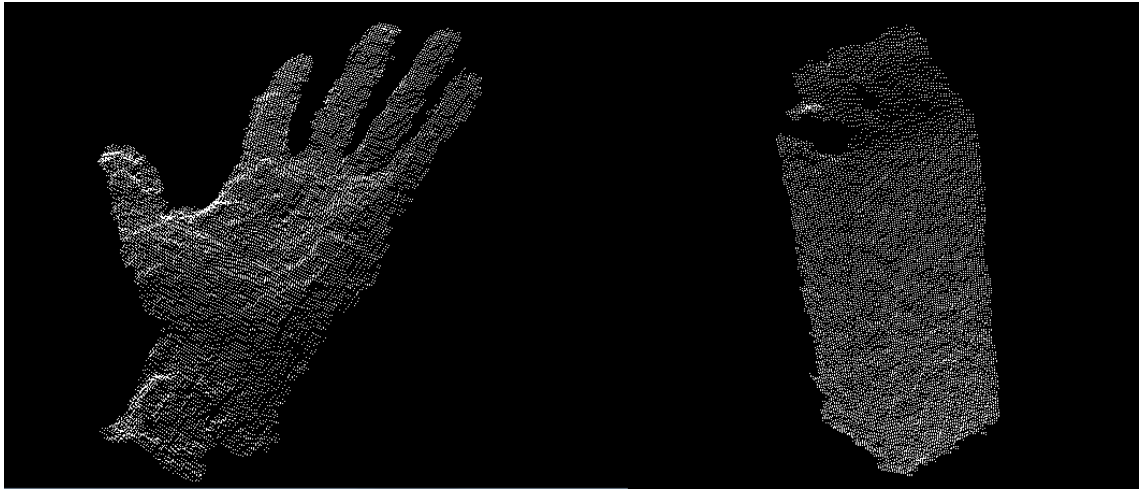
The `currentCloud` parameter contains a pointer to the point cloud being processed and the other parameters are the ones chosen through the UI.

Now we have a good and compact point cloud data set (Figure 40) limited to the objects we want to model. In Figure 41 we have two more examples, a hand and a milk carton, of point clouds captured through the Kinect and filtered with the passthrough filter.



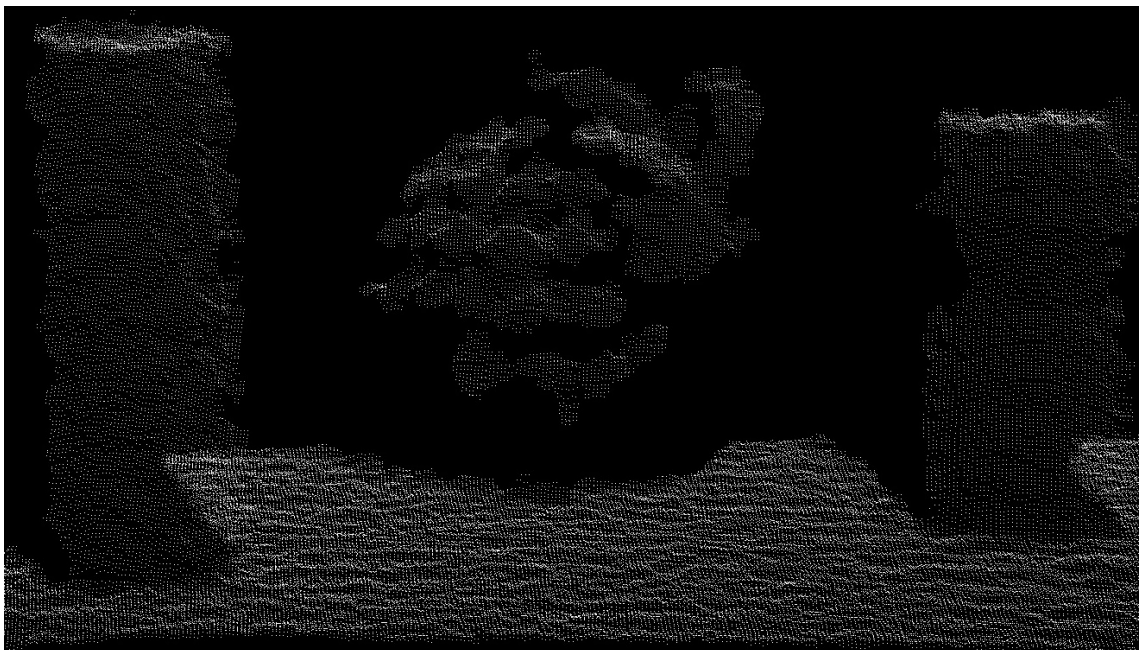
*Figure 40: Unnecessary points removed from the data.*





*Figure 41: Captured and filtered point clouds of a hand and a milk carton.*

We will continue walking through the process with the larger point cloud of multiple objects. The point cloud has a lot of noise and other anomalies (Figure 42) due to the limitations of Kinect (Figure 19).



*Figure 42: Close-up of the depth data. Visible noise on the even surfaces and distortions around the edges.*

The next step is trying to smooth out these anomalies and retrieve the normals of the data points that are needed for further processing. The smoothing and normal construction is done with the MLS-algorithm provided by PCL. The implementation of running

the point cloud through the processing method is similar with all the methods we will use: The processing function is called with the original point cloud as a parameter and it returns the outcome of the processing as the return value. The processing method, `pcl::MovingLeastSquares` in this case, is first initialized with the desired parameters and then computed. The parameters given to the MLS object determine that we wish to calculate and add the normals of the point cloud and use a polynomial fit with a KdTree type of a search method with the default radius of 3 cm:

```

pcl::PointCloud<pcl::PointNormal>::ConstPtr
run (pcl::PointCloud<pcl::PointXYZ>::ConstPtr inputCloud)
{
    // Initialize Parameters
    std::string searchRadiusStr = "";
    float searchRadius = 0.03f;
    cout << "Search Radius meters (0.03) : ";
    cin >> searchRadiusStr;
    searchRadius = std::stof( searchRadiusStr );

    std::string computeNormalsStr = "";
    bool computeNormals = false;
    cout << "filter limits negative (false) : ";
    cin >> computeNormalsStr;
    if(!computeNormalsStr.empty() && computeNormalsStr == "true")
        computeNormals = true;

    // Create the search method: KD-Tree.
    pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new
        pcl::search::KdTree<pcl::PointXYZ>);

    // Declare the output cloud with a point cloud type that includes
    // the normals.
    pcl::PointCloud<pcl::PointNormal>::Ptr outputCloudWithNormals (new
        pcl::PointCloud<pcl::PointNormal>);

    // Initialize the mls object. Set input and output cloud types.
    pcl::MovingLeastSquares<pcl::PointXYZ, pcl::PointNormal> mls;

    // Setup the parameters.
    mls.setInputCloud (inputCloud);
    mls.setComputeNormals (computeNormals);
    mls.setPolynomialFit (true);
    mls.setSearchMethod (tree);
    mls.setSearchRadius (searchRadius);

    // Process the data.
    mls.process (*outputCloudWithNormals);

    return outputCloudWithNormals;
}

```

The code follows the style used in the PCL tutorials [62]. Using constant pointers for parameters instead of direct objects follows the best practices in coding C++ by not reserving memory pointlessly.

The resulting point cloud with normals is visualized, like all other data in this thesis, with a custom visualizer. The visualization class made has a `run()`-function with overrides for each different type of point cloud. The `PCLVisualizer` class is initialized according to the given cloud type, in the case of a point cloud with normals initialization is done as follows:

```
boost::shared_ptr<PCLVisualizer> getVisualizationWithNormals
    (PointCloud<PointNormal>::ConstPtr cloudWithNormals)
{
    // Separate the PointXYZ pointcloud from the cloud with normals.
    PointCloud<pcl::PointXYZ>::Ptr regularCloud
        (new PointCloud<PointXYZ>);
    copyPointCloud(*cloudWithNormals, *regularCloud);

    // Initialize the PCLVisualizer
    boost::shared_ptr<PCLVisualizer> visualizer
        (new PCLVisualizer ("Point Cloud Viewer with Normals"));
    visualizer->setBackgroundColor (0, 0, 0);

    // Add the point cloud
    visualizer->addPointCloud<pcl::PointXYZ>
        (regularCloud, "point cloud");
    visualizer->setPointCloudRenderingProperties
        (PCL_VISUALIZER_POINT_SIZE, 3, "point cloud");

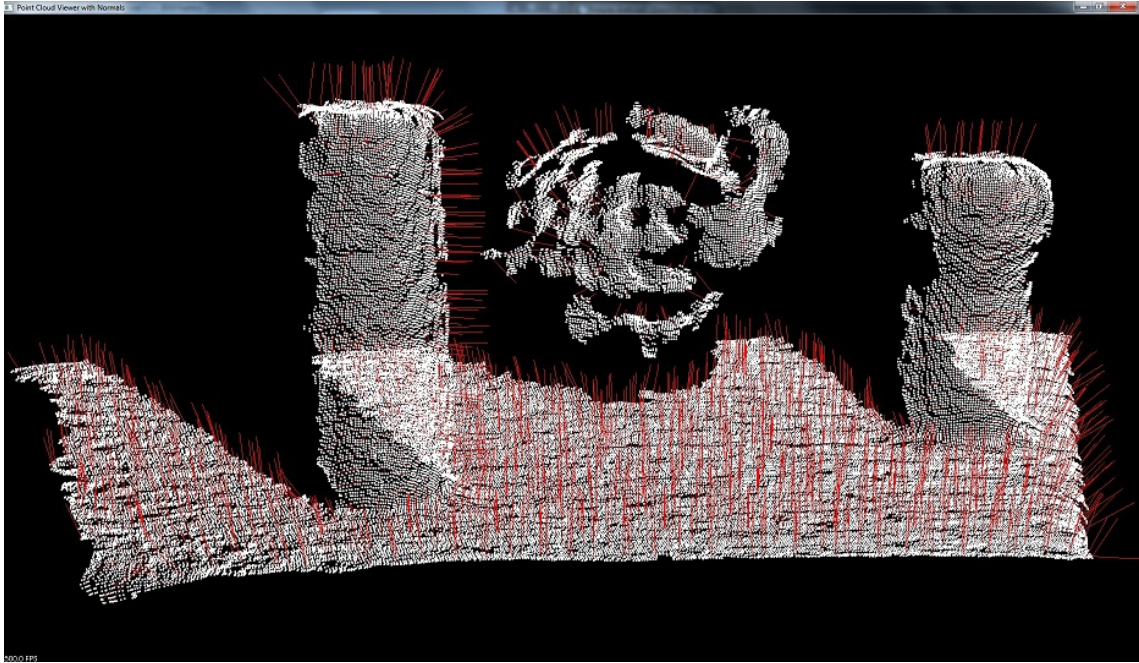
    // Add the normals
    visualizer->addPointCloudNormals<PointXYZ, PointNormal>
        (regularCloud, cloudWithNormals, 30, 0.02, "normals");
    visualizer->setPointCloudRenderingProperties
        (PCL_VISUALIZER_COLOR, 1.0, 0.0, 0.0, "normals");

    // Setup the camera
    visualizer->initCameraParameters ();
    visualizer->resetCameraViewpoint ("normals");

    return (visualizer);
}
```

The result is a visualization (Figure 43) of the point cloud with red normals that are 2cm long. Only every 30<sup>th</sup> normal is visualized since the cloud would otherwise be too crowded. The smoothed point cloud with added normals is the starting point for many of the surface construction methods. One issue with the MLS-algorithm is that it breaks the structure of the point cloud given by Kinect. The Kinect returns an organized point cloud with a height of 480 points and width of 640 points. The Kinect cannot capture all the points properly due to technical limitations or the point being out of range, but all points in the 480x640 grid have to have a value to keep the structure organized. To achieve this some of the points have undefined “NaN” values, which means that they have no real numerical value. The MLS-algorithm, among other algorithms, cannot handle these NaN points so they have to be removed. There are no clear values to replace the NaN values with, so after removing the NaN values the point cloud becomes unor-

ganized with a height of 1. This is the normal state of a point cloud for most surface construction algorithms.



*Figure 43: Custom visualization with normals.*

Using an organized cloud has the biggest advantages in speed, since finding the nearest neighbors is much faster. Some surface construction algorithms such as the Organized Fast Mesh require that the point cloud is organized.

Now that we have a processed point cloud we move to the Surface Reconstruction phase:

```
SURFACE RECONSTRUCTION
Choose Surface Reconstruction method:
[fast] Fast Meshing
[greedy] Greedy Triangulation
[marchingcubes] Marching Cubes
[poisson] Poisson
[grid] Grid Projection
[q] quit
```

The Organized Fast Mesh method is chosen as the first way to reconstruct the surface and initialized with some critical parameters:

```
Fast Mesh Triangulation
Triangle Size : 3
Max Edge Length : 0.025
```

We choose a triangle size of 3 pixels and a maximum edge length of 2.5 cm for the constructed meshes.

The point cloud used in building the resulting surface construction (Figure 44) of the Organized Fast Mesh had not been smoothed with the MLS-algorithm, and it shows in the outcome. The mesh is not smooth or spotless since all the NaN values left in the point cloud cause holes, but the method builds the mesh in just a few seconds.

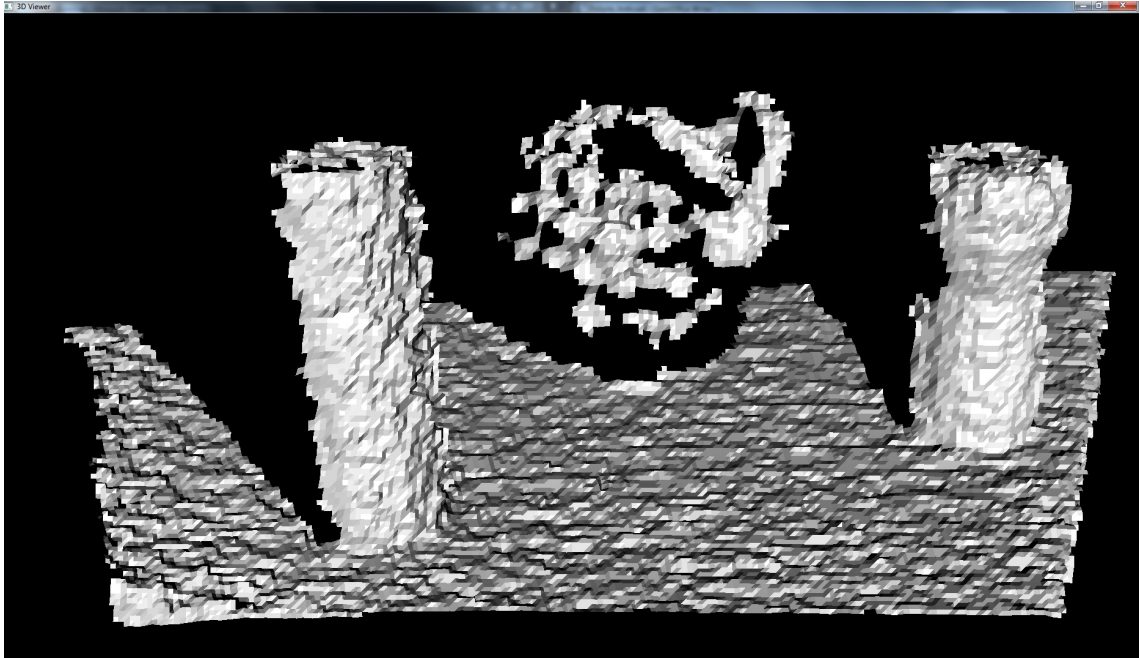


Figure 44: Organized Fast Mesh surface reconstruction.

The rest of the methods use the smoothed and normalized cloud. The next method we will demonstrate is the Greedy Projection Triangulation. The code of the function looks very similar to the MLS class. First we initialize the needed parameters we want to modify:

```
Greedy Projection Triangulation
Search Radius meters (0.05) : 0.55
NN Distance Multiplier (2.5) : 5.5
Max NN (250) : 2500
```

We select a search radius and nearest neighbor parameters higher than the default parameters, since the point cloud being processed has not been downsampled and contains a lot of points. The selected parameters are then used in the function:

```
pcl::PolygonMesh::ConstPtr
run (pcl::PointCloud<pcl::PointNormal>::ConstPtr cloud_with_normals)
{
    // ....
    // .... parameter initialization with the given parameters etc...
    // ....

    pcl::GreedyProjectionTriangulation<pcl::PointNormal> greedy;
    boost::shared_ptr<pcl::PolygonMesh> outputMesh
        (new pcl::PolygonMesh);
```

```

// Setup the parameters.
greedy.setInputCloud (inputCloudWithNormals);
greedy.setSearchMethod (kdtree);
greedy.setSearchRadius (searchRadius);
greedy.setMu (mu);
greedy.setMaximumNearestNeighbors (maxNN);
greedy.setMaximumSurfaceAngle(M_PI/4);
greedy.setMinimumAngle(M_PI/18);
greedy.setMaximumAngle(2*M_PI/3);
greedy.setNormalConsistency(false);
greedy.reconstruct (*outputMesh);

return outputMesh;
}

```

Getting the best outcome is heavily reliant on choosing the correct parameters. The parameters for this function include the definition of the search tree parameters such as search radius (in cm) and nearest neighbor details. The minimum/maximum angles for the produced triangles and the maximum angle of a surface normal were left as their default values. You can also define whether the normals are consistent or not, but with data collected through the Kinect they never are. Consistent normals can only be achieved with data that has no anomalies and completely smooth surfaces. The used parameters result from multiple attempts.

The resulting model (Figure 45) is very smooth, but because of the greediness of the search function used by the method there are still small holes in the model. The processing of the model is also much slower, in the range of minutes to almost an hour depending on the parameters used.

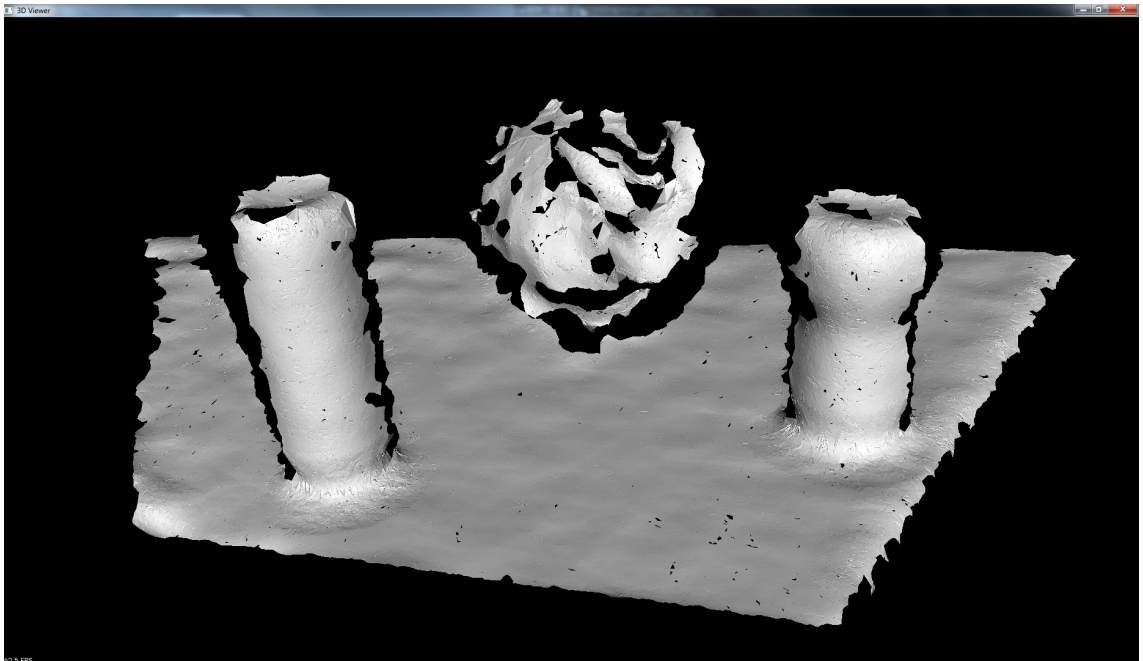


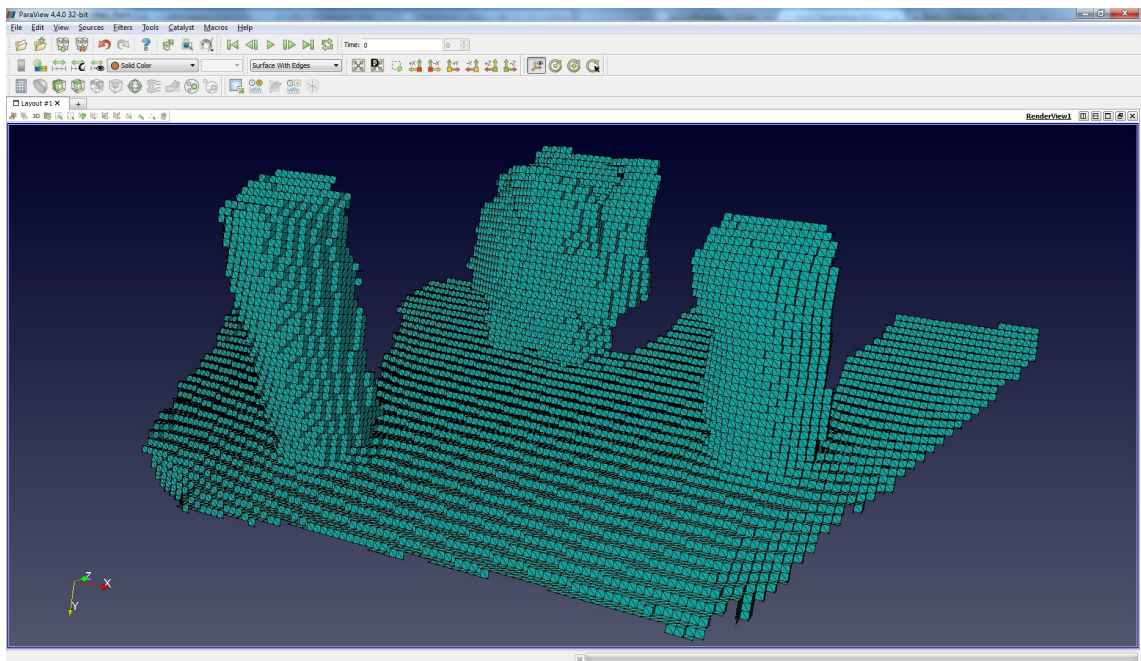
Figure 45: Greedy Projection Triangulation mesh.

Getting a “watertight” model with no holes with the given point cloud data requires more complex processing methods or the usage of parameters that require more processing time. The Grid Projection method can pad the holes in the data if a proper value for the padding size parameter is chosen:

```
Grid Projection
Padding Size (3) : 3
Grid Resolution meters (0.005) : 0.005
```

The resolution parameters tells the size of cubes used for the grid projection (5 mm). With a padding size of 3 points we get rid of all the small holes in the mesh.

The outcome (Figure 46) is a solid representation of the scene with flat surfaces being flat and shapes of the objects very visible. Choosing a grid resolution smaller than 5 mm would result in an even more accurate reconstruction, but the process of reconstructing the big surface with these parameters already took several hours.



*Figure 46: The mesh created with the grid projection method visualized in the ParaView software.*

With smaller point clouds a grid resolution of 1 mm produced very accurate results (Figure 47) with the hand point cloud, and good results with the milk carton point cloud. There were some bigger holes left in the milk carton model due to that side of the object being slightly occluded from the Kinect's point of view and having a slightly reflective surface.

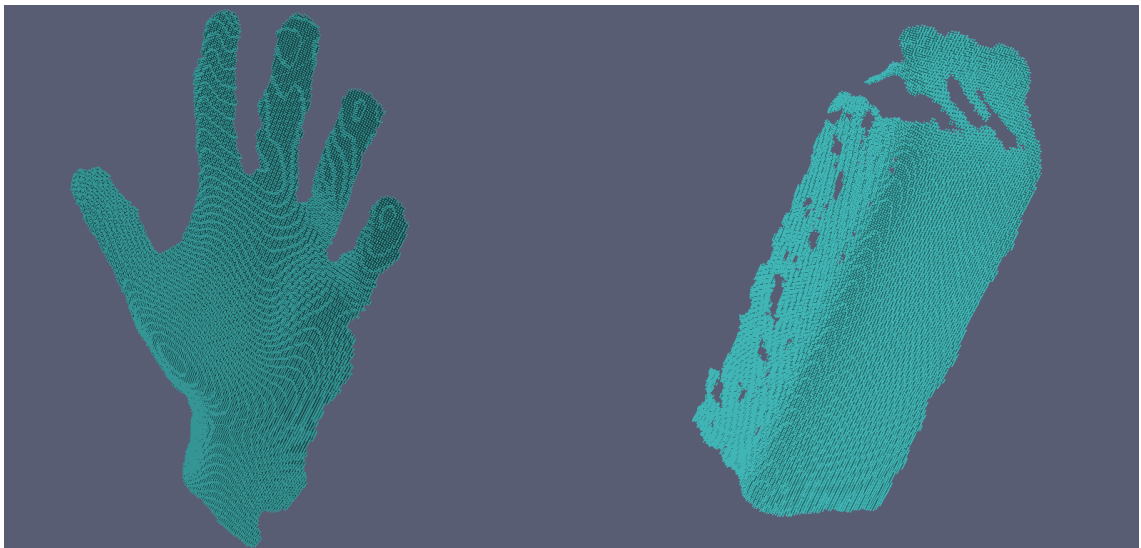


Figure 47: The hand and milk carton point clouds with surfaces reconstructed with the Grid Projection method visualized in ParaView [63].

Exporting and saving the point clouds or meshes is done with the automated functions found in the `pcl::io`-framework:

```
// Saving a point cloud in the .pcd format.
pcl::io::savePCDFile ("MLSsmoothedCloud.pcd", *pointsWithNormals);
// Saving a PolygonMesh in the .vtk format.
pcl::io::saveVTKFile ("gridProjectionResultMesh.vtk", *outputMesh);
```

The exported files also have similar import functions, so the processing can easily be continued with other suitable methods. The files can then be used in other software, for example, the Unity3D [64] game development environment (Figure 48). In Unity3D the 3D model of a hand could be used, for example, as a part of a bigger human model, as the surface of a game, or as an individual game object.

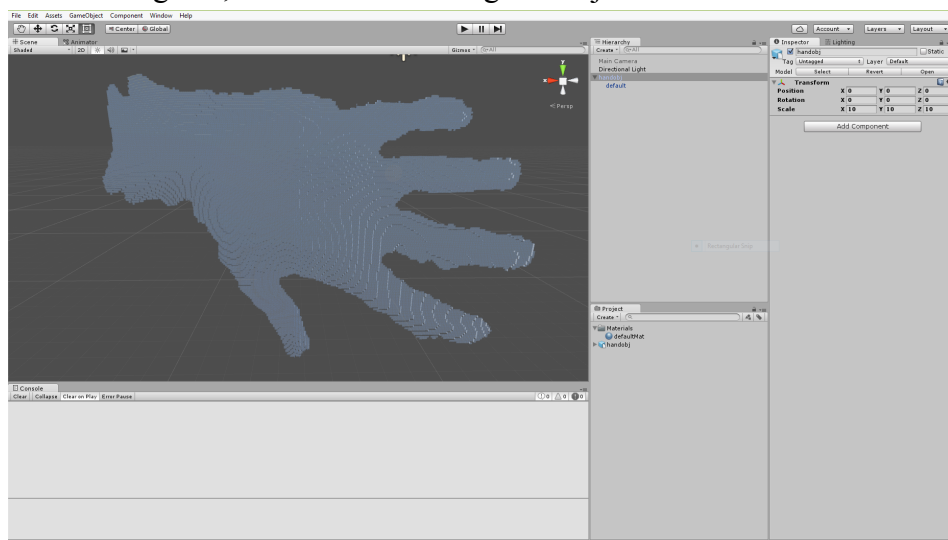


Figure 48: The generated hand model imported to the Unity3D [64] game development environment.



## 5 EVALUATION

This thesis covers the hardware, software, methods, and algorithms used in the implementation. The focus of the hardware research is on the Microsoft Kinect since it was selected to be used in the implementation part of this thesis. Performance comparison with the other devices would be interesting, but not reasonable due to the cost of these devices. The software section centers almost completely on the PCL framework since it is the largest and most mature framework and contains all the needed tools. There are however several smaller alternatives that could be better for some use-cases, and could be included in the research.

The point cloud processing and surface construction methods focus on the ones implemented in PCL. Some common methods had to be left out since the number of methods is simply too big. The decision was made not to go too deep to the mathematics of each process, since each sub-chapter could easily expand to the size of an entire thesis itself.

### 5.1 Implementation

The decision to use Windows and Visual Studio was questionable. The benefits of using a Linux based environment are obvious: almost all research and development in the field is done on Linux. After a long struggle with hardware and software compatibility issues related to Linux, the decision was made to go back to familiar tools in Windows and Visual Studio. The familiarity of Visual Studio comes from working with it on a daily basis for many years, and was the correct decision in this case. It was also interesting to see how well the libraries worked on Windows. Linux is still the recommended environment for all things related to surface construction and signal processing in general.

The software produced in this thesis provides a CLI interface to some of the point cloud data processing and surface reconstruction libraries found in PCL. It also covers the import and export of the used file formats, and gives an access to the Kinect. The usability of the program is poor, and should be expanded with a visual UI and better error management. The user of the tool has to be well informed on how the methods work to find the best parameters, and there is little room for error. The software is sufficient to produce the results needed for this thesis, and the goal of producing a proof-of-concept tool is achieved.

The resulting 3D models show the different levels of accuracy you can achieve with different amounts of processing. We show that you can build a noisy 3D mesh in almost real-time with the organized fast mesh approach. With either more powerful hardware or more processing time we get better results with greedy projection triangulation. Achieving a dense 3D model as the result requires a lot of processing, and we show it with the grid projection method. We learn that achieving the optimal result for any given scenario depends on the requirements for the 3D model and the processing power, or time, available. The results could be expanded with more methods and automatic parameter variation.

The results do show that it is very possible to create a setup for point cloud capturing and 3D model reconstruction with reasonable resources. It is shown that you can get decent results without any expensive hardware or software. All software used was free for personal use, all PC hardware was priced well under the average desktop computer standards, and the first generation Kinect is very affordable.

## 5.2 Performance

Performance of the Surface Reconstruction methods in Table 1 is critical on determining what the tool can be used for.

*Table 1: Surface Reconstruction Method performance*

	Multiple Objects	Hand	Milk Carton
Fast Mesh	40 seconds	42 seconds	40 seconds
Greedy Projection	80 minutes	16 minutes	14 minutes
Grid Projection	4 hours 35 minutes	43 minutes	98 minutes

The performance of the Surface Reconstruction depends heavily on the selected method and the amount of points in the cloud. The Organized Fast Mesh method is fast enough that with good hardware it could be used for real-time applications even for large scenes like the multiple objects scene. The reason for this is that it only accepts organized point clouds that are faster to process, and that it uses a simple algorithm. The speed shows as a low quality outcome, but the possibility of real-time processing is a big advantage. The Greedy Projection Triangulation and especially the Grid Projection methods use complicated algorithms that take a while to process large point clouds resulting in high processing times. These cannot be used for real-time applications, but produce higher quality 3D models.

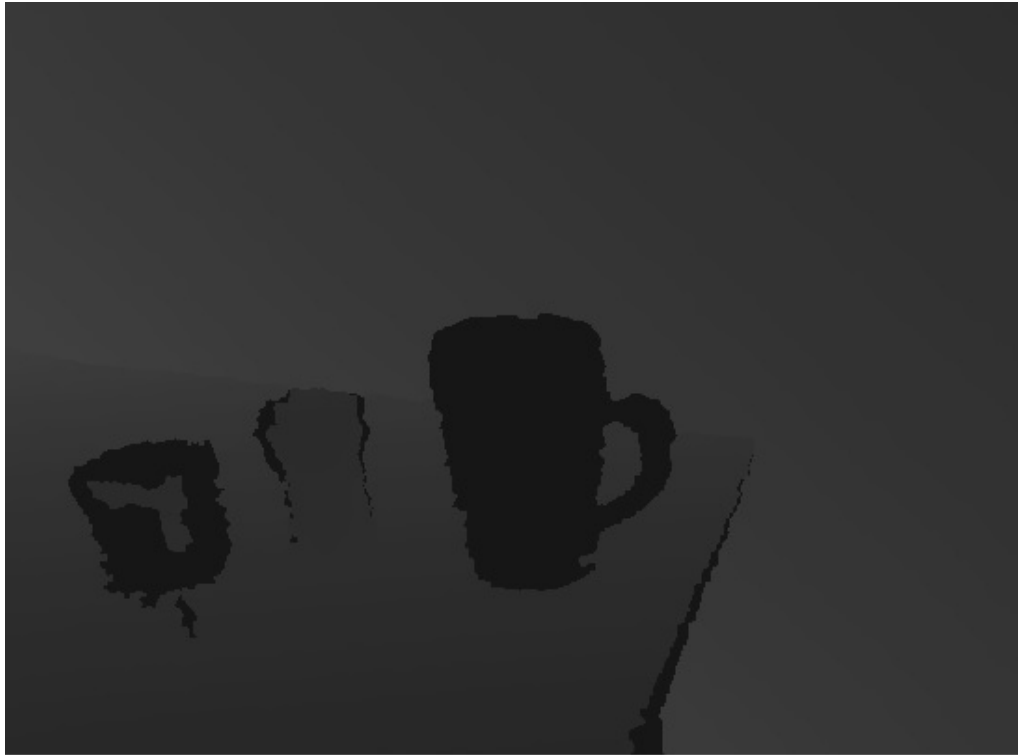
### 5.3 Technical limitations

In the implementation phase several problems occurred due to technical limitations in the software and hardware.

**Difficult Surfaces.** The properties of using infrared light as the medium to get the depth data is that it requires a good and direct reflection from the surface it hits. When the objects in the scene are made of substances such as glass or reflective metals (Figure 49) the depth data can be impossible to acquire since the light is reflected differently. The depth data retrieved will either be distorted and erroneous, or completely missing (Figure 50).



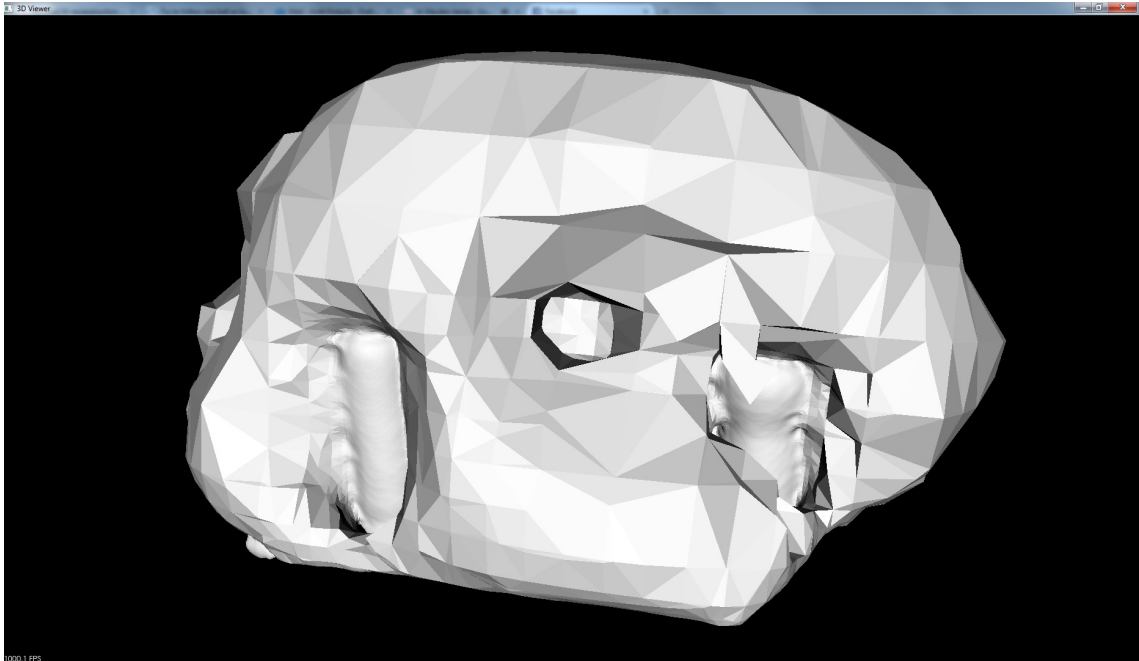
*Figure 49: RGB image from the Kinect with objects with a metal container and a beer mug. An object with a normal surface (the salt shaker) as reference.*



*Figure 50: Depth data from the Kinect with the same scene as in Figure 49. Depth visualized as different shades of grey.*

There are no good methods to recover data in situations like the ones pictured above. This has to be simply treated as a limitation to the operation of depth sensors based on light projection.

**Unsuitable methods.** Some surface construction methods like the Poisson, Concave Hull, and Power Crust always try to create a watertight mesh of the given scene. This is, of course, appropriate if the point cloud given as input is actually of a single object. But if the point cloud describes something completely different, say, a table with multiple objects, the results can be unsatisfactory (Figure 51). The results could be somewhat improved with better point cloud processing, like making sure that all the normals point to the correct directions, but, in the end, these construction methods are most likely not applicable for this selected scene.



*Figure 51: Blob created by the Poisson surface reconstruction method.*

**Software issues.** Since the used libraries and interfaces are in constant development there will be compatibility issues with the environment and hardware used. The down-sampling libraries provided by PCL use the `Eigen::std::vector` class which had compatibility problems with the Visual Studio 10 compiler. The point clouds used for surface construction were unnecessarily large and noisy because of this. The problem could be fixed by finding fixed or optimized versions for this specific environment, or by altering the code to use other libraries for data handling. The Marching Cubes implementation in the PCL framework did not work at all in this environment. The problem has been fixed in the newer version.

**Limited hardware.** The hardware of the desktop computer was out-dated. The surface construction process demands a lot of processing power and memory. Due to poor hardware the processing took a lot of time and limited the amount of parameter variation that could be tried out.

## 5.4 Further development

The software implementation part of the thesis leaves plenty of room for further development. The current CLI is slow and unintuitive to use, and the first step would be to improve the user interface by making it visual and better at handling input errors. The UI should also provide the user with information and help on how to use the algorithms and methods. Implementing a function that estimates the optimal parameters for each

method for the given point cloud would also be very useful. The methods implemented are a fraction of the methods available. The software could be expanded so that one can introduce new methods and tools with little effort. Connecting the implementation directly to the PCL github and using classes from there would also bring more maintainability and faster updates. The usage of RGB data was also very limited, and adding it to all the steps of the process should be a priority in further development. Color provides a more visually pleasing results and is needed in certain use cases of the produced 3D models. After the basic upgrades to the software the next step would be introducing the production of 3D models from multiple images. A device setup where you could capture all sides of an object and produce a complete 3D model would create more potential ways to utilize the results. Adding real-time processing would also be very interesting, especially if it would be complemented with pattern recognition algorithms that could capture and isolate objects in real-time.

## 6 CONCLUSIONS

Constructing a dense 3D model from a single RGB-D image is a process containing many steps. In this thesis, we walk through each of these steps and provide brief introduction to the available methods and tools used. We also show how to actually implement these steps by producing a tool capable of performing all the steps needed.

We present different methods and hardware used to capture a depth image. The methods of structured light and time-of-flight are highlighted as the most prominent and best supported by the common Microsoft Kinect product family. These complex processes are supported by presenting more general methods, utilizing functions also found in the human eye, with depth from stereo view and depth from light's interaction with a lens.

There are many alternatives for the software interfaces needed to retrieve the data from the depth sensor and to store it in a format that can be processed further. In this thesis we focus on the Point Cloud Library (PCL) framework that provides all the tools in a single package. The OpenNI drivers and libraries included in PCL are used to control the device and to deliver the captured data for processing. We also cover other software needed and the formats that can be used to store the produced 3D models.

The process of transforming the acquired data from a point cloud format to a 3D model is studied and described. We discuss the most common algorithms and explain the operations in the surface construction and point cloud processing methods. In the implementation part of the thesis, the theories and tools presented in the previous chapters are used. The software is developed with the Visual Studio developing environment with the C++ language. The software controls the Kinect to capture the depth and RGB-D data from the scene. We then process this data by eliminating unnecessary points, estimating the normals, and smoothing out the surfaces. The processed point cloud is then reconstructed as a 3D model with different surface construction algorithms. The algorithms vary in complexity, detail, and required processing power. We cover reconstruction methods from fast that produce rough results in almost real-time to methods that take hours to compute, but produce a high quality dense 3D representation. We also cover the issues faced with the software implementation.

The results show that generating a dense 3D model from a single RGB-D image is a complex process that can be made simpler by selecting the appropriate software and hardware, and by having knowledge of the necessary theory behind the process.

## REFERENCES

- [1] Horvath. *Mastering 3D Printing*. Apress. 3-10, 2014.
- [2] Lipson, Hod, Kurman. *Fabricated: The new world of 3D printing*. John Wiley & Sons, 2013.
- [3] Oculus Rift, [WWW], [Referenced 20.12.2015].  
Available: <https://www.oculus.com/en-us/rift/>
- [4] Google Glass, [WWW], [Referenced 20.12.2015].  
Available: <https://www.google.com/glass/start/>
- [5] Samsung Gear VR, [WWW], [Referenced 20.12.2015].  
Available: <http://www.samsung.com/global/galaxy/wearables/gear-vr/>
- [6] Etherington. "Apple Patents A VR Headset For Iphone", [WWW], [Referenced 20.12.2015].  
Available: <http://techcrunch.com/2015/02/17/apple-patents-a-vr-headset-for-iphone/>
- [7] Cukor, Judith, et al. "Virtual Reality Exposure Therapy for Combat-Related PTSD." *Posttraumatic Stress Disorder and Related Diseases in Combat Veterans*. Springer International Publishing. 69-83, 2015.
- [8] Standen, Penny, et al. "Patients' use of a home-based virtual reality system to provide rehabilitation of the upper limb following stroke." *Physical therapy* 95.3 : 350-359, 2015.
- [9] Bowman, Ellen, and Lei Liu. "Low Vision Patients Can Transfer Skills They Learned From Virtual Reality to Real Streets." *Investigative Ophthalmology & Visual Science* 56.7: 2622-2622. 2015.
- [10] Moeslund, Granum. "A survey of computer vision-based human motion capture." *Computer vision and image understanding* 81.3: 231-268, 2001.
- [11] Hecht. "Photonic Frontiers: Gesture Recognition: Lasers Bring Gesture Recognition to the Home." *Laser Focus World*: 1-5, 2011.  
Available: <http://www.laserfocusworld.com/articles/2011/01/lasers-bring-gesture-recognition-to-the-home.html>
- [12] Moynihan, "CMOS Is Winning the Camera Sensor Battle, and Here's Why" [WWW], [Referenced 06.11.2015].



Available:

[http://www.techhive.com/article/246931/cmos\\_is\\_winning\\_the\\_camera\\_sensor\\_battle\\_and\\_heres\\_why.html](http://www.techhive.com/article/246931/cmos_is_winning_the_camera_sensor_battle_and_heres_why.html)

- [13] Reza, “Kinect With Nightshot” (VIDEO) [WWW], [Referenced 06.11.2015]. Available: <https://www.youtube.com/watch?v=nvvQJxgykcU>
- [14] “Mundie: Microsoft's Research Depth Enabled Kinect” [WWW], [Referenced 15.8.2015]. Available: <http://www.pcworld.com/article/202184/article.html>
- [15] “Inside Xbox 360's Kinect Controller” [WWW], [Referenced 15.8.2015]. Available: [http://www.eetimes.com/document.asp?doc\\_id=1281322](http://www.eetimes.com/document.asp?doc_id=1281322)
- [16] Shpunt, Zalevsky, “Depth-varying light fields for three dimensional sensing”, Patent US 20080106746 A1, 2008. Available: <http://www.google.com/patents/US20080106746>
- [17] Structure Sensor [WWW], [Referenced 11.11.2015]. Available: <http://structure.io/>
- [18] Medina, “Three dimensional camera and range finder”, Patent US 5081530 A, 1992. Available: <http://www.google.com/patents/US5081530>
- [19] “Microsoft to consolidate the Kinect for Windows experience around a single sensor” [WWW], [Referenced 15.8.2015]. Available: <http://blogs.msdn.com/b/kinectforwindows/archive/2015/04/02/microsoft-to-consolidate-the-kinect-for-windows-experience-around-a-single-sensor.aspx>
- [20] Xbox One technical specification [WWW], [Referenced 06.11.2015]. Available: <https://dev.windows.com/en-us/kinect/hardware>
- [21] The Kinect for Windows Team, “Detecting heart rate with Kinect” [WWW], [Referenced 06.11.2015]. Available: <http://blogs.msdn.com/b/kinectforwindows/archive/2015/06/12/detecting-heart-rate-with-kinect.aspx>
- [22] Xbox-One-Kinect.jpg [WWW], [Referenced 06.11.2015]. Available: <https://en.wikipedia.org/wiki/File:Xbox-One-Kinect.jpg>
- [23] Xu, Ahuja, “On the use of depth-from-focus in 3d object modelling from multiple views”, *Department of Electrical and Computer Engineering and Beckman Institute University of Illinois at Urbana- Champaign, Urbana, IL61801, USA*, 2004.

- [24] DOF-ShallowDepthofField.jpg [WWW], [Referenced 06.11.2015].  
Available: <https://commons.wikimedia.org/wiki/File:DOF-ShallowDepthofField.jpg>
- [25] Spring, Davikson, "Astigmatism" [WWW], [Referenced 20.10.2015].  
Available: <http://www.microscopyu.com/tutorials/java/aberrations/astigmatism/>
- [26] Freedman, Shpunt, Arieli, "Distance-Varying Illumination and Imaging Techniques for Depth Mapping", Patent US 20100290698 A1, 2010.  
Available: <http://www.google.com/patents/US20100290698>
- [27] Calin, Roda. "Real-time disparity map extraction in a dual head stereo vision system." *Latin American applied research* 37.1: 21-24, 2007.
- [28] Battle, Joan, E. Mouaddib, and Joaquim Salvi. "Recent progress in coded structured light as a technique to solve the correspondence problem: a survey." *Pattern recognition* 31.7 (1998): 963-982.
- [29] Peterson, "Computing Constrained Delaunay Triangulations" [WWW], [Referenced 30.10.2015].  
Available: [http://www.geom.uiuc.edu/~samuelp/del\\_project.html](http://www.geom.uiuc.edu/~samuelp/del_project.html)
- [30] Coxeter, Greitzer. *Geometry revisited*. Vol. 19. 51-56, 1967.
- [31] Guibas, Leonidas, Stolfi. "Primitives for the manipulation of general subdivisions and the computation of Voronoi." *ACM transactions on graphics (TOG)* 4.2: 74-123, 1985.
- [32] Binary search tree, [WWW], [Referenced 09.11.2015].  
Available: [https://en.wikipedia.org/wiki/File:Binary\\_search\\_tree.svg](https://en.wikipedia.org/wiki/File:Binary_search_tree.svg)
- [33] Marlon, *Focus On Photon Mapping (Premier Press Game Development)* 111, 2003.
- [34] Bolitho, Grant. "The Reconstruction of Large Three-dimensional Meshes". Johns Hopkins University, 2010.
- [35] Voxel Grid, [WWW], [Referenced 09.11.2015].  
Available: <https://en.wikipedia.org/wiki/Voxel#/media/File:Voxels.svg>
- [36] Nealen. "An as-short-as-possible introduction to the least squares, weighted least squares and moving least squares methods for scattered data approximation and interpolation." [WWW], [Referenced 30.10.2015].  
Available: <http://www.nealen.com/projects>
- [37] Lancaster, Salkauskas. "Surfaces generated by moving least squares methods." *Mathematics of computation* 37.155 (1981) : 141-158, 1981.

- [38] “Smoothing and normal estimation based on polynomial reconstruction” [WWW], [Referenced 30.10.2015]. Available: <http://pointclouds.org/documentation/tutorials/resampling.php>
- [39] Edelsbrunner, Herbert, and Ernst P. Mücke. "Three-dimensional alpha shapes." *ACM Transactions on Graphics (TOG)* 13.1: 43-72, 1994.
- [40] Edelsbrunner. *Weighted alpha shapes*. University of Illinois at Urbana-Champaign, Department of Computer Science, 1992.
- [41] Amenta, Choi, Kolluri. "The power crust." *Proceedings of the sixth ACM symposium on Solid modeling and applications*. ACM, 2001.
- [42] Lorensen, Cline. "Marching cubes: A high resolution 3D surface construction algorithm." *ACM siggraph computer graphics*. Vol. 21. No. 4. ACM, 1987.
- [43] Bourke, “Polygonising a scalar field” [WWW]. [Referenced 30.10]. Available: <http://paulbourke.net/geometry/polygonise/>
- [44] Holz, Behnke. "Fast range image segmentation and smoothing using approximate surface reconstruction and region growing." *Intelligent Autonomous Systems 12*. Springer Berlin Heidelberg. 61-73, 2013.
- [45] Marton, Csaba, Rusu, Beetz. "On fast surface reconstruction methods for large and noisy datasets." in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2009.
- [46] Li, Liu, Phan, Abeysinghe, Grimm, Ju, Polygonizing extremal surfaces with manifold guarantees, *Proceedings of the 14th ACM Symposium on Solid and Physical Modeling*, 2010.
- [47] Ju, Tao, et al. "Dual contouring of hermite data." *ACM Transactions on graphics (TOG)* 21.3: 339-346, 2002.
- [48] Kazhdan, Bolitho, Hoppe. "Poisson surface reconstruction." *Proceedings of the fourth Eurographics symposium on Geometry processing*. Vol. 7, 2006.
- [49] Point Cloud Library API Documentation [WWW], [Referenced 15.8.2015] Available: <http://docs.pointclouds.org/1.7.2/>
- [50] Robot Operating System (ROS), [WWW], [Referenced 06.11.2015]. Available: <http://www.ros.org/about-ros/>
- [51] “Normal, Gaussian and Mean Curvatures at the Regular Point of a Surface”, [WWW], [Referenced 09.11.2015]. Available: [http://www.grad.hr/itproject\\_math/Links/sonja/gausseng/introduction/introduction.html](http://www.grad.hr/itproject_math/Links/sonja/gausseng/introduction/introduction.html)

- [52] OpenNI 2 Framework [WWW], [Referenced 11.11.2015].  
Available: <http://structure.io/openni>
- [53] Armstrong, “OpenNI To Close” [WWW], [Referenced 11.11.2015].  
Available: <http://www.i-programmer.info/news/194-kinect/7004-openni-to-close-.html>
- [54] Boost C++ source libraries [WWW], [Referenced 11.11.2015].  
Available: <http://www.boost.org/>
- [55] Eigen C++ template library [WWW], [Referenced 11.11.2015].  
Available: <http://eigen.tuxfamily.org/>
- [56] FLANN [WWW], [Referenced 11.11.2015].  
Available: <http://www.cs.ubc.ca/research/flann/>
- [57] VTK [WWW], [Referenced 11.11.2015].  
Available: <http://www.vtk.org/>
- [58] PLY – Polygon File Format, [WWW], [Referenced 06.11.2015].  
Available: <http://paulbourke.net/dataformats/ply/>
- [59] VTK File Formats, [WWW], [Referenced 06.11.2015].  
Available: <http://www.vtk.org/wp-content/uploads/2015/04/file-formats.pdf>
- [60] Raggett, 1994, “Extending WWW to support Platform independent Virtual Reality” [WWW], [Referenced 06.11.2015].  
Available: <http://www.w3.org/People/Raggett/vrml/vrml.html>
- [61] Brutzman, Daly. “X3D: "extensible 3D graphics for Web authors.”, *Morgan Kaufmann*, 2010.
- [62] PCL Tutorials, [WWW], [Referenced 09.11.2015].  
Available: <http://pointclouds.org/documentation/tutorials/>
- [63] ParaView [WWW], [Referenced 11.11.2015].  
Available: <http://www.paraview.org/>
- [64] Unity3D, [WWW], [Referenced 10.01.2016].  
Available: <https://unity3d.com/>