REALTIME 3D MAPPING, OPTIMIZATION, AND RENDERING BASED ON A DEPTH SENSOR

By

CRAIG MOUSER

Bachelor of Science in Computer Engineering

Kansas State University

Manhattan, KS

2011

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 2012

REALTIME 3D MAPPING, OPTIMIZATION, AND

RENDERING BASED ON A DEPTH SENSOR

Thesis Approved:

Dr. Weihua Sheng

Thesis Adviser

Dr. Damon Chandler

Dr. David Cline

Name: CRAIG MOUSER

Date of Degree: DECEMBER, 2012

Title of Study: REALTIME 3D MAPPING, OPTIMIZATION, AND RENDERING BASED ON A DEPTH SENSOR

Major Field: ELECTRICAL AND COMPUTER ENGINEERING

Abstract: This thesis provides a method for using a portable scanner to create an optimized 3D map for real time rendering. This thesis uses a cloud computing software as a service architecture which allows for a portable scanner to acquire depth maps. Using a portable scanner allows for the mapping of large areas. It will then send the depth maps to a server for a 3D map to be created in real time. This thesis discusses the acquisition of the point cloud using an open source program for 3D mapping with a depth sensor. It then covers the triangulation into a mesh using the marching cubes algorithm. The optimization of the mesh to allow real time rendering is then introduced. Finally the mesh is imported and rendered in the Unreal Engine 3 for an interactive and intuitive display.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Figure Page

CHAPTER I

INTRODUCTION

In this thesis, a method is developed for the capturing, representing, and rendering of 3D environments based on a depth sensor. This chapter will first cover the motivation in Section 1.1. Related work will then be presented in Section 1.2. Finally, an overview of this thesis will be covered in Section 1.3.

1.1 Motivation

The ability to create 3D maps has been a popular research topic for many years. 3D maps have intrinsic values in many different fields of study from robotics to visualization. Typically, these maps have been created by reading in data from one or more types of sensors. These readings are then processed into a map of their surroundings. Depending on the application requirements, a 2D or 3D map could be created. Most often, these maps have been created from either a LIDAR (Light Detection and Ranging) sensor or SONAR (Sound Navigation and Ranging) sensor. However, since Microsoft released the Kinect sensor in November 2010 [1], using RGB-D images to create 3D maps has been an aggressively researched topic [4][8][9][14][15][17][19][20][21][22].

Map representation is an important issue. Using a mesh to represent 3D maps is valuable in many applications. By triangulating a 3D map, it enables the use of decades of research into polygon rendering. Industry has provided a huge drive of research into this area because of the popularity of video games and computer generated imagery (CGI). All modern rendering engines are based on the concept of rendering a series of triangles into a 2D image. We now have tools to create photo-realistic renderings, process very large meshes, apply physics simulations and more. We can use these tools to create interactive and visually appealing displays for our 3D maps. One application where this would be useful is in real estate. A property's interior could be scanned into a 3D map and uploaded online. Potential buyers would then be able to interactively tour the property in its entirety. This would give a much more natural feeling than viewing panoramic images of the interior of the house. Figure 1 shows a rendering of an interior room based on a triangulated mesh. Similarly, robots can use this technology to create maps that can help autonomous navigation and be easily viewed by operators. This can be very useful for search and rescue, as well as reconnaissance. It would be easy to identify and mark areas of interest on an intuitively displayed 3D map.

This thesis aims to develop a portable 3D mapping system using a depth sensor. Three main issues are involved in this process: map creation, map optimization, and map rendering. Map creation is the process of acquiring the point cloud representing the 3D space from a sequence of depth images. Map optimization is the triangulation of the point clouds in to a mesh and the optimization of the triangulated mesh. Map rendering is creating a visual display of the optimized map for a viewer.

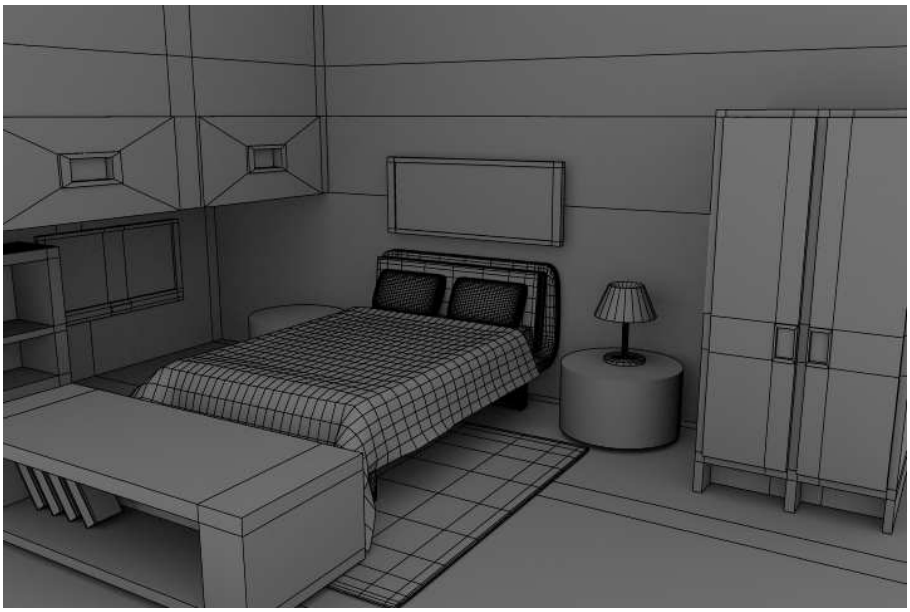

Figure 1: An interior represented by a hand created 3D mesh [2].

There are several constraints that should be met in order to create a practical 3D mapping application. These constraints are:

- Portable: The system should allow the user to move freely through their environments to acquire complete maps.
- Cost: The price of the system should be reasonable for its purpose. For this paper we are proposing a method that could be used by the general population. This means that the system should be affordable to an average consumer.
- Accuracy: The system should be able to create the map with the accuracy required for various 3D applications.
- System Requirements: The system should only use off- the-shelf parts that are available to all potential users.

Depth sensors such as the Kinect sensor or the Xtion sensors [42][43] provide a cheap sensor for creating 3D maps with the required accuracy. LIDAR sensors can provide very accurate

results, but cost several thousand dollars. SONAR on the other hand is very cheap, but has poor accuracy. The Kinect provides ample accuracy while costing only a small fraction of what LIDAR costs [3].

1.2 Related Work

This section will briefly discuss recent works that are related to this thesis. As previously mentioned, there has been a lot of research on the use of RGB-D images since the release of the Kinect sensor.

1.2.1 RBG-D Mapping

Henry *et al.* introduced a method for mapping through the creation of a dense point cloud from RGB-D images [4]. They created these maps by combining both visual features and shape based alignment. Their implementation detects visual features using the scale-invariant feature transform (SIFT) algorithm [5]. It uses an RGB-D iterative closest point (ICP) algorithm [6] for the shape based alignment. The RGB-D ICP is an enhancement of ICP specialized for the RGB-D sensors. It uses the SIFT feature points projected into 3D using the depth data as 3D features. The random sample consensus (RANSAC) [7] algorithm is used to find features that are considered inliers between the source and target frame. ICP is then used to determine the transformation from the previous frame. They also use visual and depth information on key frames to detect loop closures and applied correction for drift. A key frame is created any time there are less than 3 inliers found between the current frame and last key frame. Each time a key frame is added, it is compared to previous key frames in an attempt to detect loop closure. Figure 2 shows two dense point clouds from their experiments. This implementation can run at approximately one frame per second without merging points.

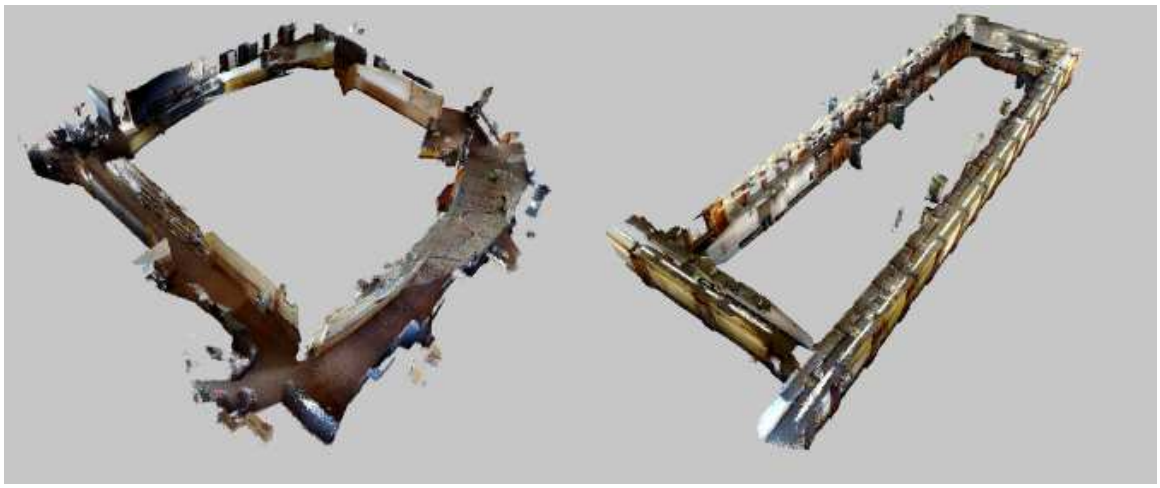

Figure 2: Dense point clouds created with RGB-D Mapping [4].

Huang *et al.* combined the previous work with a flying quadrotor and was able to create dense 3D point cloud maps for localization using only the onboard computer [8]. The dense point cloud map was also used for stabilizing and controlling the flight of the quadrotor. Engelhard *et al.* implemented a similar solution to Henry's to enable real-time 3D SLAM (Simultaneous Localization and Mapping) with an RGB-D camera [9]. In their implementation they used speeded up robust feature (SURF) [10] feature extraction on the RGB image combined with RANSAC to estimate the relative transformation of the camera. They then refine this estimate with a second estimate from a variant of the ICP algorithm. This is all stored using a pose graph optimization [11] to help form a globally consistent map. However, their implementation is only able to run at approximately 2 seconds per frame. All of the previous work mentioned above has used dense point clouds to store their 3D maps. Dense point clouds can use a large amount of memory when working with large areas.

The following papers created similar systems using various methods to reduce memory usage. Hartmann *et al.* presented a method of RGB-D SLAM using a Kinect and two feature extraction algorithms, Oriented FAST [12]and Rotated BRIEF (ORB) [13][14]. In their paper they use a landmark based mapping system that is required for their FastSLAM algorithm. In this

method, it stores a series of landmarks which represent a point of interest. Each landmark will hold the 3D position, the error covariance, and a descriptor. While this implementation is very fast and memory efficient, it does not retain the necessary information to create a realistic rendering. In another paper, Endres *et al.* evaluated RGB-D SLAM using three different algorithms [15]. They tested SIFT, SURF, and ORB. They stored their results in an OctoMap, an octree-based voxel map proposed by Wurm *et al.* [16]. Their research showed similar accuracies for SURF and SIFT, while ORB did not perform quite as well. ORB and SIFT had similar run times that were much faster than SURF on average. The SIFT implementation was executed on the GPU however. Zou *et al.* presented a method for indoor localization and 3D scene reconstruction using the Kinect [17]. Their research also decided to use the aforementioned OctoMap to efficiently deal with the large amounts of data. In their implementation they tested four feature detectors: ORB, SURF, Shi-Tomasi (ST) corners [18], and FAST. They did not utilize any GPU acceleration as the goal of their research was to run their software on mobile robotics platforms with limited computing power. ORB and SURF were the only two to complete both test environments successfully. FAST, however, was the fastest algorithm. It was closely followed by ORB and ST corners.

The first virtual reality applications came along when Izadi *et al.* of Microsoft Research released their paper on KinectFusion [19]. KinectFusion presented a surface mapping and tracking system based only on the depth frame. It utilizes a coarse-to-fine ICP algorithm for tracking. This implementation compares the current depth map to the entire observed model. This helps reduce drift over time that can be seen in other implementations. It also utilizes a highly parallelized set of algorithms, which allows almost all the software to be run on the GPU. The limitation of their implementation is it only works on a 3 meter cube. This makes it impractical for scanning large environments. However it did offer the greatest quality tracking and rendering. Figure 3 shows two examples of the original KinectFusion algorithm constructing models from

RGB-D images. This publication along with another on the same project [20] prompted a barrage of research on extending this system.

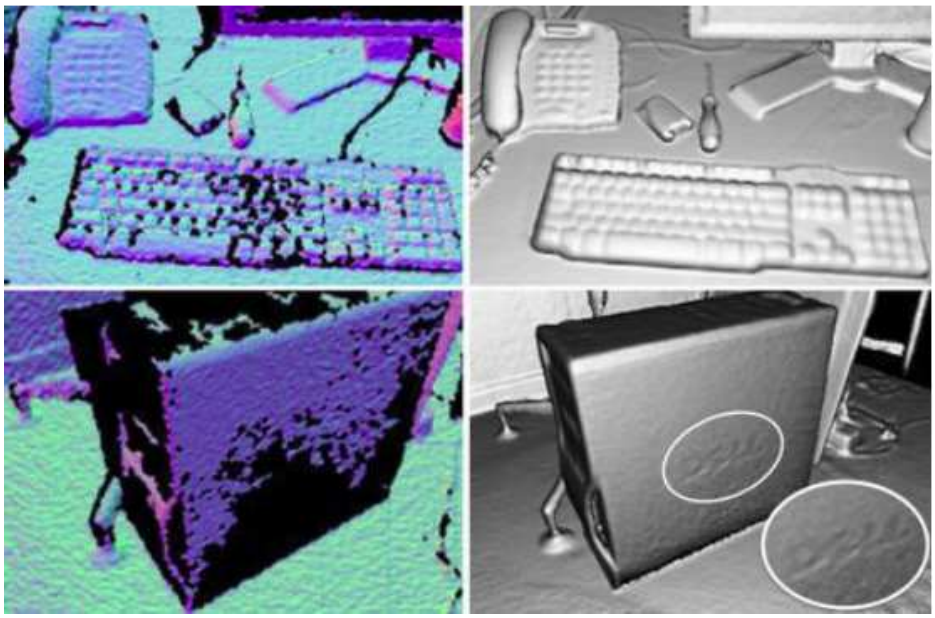

Figure 3: KinectFusion output. Left: Raw Kinect data with normals. Right: Constructed map. [19].

Whelan *et al.* implemented an extension to the KinectFusion algorithm named Kintinuous [21]. This system runs the KinectFusion algorithm on the 3 meter cube, but when the camera moves towards the edge of the cube it exports a dense point cloud and re-centers the cube on the cameras location. This is performed by treating the cube as a cyclical array. When the cube is re-centered, or shifted, the area that is no longer in the cube is exported to CPU memory and reassigned to the new area being shifted into the cube. Open source implementations similar to KinectFusion and Kintinuous will be discussed in greater detail in Section 3.1 of this thesis. Roth and Vona developed a similar approach to extending KinectFusion that also uses a moving cube [22]. Their implementation works by storing two cubes in GPU memory and swapping back and forth as the user moves. Therefore their implementation requires twice the memory of the Kintinuous implementation.

1.2.2 Triangulation

Triangulation has been a research topic for decades that spans many different fields. In this case triangulation of points in a dense point cloud is necessary in 3D space. This has been a widely researched area which has yielded many excellent solutions. In 1998 Owen provided a survey of the top triangulation methods [23]. These methods were Octree [24], Delaunay [25], and Advancing Front triangulation [26]. A method for efficient and constrained Delaunay triangulation in 3D space was introduced in 1994 By Weatherill and O'Hassan [27]. This method is still widely used today and is much more popular than the other two.

The method which was used in this thesis is known as marching cubes triangulation [28]. Marching cubes was introduced in 1987 by Lorenson and Cline as a method to triangulate models of constant density. It was originally designed to deal with medical data, but has since been expanded to computer graphics. This method was implemented using a GPU in 2006 by Johansson and Carr [29]. This research allowed GPUs to run the marching cubes algorithm in a fraction of the time. This is important for working with large data sets such as dense point clouds.

1.2.3 Mesh Simplification

Mesh simplification is another research topic that has been popular for years. When computer graphics were just beginning, the computing power did not exist to process large meshes. Many researchers created algorithms to simplify meshes. Cignoni, Montani, and Scopigno created a comparison of various mesh simplification methods in 1998 [30]. In their paper, the authors compared 21 existing methods for simplifying a mesh. They analyzed everything from number of triangles, runtime, local and global error, and memory usage. Each method was applied over various models to extensively test the strengths and weaknesses of each approach. The most consistent method was using quadric error metrics. Multi-resolution decimation provided similar

quality but had nearly 20 times the runtime. Since their paper was published, using quadric error metrics has been the most popular method for mesh optimization.

Using quadric error metrics for mesh optimization was first used by Garland and Heckbert [31]. Their method is capable of creating high quality approximations of polygon meshes in a relatively small amount of time compared to similar methods. Their method works by iteratively contracting vertex pairs and merging them. They would even conjoin vertices that were not connected by an edge. The most important aspect however is the use of the quadric error metric. The algorithm works by computing a quadric error matrix for each vertex. It then selects all valid pairs. Each pair is given a contraction cost where the lower the cost means it is a more ideal contraction. The pairs are then sorted from lowest to highest and the contractions are iteratively performed until it reaches a stop condition. After each contraction, affected vertices will have their costs recalculated. Figure 4 shows an example of Garland and Heckberts results using their method.

Figure 4: Mesh optimization of a model from 5,804 to 994, 532, 248, and 64 faces respectively using the quadric mesh optimization method [31]

1.2.4 Rendering Engines

Trenholme and Smith published a survey of using computer game engines for visualizing first person virtual environments in 2007 [32]. They surveyed several key aspects of the 6 most popular engines at the time: CryENGINE [33], id Tech 3 and 4 [34], Jupiter EX, Source [35], and Unreal 2 [36]. The id Tech 3and 4 engines as well as the Jupiter EX engine are no longer available. CryENGINE, Source, and Unreal Engine are still widely used and supported. Each of these was found to have similar qualities by Trenholme and Smith. All three have very wide

community support, however CryENGINE and Unreal 2 were praised for their graphics abilities while Source was praised for its ability for modularity, scripting, lip synchronization, and more game specific features that are not necessary in this thesis.

Another engine that has been developed is the Unity 3D Engine [37]. It is a cross platform game engine that supports several programming languages, 3D software packages, and many operating systems including Windows, Mac, iOS, Android, and others. It is a newer engine than the others mentioned so far and has less community support. It is growing quickly however. At this time its graphics quality does not meet the levels of CryENGINE and the Unreal Engine. The cross platform capabilities of this engine make it a potential solution in the future.

However, for this thesis the Unreal Engine was chosen because of its graphics quality and wide community support. It also has a new version and many new features that have been added since Trenholme and Smith's survey in 2007. One key feature is that the engine can now publish games for mobile devices, similar to the Unity engine. There are existing apps for both iOS and Android that utilize the Unreal Engine and have very realistic rendering. The graphics quality has also progressed to such a realistic level that it is being used for films and television shows, as well as visualization by architectural firms. Figure 5 shows the realism that is achievable with the Unreal Engine.

Figure 5: Screenshot from the Unreal Engine 3 [36]

1.3 Overview of the Thesis

This thesis implements a portable 3D mapping system for large indoor environments. The system will utilize both a client and server computer. The client computer will interface with the depth sensor needed for mapping the environment. The depth image will be streamed to the server where the computation to create the 3D map will be performed. An image will then be streamed back to the client computer to provide a real time status update of the map creation. The client computer will therefore only need the capability to upload 16 bit video while downloading 24 bit video. These connection speeds are typical and internet speeds are increasing regularly. The resulting map can then be shared and viewed in a rendering engine after mesh simplification. This system will meet all the criteria previously stated for this project as explained below:

- Portable: A laptop or netbook using wireless internet would allow the user to explore and map their environment without restriction.

- Cost: A laptop or netbook is very affordable and already commonly owned. RGB-D sensors are also affordable. An Xtion sensor and netbook can be purchased for $120 and $230 dollars respectively [38][39]. It is also very common for users to already have an existing wireless internet connection.
- Accuracy: The implementation will provide accuracy that is suitable for large environments by utilizing the methods from previous work.
- System Requirements: All components necessary for the user are widely available off-the-shelf.

This thesis is organized as follows: Chapter 2 will present the hardware setup for the map creation system. It will discuss both the user's end and the cloud based computation, as well as the networking connecting them. Chapter 3 will discuss the software created in this project. Chapter 4 will discuss and present the results the experiment performed for this project. Chapter 5 will discuss the conclusions and future work.

CHAPTER II

HARDWARE PLATFORM

This chapter will describe the different types of hardware used in this project. There are several key hardware components for this thesis. The overall system setup will be discussed in section 2.1. The importance of each individual piece of hardware will then be discussed. Section 2.2 will cover the RGB-D sensor. Section 2.3 will discuss the portable client computer. Section 2.4 will discuss the server computer and the networking.

2.1 Overall System Setup

The system designed for this thesis is composed of two main parts. The first part is the portable scanner, which in turn is composed of two parts. The first part is the RGB-D sensor to capture the depth images necessary for creating the 3D maps. The second part is the computer the user will need to interface with the RGB-D sensor and to communicate with the server computer. They will also need to have an active internet connection to communicate with the cloud computing service. The second main component is the server side computer. It is the computer that runs the cloud computing software. It must have a CUDA enabled GPU in order to run the software needed for this thesis. It must also have access to an internet connection to communicate with the client computer. Figure 6 shows a block diagram of the overall setup of this system.

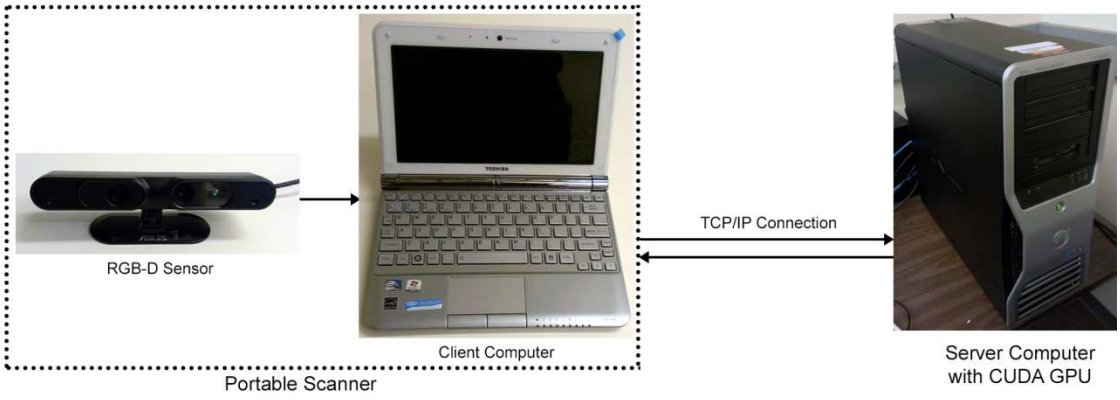

Figure 6: A block diagram of the system setup

The overall system works as follows. The client computer will capture a stream of depth images from the RGB-D sensor. It will then compress the images and upload them over a TCP/IP connection to the server computer. The server computer will use the depth images to build up a 3D map. During this process the server will generate a stream of compressed images showing the current 3D map so the user can see the progress of the scan. This will also allow the user to see what areas need to receive more attention to create a complete map. When the process is complete, the server will generate the 3D mesh file. This mesh file can be optimized by the client using the quadric error optimization method. This allows the user to select a level of optimization that suites their purpose. This process could be automated to provide a set amount of optimization if that is preferred. The file could then be shared and displayed using a game engine such as the Unreal Engine.

2.2 RGB-D Sensor

The first popular RGB-D sensor released was Microsoft's Kinect sensor [40]. It was originally designed for use playing video games with Microsoft's X-Box, but has since been used for many other applications. The Kinect sensor uses a video frame rate of 30 frames per second and a resolution of 640 x 480. The RGB stream uses standard 8 bit/channel or 24 bit color. The depth

sensor uses 11 bit depth which provides 2048 levels of depth [41]. To determine depth the Kinect sensor emits a series of infrared dots. It then uses the deformation of these dots in the image to calculate the depth map. Because all the parameters of the camera are known, the Kinect is capable of calculating absolute depth from the sensor, as opposed to the relative depth of each pixel. Figure 7 shows an example of a Kinect sensors output while looking at a typical desk. The Kinect struggles to calculate the depth of objects too close to the sensor or that are too bright, such as the monitors. This issue causes problems for the Kinect in daylight. The infrared dots are not strong enough to be picked up in direct sunlight. It may also struggle with reflective or clear objects. You can also see there is a shadow in the depth image behind the monitor because the infrared emitter and camera are slightly offset within the Kinect device.

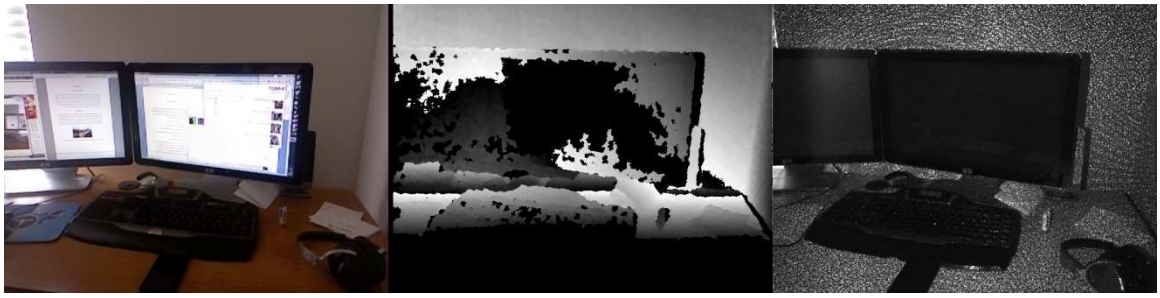

Figure 7: Kinect sample output. Left: RGB image. Center: Depth image. Right: Infrared pattern.

Following the release of the Kinect, Asus released two similar sensors. These sensors are named the Xtion Pro and Xtion Pro Live [42][43].The Xtion Pro is only capable of sending the depth image, while the Xtion Pro Live contains depth image, RGB image, and audio information. A key difference between the two Xtion sensors and the Kinect sensor is the Xtion sensors only require a USB connection. The Kinect requires a USB and AC power connection. This makes the Xtion sensors ideal for this project because they can be carried around with a laptop. They will not be constrained to being plugged into a wall. The Xtion sensors are also more compact and lighter than the Kinect. These three sensors can be seen side by side in Figure 8. This figure helps provide an idea of the difference in size between the three sensors.

Figure 8: Left: Microsoft Kinect. Center: Asus Xtion Pro. Right: Asus Xtion Pro Live

All three of these devices are compatible with the OpenNI drivers [44]. OpenNI is an open source framework for Natural Interaction (NI). It is an industry-led, not-for-profit organization that maintains this framework. The OpenNI framework supports both Windows and Ubuntu. The OpenNI framework provides code to receive streams of depth maps, RGB images, infrared images, and others such as audio, hand position, and gesture tracking. Any code developed using these drivers can use any of the three sensors interchangeably without adjusting the code.

2.3 Portable Client Computer

The client computer is the device the user will use to obtain the depth images needed for creating the 3D map. It must be capable of meeting a number of requirements. It must be portable so that the user can easily scan large environments. It must be able to connect to the RGB-D device through USB. It should also be able to connect wirelessly to the internet. This could include the 802.11 standard or a cellular data network. Finally, it should have the processing power and memory requirements to compress and upload the depth sensor image and download, decompress, and display the status image if necessary. For this thesis many possibilities for the portable client computer were researched. Only one configuration was implemented for this thesis, the netbook. First, using a netbook will be discussed in section 2.3.1. Then alternate

configurations will be discussed such as the Fit PC2i in Section 2.3.2, the single-board computer configuration in 2.3.3, and using a mobile device in 2.3.4.

2.3.1 Netbook

For the implementation of this thesis, a netbook was used for the client computer. A netbook is a relatively new category of computers that arose in 2007. They are a small laptop computer that is inexpensive, has extended battery life, and is much lighter. They often, however, have less computing power than their laptop counterparts. For this thesis, the Toshiba mini NB305 model netbook [45] is used. It has a 1.66 GHz Intel Atom N450 Processor, 1 GB of DDR2 Memory, and a 10.1 inch 720p resolution screen. It also has 3 USB 2.0 ports which can be used to connect the RGB-D device. Finally, it also has built in Wi-Fi wireless that can connect to 802.11b/g/n networks. 802.11n is an important feature as it is supports 300 Mbps wireless transfer speeds. This is useful for the streaming of the depth images and status images. For this thesis, Ubuntu 10.04 is used for the operating system. This netbook can be seen in Figure 9.

Figure 9: The Toshiba NB305 netbook used in this thesis

Using a netbook for this thesis is very convenient for many reasons. It meets all the requirements stated in Section 1.1. Netbooks are inherently portable by design. They are lightweight and easy to carry around which allows the user to map large 3D areas. They are also already owned by many people. Alternatively, a full-size laptop could be used in place of the netbook with little or no change to the system. A laptop has higher performance in every category except battery life and portability. Laptops are still very portable, but are slightly larger and heavier than netbooks.

2.3.2 Fit PC2i

The FitPC2i is a small form computer that uses very little power [46]. It packs a full computer's capability into a 4" x 4.5" x 1.05" case. At the same time it uses 94% less power than a standard desktop computer. It uses from 6-8 watts when in use and only 1 watt on standby. The Fit PC2i has an Intel Atom Z530 at 1.6 GHz. This is an older model than the N450 that was in the Toshiba netbook. It also has 1GB of memory. It has 4 USB 2.0 ports, though 2 of them use the mini-USB connections. It also has built in 802.11n wireless support. The Fit PC2i can be seen in Figure 10 below.

The Fit PC2i uses a 12V DC input. This makes it convenient to use a battery to power the device. By using a battery we make the device much more portable. The weakness to this design is there is no screen to display the current status image. This makes it difficult for a person to map an environment. However, this could be ideal for a robotics application. The depth sensor could be attached to a robot and the status image could be displayed in a remote location to a user.

Figure 10: The Fit PC2i with a battery and an Asus Xtion Pro Live.

2.3.3 Single-Board Computer

Single-board computers are complete computers built onto a single circuit board. This includes the processor, memory, and many IO devices. These types of systems have grown in popularity recently. Some of the more notable single-board computers are the Arduino [47], BeagbleBoard [48], and the Raspberry Pi [49]. These are just a few of the most popular single-board computers, but there are many more. Each of these boards has their own unique qualities that would need to be considered for implementing this thesis. The advantage to using a single-board computer is that you could create a very small, low-power, standalone device for 3D mapping. Most single-board computers use DC power supplies and use a relatively low amount of power. The single-board computer could be powered by battery similar to the Fit PC2i in Section 2.3.2. Figure 11 shows an example of how this system could be configured. It is an in progress project that is being built in our ASCC lab that combines a Beagleboard xM with an

Asus Xtion Pro sensor for tracking gestures. By reversing the direction of the camera it could easily be repurposed for this thesis.

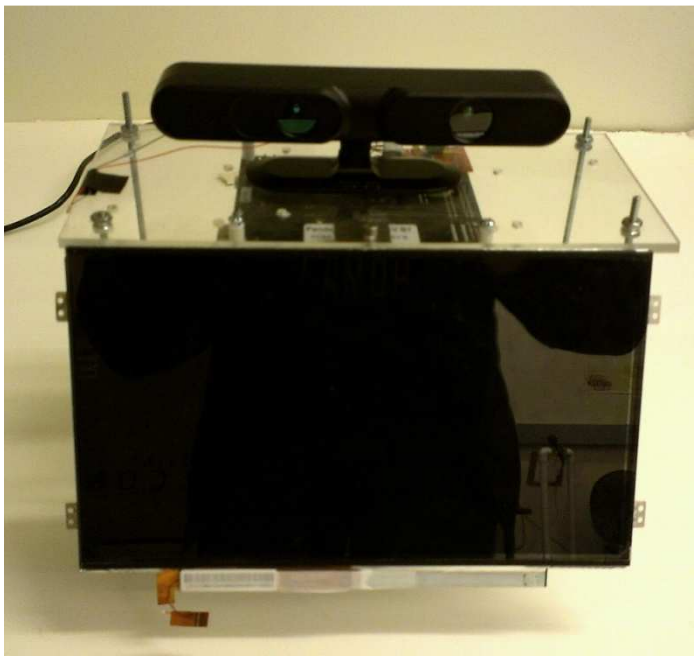

Figure 11: A single-board computer with an Asus Xtion Pro.

2.3.4 Mobile Device

It would be desirable to be able to use a mobile phone or tablet as the client computer. Almost everyone these days has a smart phone or tablet. Worldwide tablet sales are estimated to be near 119 million in 2012 [50]. This is following 60 million sold in 2011. This is in addition to the 472 million smart phones that were sold in 2011 [51]. Smart phones and tablets are beginning to replace netbooks or even laptops for many people. These mobile devices meet all the requirements set forth in Section 1.1. They are extremely portable, have long battery life, and are very easily obtainable. However, each device has its own set of specifications that may or may not meet the system requirements for this project. Most modern devices should have the required computing capabilities and network speed for this thesis. They also typically have a micro-USB connection that could connect to the Kinect or Xtion devices using an adapter. Raymond Lo has

been able to interface both the Kinect and Xtion sensors with an Android device and perform finger tracking in real time [52][53]. He is using an NVIDIA developer tablet with a Tegra 3 processor, which is considered state of the art at this time. Figure 12 shows a possible configuration of this thesis using the Motorola Xoom tablet.

Figure 12: A Motorola Xoom with an Asus Xtion Pro Live.

2.4 Server Computer

The server computer will have a few tasks. Its first task is to receive the stream of depth images from the client computer. It should decompress the stream and use the images for developing the 3D map. As the map is being developed in real time, the server computer should be able to compress the current status image and stream it back to the client computer or display it if necessary. It is required that the server computer should use a CUDA enabled GPU. This is necessary to run the software used in this thesis project.

2.4.1 Server Hardware

For this thesis, a workstation PC in the ASCC lab is used to act as the server computer. This computer is the Dell T7500 Workstation. This workstation is running two Intel Xeon X5570 quad-core processors at 2.93 GHz. It has 12 GB of RAM. We swapped the NVDIA Quadro 4800 with the NVIDIA GTX 560 Ti GPU. The workstation is running Ubuntu 12.04 LTS 64-bit. This workstation is connected through Ethernet to a wireless router. This workstation meets the performance requirements for this thesis.

2.4.2 CUDA enabled Graphics Processing Unit (GPU)

To utilize the research published in the KinectFusion papers by Microsoft [19][20], it is necessary to perform a large amount of processing on a GPU. The most popular method for programming on a GPU is to use CUDA . CUDA is a parallel computing platform and programming model invented by NVIDIA [54]. It allows programmers to simply convert their code to utilize parallel computing across hundreds or thousands of cores. CUDA has been widely used in the scientific community since its release. Many computationally intensive programs can greatly improve by using parallel processing. Most of the algorithms used in this project fall into this category. CUDA allows you to send C, C++, or Fortran code directly to the GPU for processing.

GPUs are ideal for parallel programming because they are designed to do graphical calculations on each pixel. This made a highly parallelized architecture ideal. The CUDA architecture is composed of grids, blocks, and threads. A grid is created for each function, or kernel, that is called. The grid is assigned to a device and split up into many blocks. Each block contains many threads. A block is assigned to a streaming multiprocessor (SMP). Each SMP may process several blocks. This is done to allow latency hiding. If one block is waiting on a memory operation, the GPU will perform calculations on another block while it is waiting. Each thread

within the block is assigned to a single processor (SP). Each SMP contains several SPs allowing multiple threads to be computed simultaneously. Figure 13 provides an example of how grids, blocks, and threads are divided.

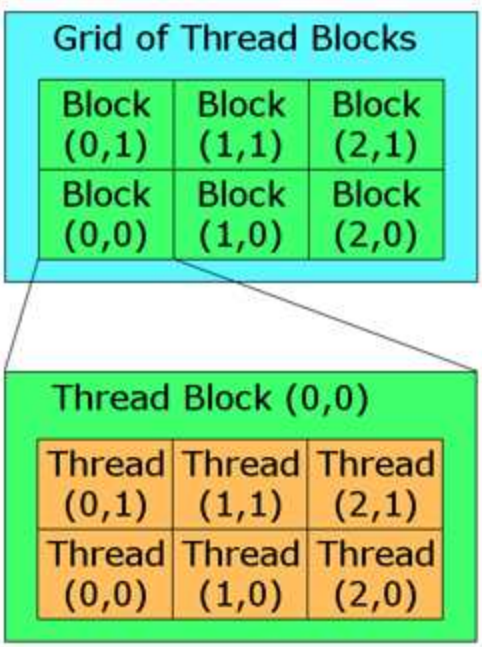

Figure 13: Example of CUDA Grid, blocks, and threads [55].

To use CUDA in your applications, you must have a CUDA enabled NVIDIA GPU. Every NVIDIA GPU created since the 8000 series in 2006 is compatible with CUDA. However, different GPUs have different compute capabilities ranging from 1.0 to 3.5. The newer cards with higher compute capabilities can make use of the newer features of the CUDA architecture. For this project a compute capability of 2.0 or higher is used. In our testing we used both a GTX 480 and GTX 560 Ti graphics cards.

2.4.3 Networking

This thesis uses a relatively simple networking setup based off existing technologies. By the NIST definition of cloud computing, this project would be considered Cloud Software as a

Service (SaaS) by providing a consumer the ability to use the provider's applications running on a cloud infrastructure [56]. By performing the processing in the cloud we are able to decrease the system requirements of the user to a low end computer and an internet connection. This makes the system much more practical to users. Instead of purchasing an expensive system, they can use an existing computer they already own. Performing computing in the cloud is a relatively new concept that is becoming more popular.

CHAPTER III

SOFTWARE PLATFORM

This chapter will present the various pieces of software used and created for this thesis project. First, in Section 3.1, KinectFusion will be discussed in greater detail. Triangulation will then be described in Section 3.2. In section 3.3 mesh optimization will be described. Section 3.4 will present the use of ROS (Robot Operating System) for network communication. Section 3.5 will cover the rendering of the final 3D map.

3.1 Point Cloud Aquisition

Microsoft published their research paper on KinectFusion, but did not provide the source code to the public. The Point Cloud Library (PCL) is a standalone, large scale, open source project for 2D/3D image and point cloud processing [57]. They implemented their own version of Microsoft's KinectFusion algorithm based off the information provided in the research papers. The KinectFusion algorithm in PCL is known as Kinfu and is expected to be released in version 1.7. They have also begun developing a KinfuLS or Kinfu Large Scale implementation which is similar to the research by Whelan *et al.* mentioned in the previous work section. This software is still in the process of being developed. Access to these two projects is currently available by

downloading the source code from the trunk of their SVN. The main difference between Microsoft's research and the PCL implementation is PCL does not use RGB information

The KinectFusion algorithms works through a complex set of steps. The first step is to run a bilateral filter on the depth image. The bilateral filter works to remove noise through a weighted average based on a Gaussian distribution while preserving hard edges. The depth image can then be converted into a point cloud, or a series of vertices in 3D space. This is a relatively simple calculation of mapping each pixel (x_d, y_d) to a 3D point, $P(x, y, z)$ using the following equations:

$$x = \frac{(x_d - c_{xd}) \times depth(x_d, y_d)}{f_{xd}}$$

$$y = \frac{(y_d - c_{yd}) \times depth(x_d, y_d)}{f_{yd}}$$

$$z = depth(x_d, y_d)$$

c_{xd} and c_{yd} are the cameras principle points. f_{xd} and f_{yd} are the focal points of the camera [58]. This creates the points in the camera's coordinate system. It must then be translated into the global coordinate system using the translation and rotation information from the ICP algorithm. The normal of each vertex is also estimated at this point. A GPU implementation of the Iterative Closest Point (ICP) algorithm is now run to estimate the transformation of the newest frame with 6 Degrees of Freedom (DOF). ICP finds correspondences between the new point cloud and the existing model. The ICP algorithm will also remove outlying vertices at this time by testing the Euclidean distance and angle. The point cloud is then integrated into the volumetric storage type. The data is stored in a truncated signed distance function (TSDF) based of Curless and Levoy's research [58]. The TSDF is a three dimensional grid divided into many voxels. By default, the TSDF is broken into 512x512x512 voxels and the grid is 3 meters. In my

implementation a 256x256x256 TSDF was used to decrease the amount of data. Finally, raycasting is then used to render an image of the TSDF for real time feedback to the user. Figure 14 shows a flowchart of this algorithm with example images of each step.

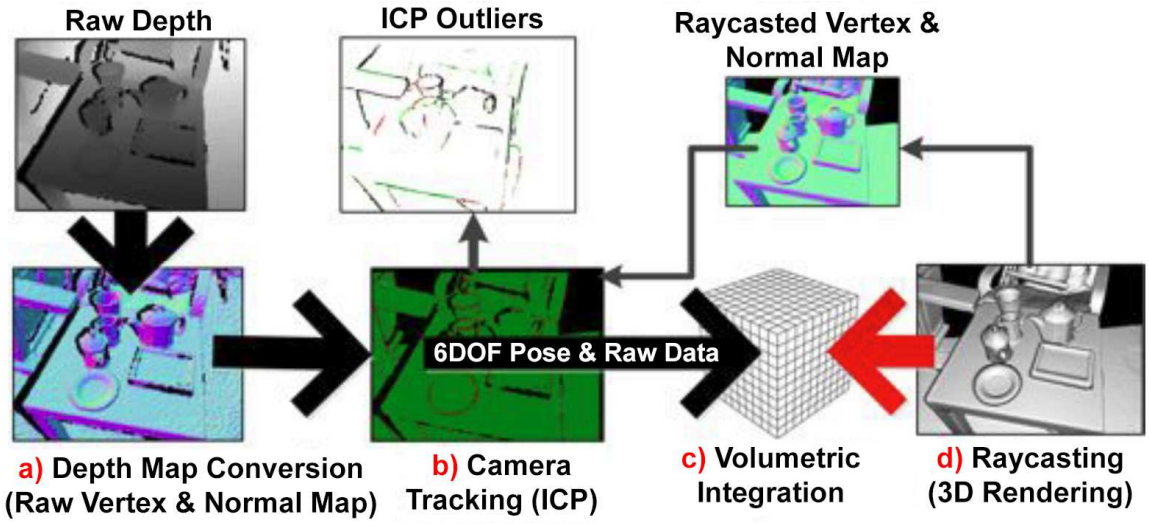

Figure 14: Flow chart of the KinectFusion algorithm [19].

While the KinectFusion algorithm worked well, it was bounded by the size of the TSDF volume. This made it very impractical for many applications as it is limited to about a three meter cube. KinfuLS looked to expand KinectFusion by allowing the user to shift the TSDF as the user moved. The TSDF became a three dimensional circular buffer. When the camera had moved a given distance away from the center of the TSDF, the TSDF would be realigned so that the camera would be in the center. The area that would no longer be inside the TSDF was converted to a point cloud and extracted from the GPU memory to main memory. Figure 15 shows a visual representation of this process. By allowing the TSDF volume to shift, much larger areas could be scanned. This is the method used by KinfuLS in the Point Cloud Library.

The TSDF stores a value at each voxel that represents the signed distance from the voxel to the surface along the line of sight from the camera. The distance of the voxel is calculated by summing the estimated distance for each frame and multiplying it by the weight for that frame.

This value is then divided by the total weight for that voxel. The weight is calculated as the sum of the weights for each frame. These equations can be seen below:

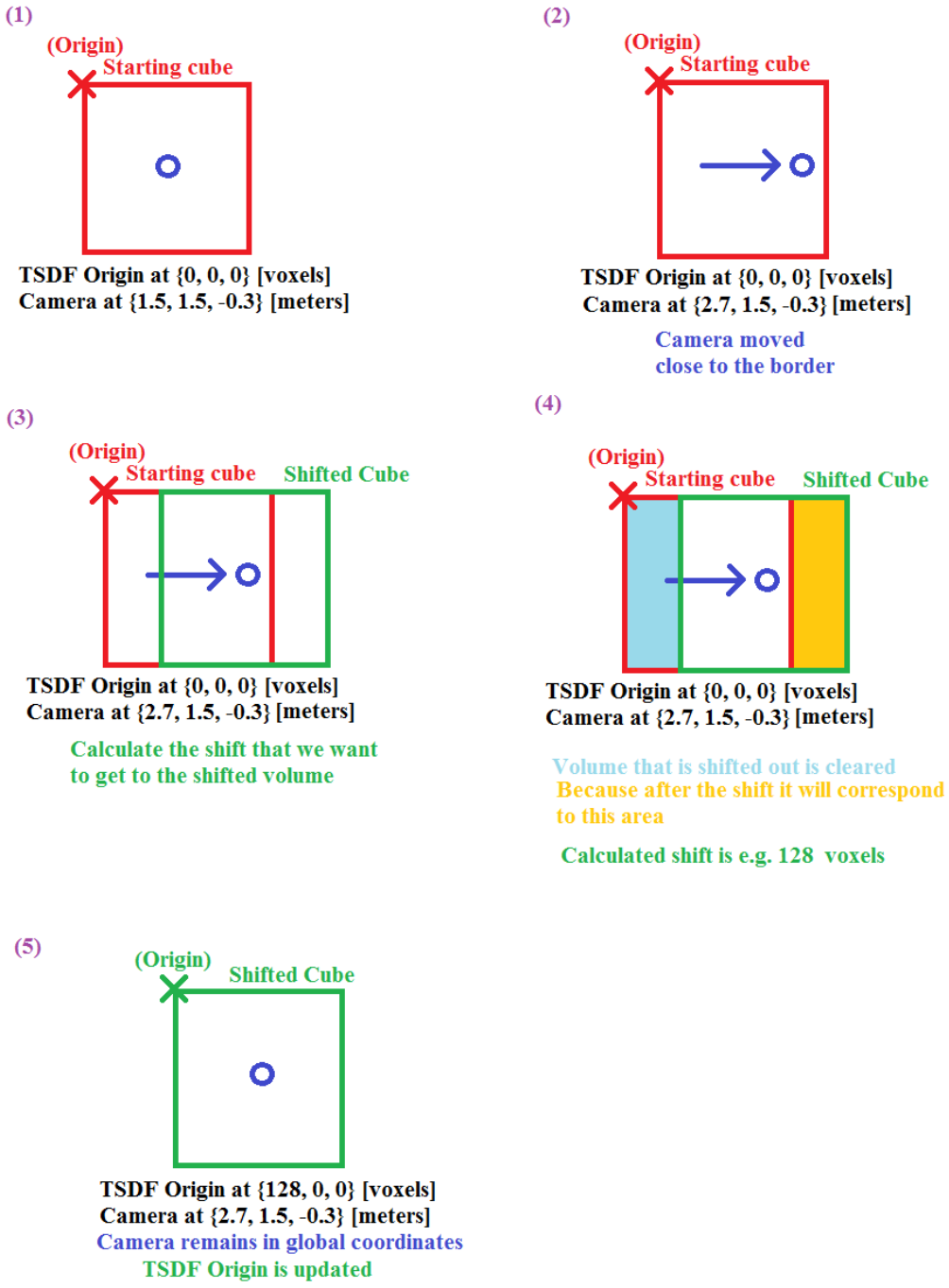

Figure 15: Method for shifting the TSDF in the PCL implementation [60].

$$D(x,y,z) = \frac{\sum w_i(x,y,z) d_i(x,y,z)}{\sum w_i(x,y,z)}$$

$$W(x,y,z) = \sum w_i(x,y,z)$$

The distance of each voxel is signed to represent weather it is inside or outside of the surface. If the number is in the range of $0 < D(x,y,z) \leq 1$ then it is considered outside the surface. While the range from $-1 \leq D(x,y,z) < 0$ is considered inside the surface. This leaves the surface represented by the function $D(x,y,z) = 0$. The point cloud can be extracted by iterating the TSDF finding the transitions from negative to positive numbers. Note that the value of $D(x,y,z)$ is not in meters but is a relative measurement. An example of this can be seen in Figure 16 below.

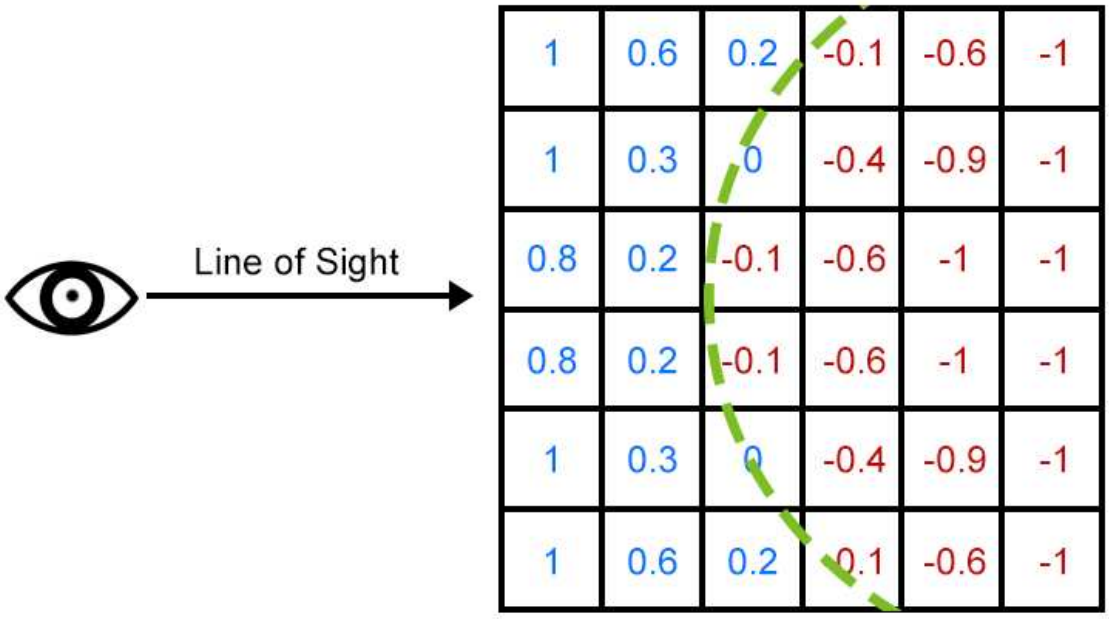

Figure 16: An example of a TSDF volumetric surface representation

For this project a modified version of the PCL implementation of KinectFusion Large Scale (KinfuLS) was used. The version used has been modified to work with the Robot

Operating System (ROS) that will be discussed in Section 3.4. Another modification made was using a 256 x 256 x 256 TSDF size as opposed to the default 512 x 512 x 512. This eliminates 87.5% of the data which allows for faster transitions when shifting the TSDF on our system. With higher end hardware it should be possible to use the 512 setting. The 512 settings has a total of 134 million voxels while the 256 setting only has 16 million voxels. The drawback from using the smaller TSDF is you lose some detail in the models. With the 512 setting each voxel represents a 5.9 mm or 0.23 inch cube. The 256 setting on the other hand has each voxel representing a 1.172 cm or 0.46 inch cube. While you do lose some accuracy, the details are not as necessary when scanning large environments. The TSDF size could be configured to use any multiple of 32, so it could be configured to use 288 or 320 for instance.

The main modification added to the PCL implementation of KinfuLS was to triangulate the point cloud into a mesh and optimize it. This enables the output file to be smaller in size and also allows it to be rendered in real-time. Using modern mesh optimization techniques allows us to significantly decrease the amount of points without losing much quality to the model.

3.2 Triangulation

Triangulation is the process of connecting unordered vertices to create a series of triangles. Triangulation has applications in many fields and problems. In this instance, the series of triangles is used to create a mesh where the triangles represent the surface of an object. The idea of using triangulated vertices to represent 3D objects is very common and is the backbone of modern rendering. Rendering is the process of creating a 2D image from the three dimensional data. Figure 17 shows an example of a triangle mesh that represents a dolphin. The same concept is applied to all 3D models.

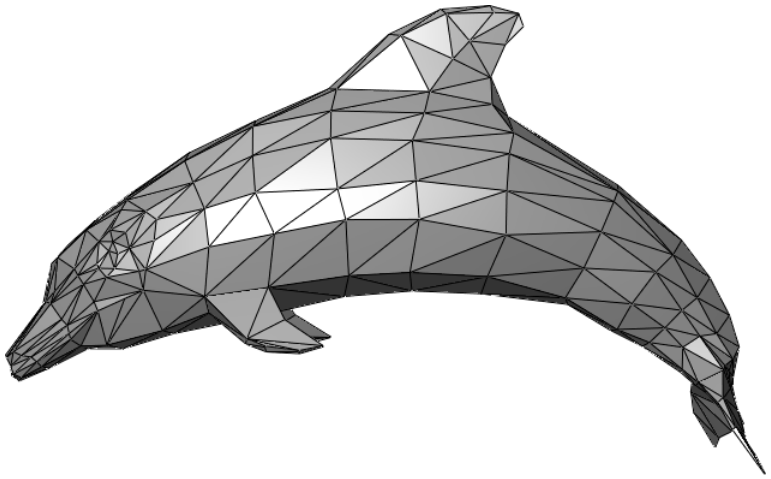

Figure 17: A triangle mesh representing a dolphin [61]

In this thesis project the marching cubes method was used to triangulate the point clouds as they were exported from the TSDF. The necessity to have a series of smaller meshes, as opposed to one large mesh, is described later in Section 3.5. As mentioned in Section 1.2, the marching cubes method is ideal for meshes with consistent density of points. This works well for our implementation because the TSDF which outputs the point cloud is composed of an even density of voxels. While not every voxel outputs a point to the point cloud, adjacent points are all evenly spaced. The second benefit to the marching cubes algorithm is that it can be run in parallel which makes it ideal for utilizing the power of our CUDA GPU. For this project the marching cubes implementation provided within the Point Cloud Library was used. This implementation is already optimized for running on the GPU using CUDA. The GPU accelerated version is only available on version 1.7 and higher.

The marching cubes algorithm works by creating a cube where each vertex is given a single bit. Each cube's results can be stored in an 8 bit value, using the one bit for each of the 8 vertices in the cube ( ). The vertex is given a value of 1 if it is found to be inside the surface and a 0 if it is outside of the surface. You can see how this algorithm has similarities to

the way the TSDF stores a volumetric representation of a surface. This provides us 256 different possibilities for each cube. By utilizing two forms of symmetry, the number of different possibilities can be reduced to 15 possibilities. The first form is rotational symmetry; the second is using complimentary cases. Cases with over 4 vertices can invert the value of each bit and use the complimentary configurations result and invert the normal of the surface. A lookup table is created for each of these 15 configurations shown in Figure 20. The output triangles can generate from 0 to 12 vertices (e_0 ... e_{11}) to represent the surface as seen in Figure 18. Each vertex e_i is located along an edge of the cube connecting v_j and v_k. The exact location of e_i along the edge is linearly interpolated based on the distance from v_j and v_k to the surface. After the cube is done being completed, the cube marches to the next location and performs the same operation. After all areas have been operated on, a resulting mesh is completed [28].

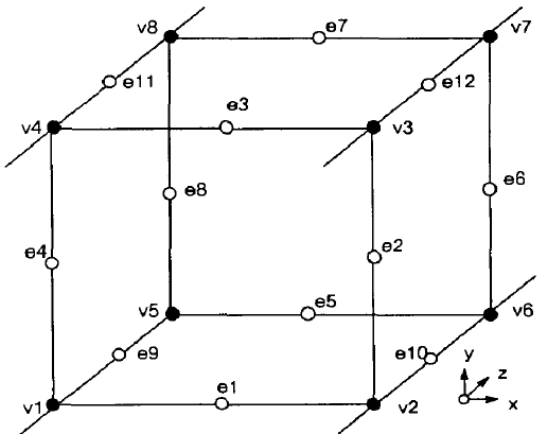

Figure 18: The vertices of a marching cube [28]

This algorithm is designed well for parallel computing. Each cube can be calculated independently of the others. This allows for many cores to be working on their own cubes simultaneously. By using the CUDA implementation, the triangulation of the point cloud can be calculated in a short amount of time.

The point cloud that is exported from the TSDF volume saves four values for each point. It saves the x, y, and z coordinate and intensity. The x, y, and z coordinates map each point into the 3D coordinate space. Each of these points should represent exactly one voxel from the TSDF volume. The intensity value is the result of the distance function $D(x, y, z)$ in the TSDF calculations. The TSDF extracts a point for every non-one value in the TSDF. The points' values can be directly applied to each cube for the marching cubes implementation. This can be done by finding the point from the point cloud that would apply to each vertex of the cube. If that point's intensity value is negative, it receives a value of 1. If the point's intensity value is positive then it receives a value of 0. If there is not a vertex in the point cloud that correlates to one of the cubes vertices, than no mesh is generated for that cube. An example of triangulation can be seen in Figure 19.

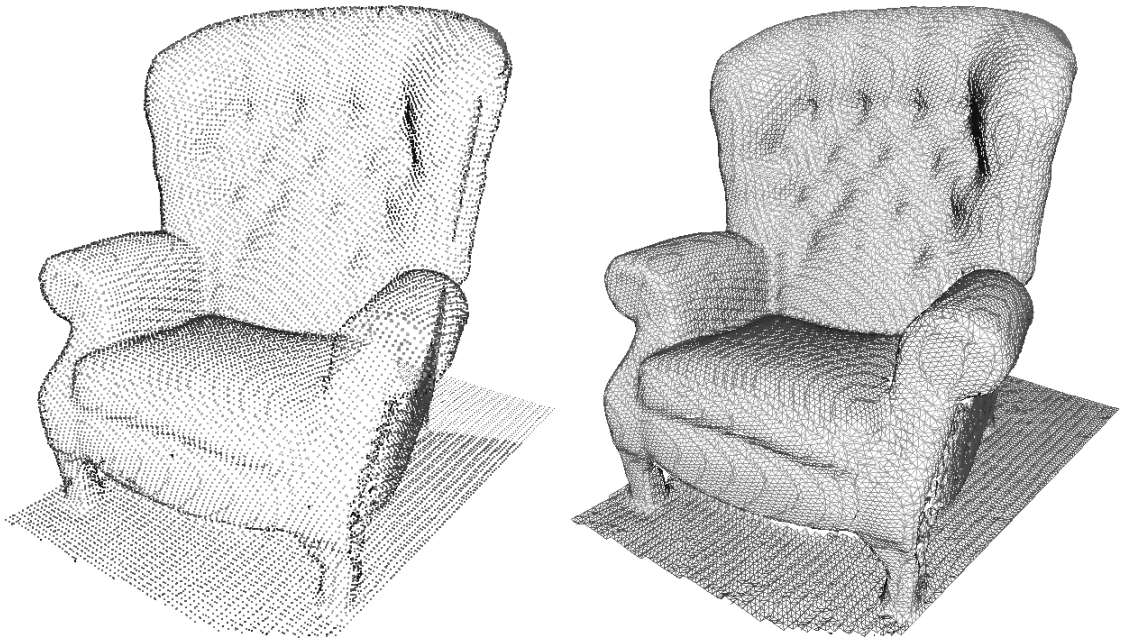

Figure 19: An example of triangulation with marching cubes. Left: A point cloud of a chair. Right: The chair triangulated with marching cubes.

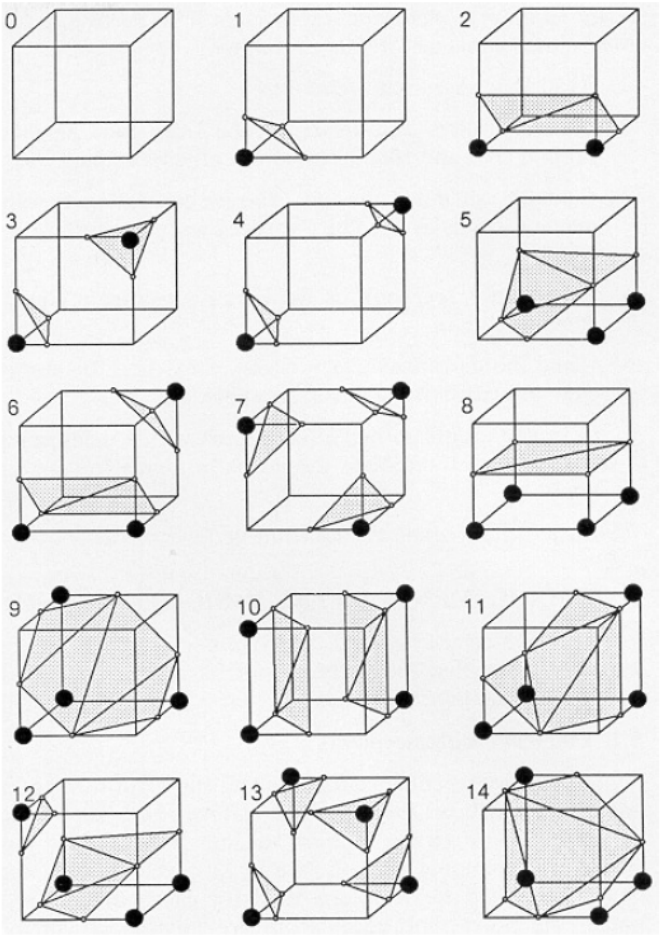

Figure 20: The 15 possible marching cubes results [28].

3.3 Mesh Optimization

Mesh optimization is the technique of simplifying a mesh to use less vertices and triangles while maintaining while still accurately representing its surface. This is a very important process in this thesis because point clouds do not efficiently represent a space, especially after triangulation. If you imagine a large planar surface such as a wall, ceiling, floor, or tabletop, a point cloud will represent this surface using thousands or millions of points. This is due to the fact that a point cloud doesn't have a surface representation; each point represents an individual point along a

surface. To properly represent the large surface, point clouds must be densely packed along its surface. However, a triangulated mesh does have a surface representation. This allows any perfectly rectangular surface to represented, ideally, with only 4 vertices connected by 5 edges. This is far more efficient than the thousands or millions of points. In practice we cannot optimize it this perfectly, but we can remove many unnecessary points without degrading the quality of the mesh. This process does not only apply to planar surfaces. Most surfaces that are created from the point cloud data contain a very dense mesh that contains a higher level of detail than is necessary. While simplifying the mesh does not perfectly represent the original mesh, the difference is minimal, depending on the level of simplification.

Figure 21: A chair at 3 levels of optimization. From left to right, each contains 42k, 10.6k and 7.4k triangles. The Top shows a smoothed mesh. The bottom shows a wireframe rendering.

An example of a chair that was scanned during the development of this thesis being optimized can be seen in Figure 21. The original mesh contained 128,328 vertices and 42,776 triangles. It was imported into MeshLab [62] and optimized using quadric edge collapse decimation. MeshLab is open source software for processing and editing of 3D meshes. The chair was optimized at different levels using the settings shown in Figure 22. The first level optimized the mesh to 25% of the original mesh resulting in 7,467 vertices and 10,687 triangles. A second level of optimization was applied resulting in 5,863 vertices and 7,479 triangles. You can see that the first level of optimization has a very similar appearance to the original mesh with only a small fraction of the original data. This is our goal for the mesh optimization process.

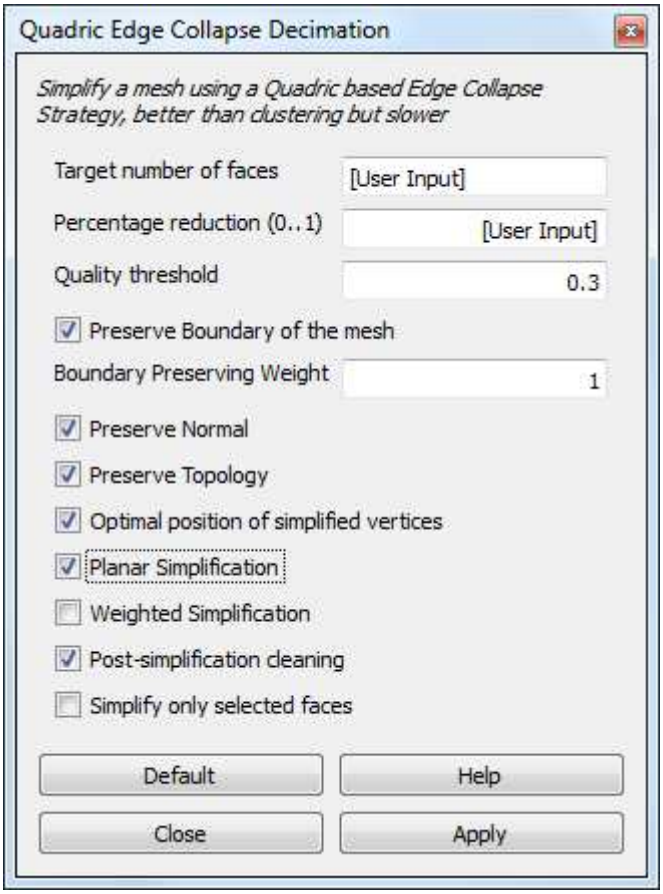

Figure 22: The setttings used for mesh optimization in MeshLab.

For this thesis,the quadric edge decimation algorithm in the MeshLab software was used. This allowed feedback from the user. As mentioned earlier, the scene is exported into several separate meshes. Depending on the content in each mesh, it can be optimized to a different level. By allowing the user to control the level of optimization we can achieve maximum compression without compromising the validity of the mesh. This solution is ideal because setting a constant level of compression can be very wasteful. Large flat areas without significant detail can be compressed to a much higher level than a mesh with many details such as a desk with many objects on it.

Quadric mesh optimization is a novel method for optimizing a mesh without compromising the overall structure of the mesh. They key idea is that each vertex, $= [v_x \ v_y \ v_z \ 1]^{\mathrm{T}}$, is associated with a 4x4 symmetric matrix, $\boldsymbol{Q}$. The error for each vertex is defined by the function $\Delta(\boldsymbol{v}) = \boldsymbol{v}^T \boldsymbol{Q} \boldsymbol{v}$. The $\boldsymbol{Q}$ matrix is a heuristic used to measure the geometric error at each vertex. Its definition and derivation can be found in [31]. The algorithm then works as follows. First, the $\boldsymbol{Q}$ matrix is calculated for each vertex. Second, each valid pair of vertices is selected. A pair of vertices is considered valid if they are connected by an edge, or within a threshold distance. Other parameters can be used, such as if the vertex is on the edge of a mesh, you can not include it for consideration as a valid pair. Next, you compute the optimal position for the new vertex created by contracting a pair. This vertex is called $\overline{\boldsymbol{v}}$. This can be computed by minimizing the function $\Delta(\overline{\boldsymbol{v}})$. This can be found by solving $\partial\Delta/\partial x = \partial\Delta/\partial y = \partial\Delta/\partial z = 0$. It may be rewritten as:

$$\begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{12} & q_{22} & q_{23} & q_{24} \\ q_{13} & q_{23} & q_{33} & q_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} \overline{\boldsymbol{v}} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

The function $\Delta(\overline{\boldsymbol{v}})$ is now considered the cost of the contraction of the vertex pair. These costs can now be sorted from lowest to highest. The contraction with the lowest cost will be performed.

After each contraction, the costs of each affected vertex and valid vertex pair can be updated. This process is repeated until you meet the desired number of triangles.

3.4 Robot Operating System

The Robot Operating System (ROS) is an open source software frame work developed for robotics [63]. It is not actually an operating system, but provides operating system like functionality such as low level hardware abstraction. ROS is supported on Ubuntu, but an experimental version can be used on many other operating systems. ROS is built off a system of packages. ROS itself comes with several packages that provide many common functions. Other users can contribute their own packages that they develop. This has created a huge repository of useful functions that can be used by anyone. Figure 23 shows how rapidly the number of packages within ROS is growing. ROS uses a publish-subscribe architecture that allows communication between packages. Each package that is run is considered a *node*. Each *node* can subscribe to data that may be published by other *nodes*. The *node* may also publish its own information that it creates. Published information is called a *topic*. Each *topic* has a type that could be anything from an integer to an image or point cloud. When information is published over a *topic* it is called a *message*. Another important feature built into ROS is the networking capabilities. In many instances, the multiple nodes will exist on the same computer. However, ROS also supports running the nodes across multiple computers. All computers will connect through the “ROS master” which provides lookup information similar to a DNS server.

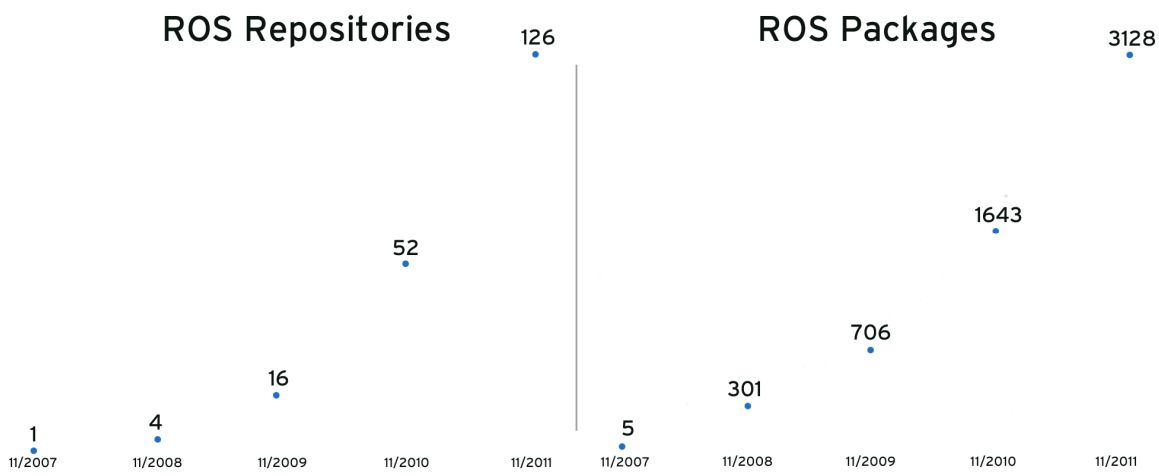

Figure 23: Growth in ROS repositories and packages through 2011 [64]

ROS was used in this thesis project for several purposes. It was used for the networking capabilities, image compression, and for OpenNI support. Several existing ROS packages were utilized. The KinectFusion and triangulation code was also ported to a ROS node. Each of the packages used will be briefly covered, followed by the overall ROS setup for this thesis.

The first package utilized in this thesis is the *openni_camera* package [65]. This package provides the OpenNI drivers needed to use the Kinect or Xtion sensors on the computer. This package handles the interaction with the RGB-D sensors and publishes several topics. It publishes images and camera information for RGB, depth, depth registered, and infrared type images. The camera information is published using the sensor_msgs/CameraInfo type [66]. It provides many details that are needed such as a distortion matrix, and the intrinsic camera matrix. The depth images are published using unsigned 16 bit images where each pixel represents a depth in millimeters. The *openni_camera* publishes all images using the *image_transport* package.

The *image_transport* package is the standard method for publishing and subscribing to images within ROS [67]. It provides an abstracted layer of processing that allows images to automatically be published in several ways. Without adding any extra code, images can be

published with raw data, image compression, or using Theora video codec. There are two types of image compression, both of which are used in this thesis. The first is simply called *compressed* in ROS. It uses JPEG compression on the images. The second type is called *compressedDepth* and uses PNG compression. This method is designed for compressing the 16 bit depth images. PNG compression is lossless, unlike JPEG compression. The *theora* type is based off the Theora compression developed by Xiph.Org [68]. It is a free and open lossy video compression format. The Theora video compression currently has a bug and could not be used in this project.

The *image_view* package is used for viewing the images that are published in this thesis [69]. Primarily, it is used for displaying the status image that is published back to the user on the client computer. An example of *image_view* can be seen in Figure 24.

The *roscore* package is a collection and nodes and programs that must be running for ros nodes to communicate [70]. The computer that is running the *roscore* package is considered the "ROS master" that was mentioned previously. The *roscore* package is in charge of handling the communication of topics between nodes across multiple machines. Two shell variables must be set before running any nodes to ensure the network is set up correctly. The first is ROS_IP. It should be set to the IP address of the node is going to be run on. The second is ROS_MASTER_URI. This variable needs to be set to the IP address of the "ROS master". It must include "http://" followed by the IP address and then the port number proceeded by a colon.

The final package used was the one developed for this thesis named *kinfu*. This package combined the code written for 3D mapping and triangulation and combined it with ROS. It was able to subscribe to the *openni_camera*'s topics to allow the user to create 3D maps from a remote computer. It is also able to publish the status image back to the user. These images can be seen in Figure 24. The depth image is compressed using the *compressedDepth* mode. The status image is compressed using the *compressed* mode. The *kinfu* node also publishes the relative

transformation. When the mapping is being performed, the software tracks the cameras location as it moves with 6DOF. This information can be published over the ROS network. It is very common in robotics applications to have several of these topics being published. Typically, they are published to the topic name */tf*. This was also done in the *kinfu* package created in this thesis. ROS has packages that can manage several coordinate frames and their relationships such as the *tf* package [71]. This can be used for localization and sensor fusion. This information is not used for anything in this thesis, but has made it available for future uses of this project.

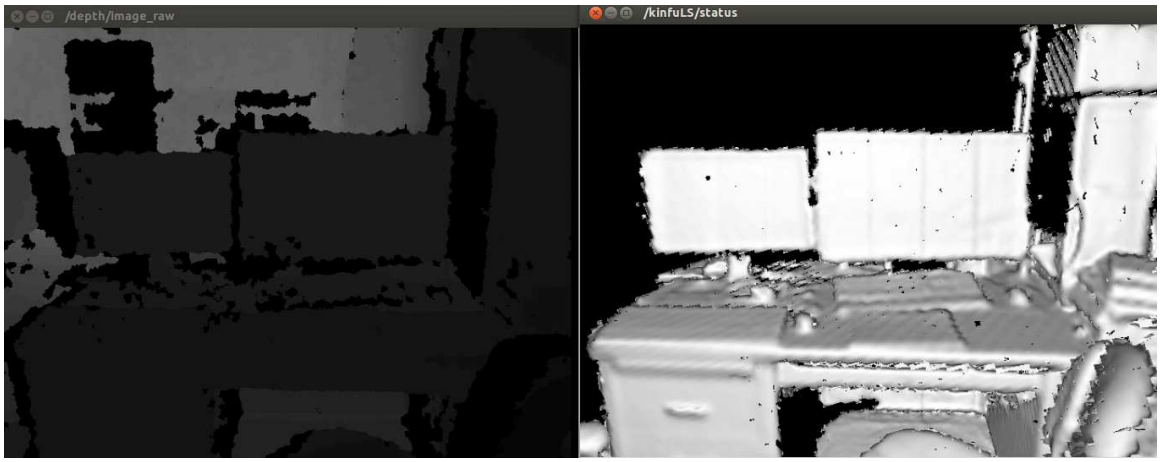

Figure 24: The two images published with ROS in this thesis. Left: The depth image published by the *openni_camera* node. Right: The status image published by the *kinfu* node.

Figure 25 shows the ROS graph created by running this thesis. It is created by running the *rxgraph* tool provided with ROS [72]. It is configured to only show topics that are connected. There are several other topics that are being published, but do not have any subscribers. These include the RGB and infrared images being published by *openni_camera*. The */openni_driver* node is created by running the *openni_camera* package. It is publishing both the depth image and camera information to the */kinfuLS* node. The */kinfuLS* node is also publishing the status image over the topic */kinfuLS/status*. You can see an instance of image view subscribed to this topic. This is used to display the status image to the user. You can also see */kinfuLS* publishing the */tf* topic as mentioned previously. In this instance it is being subscribed to by *rxplot*, a ROS package

used for plotting data from published topics *[73]*. Finally, the */rosout* topics and node you see are created by *roscore* and performs logging. In this setup, the */kinfuLS* node and *roscore* would be running on the server computer while the */openni_driver* and *image_view* nodes would be running on the portable client computer.

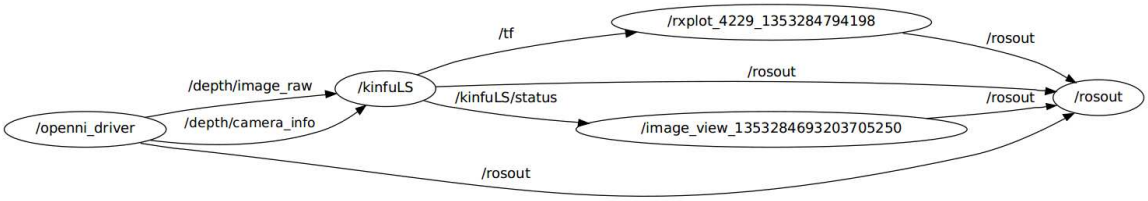

Figure 25: The ROS graph when running this thesis.

3.5 Rendering

At this point we have created a 3D map using cloud computing. The 3D map is created by exporting a point cloud which is then triangulated into several meshes. These meshes are then all optimized using quadric mesh optimization. We now have several meshes which represent a 3D space and would like to be able to present them in a visually appealing manner. To accomplish this task the Unreal Engine has been used. As mentioned in the previous work section, the Unreal Engine is a state-of-the-art rendering engine that can be used for everything from video games to television shows. It has many features including photorealistic rendering, real-time physics simulations, and the capability of supporting very detailed scenes. It has numerous other features; however they are not relevant to this thesis.

The Unreal Engine provides a development kit named the Unreal Development Kit (UDK). The UDK provides a free edition of the newest Unreal Engine 3 to the public. It is free

for non-commercial use. The UDK and the tools it provide have won several awards for their technologies and availability. They also release regular updates to give users access to the newest technology and features, the most recent being in November 2012 [74]. The UDK provides developers with access to tools needed to develop entire games based on Unreal Engine 3. It also provides them with a level creator that can be used to create custom levels and environments. This is the feature used in this thesis for importing and rendering our meshes and can be seen in Figure 26.

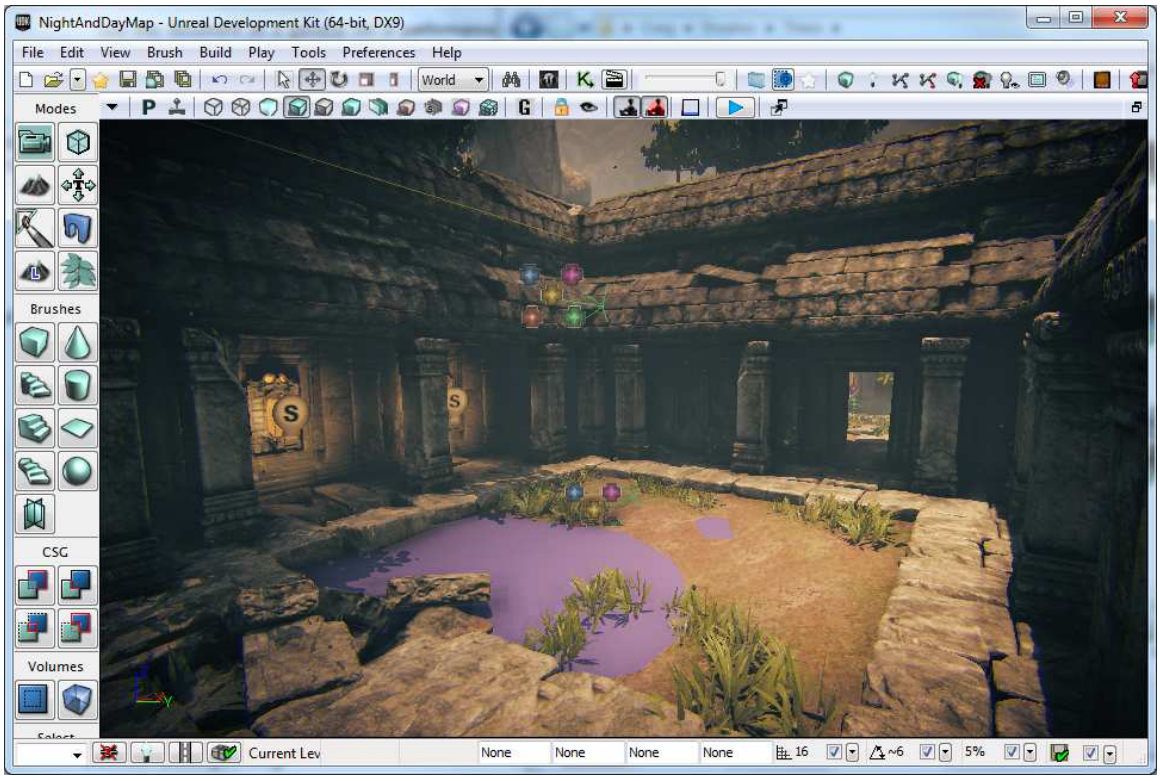

Figure 26: A screenshot of the Unreal Developement Kit Level Editor running a provided demo level.

There are several aspects of the Unreal Engine that must be considered for this project. The first aspect is the world unit size. The Unreal Engine has its own unit size, dubbed the Unreal Unit. An Unreal Unit is equal to two centimeters [75]. One unit in our *kinfu* ROS package is equal to one meter. This discrepancy was handled by scaling the size of the point cloud input to the

marching cubes algorithm. This was done using the PCL's transforms functions. The point cloud for each mesh was altered from the original 3 meters to 150 Unreal Units. This ensures the output of the marching cubes algorithm is consistent with the sizes in the Unreal Engine.

A second consideration of the Unreal Engine is the maximum file size. Unreal Engine only supports up to 65,535 vertices per static mesh [76]. In our 3D mapping application, this eliminates the possibility of using one mesh to represent the entire map. As explained earlier in this thesis, we export a series of smaller meshes as the user moves around. Each mesh maintains its global coordinates so that when it is imported into the UDK, each mesh will be correctly aligned with the others.

To utilize the physics engine of Unreal Engine 3, we need to create a collision mesh. A collision mesh is a very low polygon mesh that is not rendered to the user. The collision mesh is only used in physics calculations. By using a low polygon mesh, the physics engine can check for collisions in a small fraction of the time compared to the full mesh. Many times physics calculations do not need the details that rendering needs, so using a collision mesh still creates a realistic visualization for the user. An example of a collision mesh is shown in Figure 27. The green, lower polygon mesh is the collision mesh. The blue, dense mesh is the graphic mesh. You can see that they are very similar in size and the user would most likely not notice the difference in the physics.

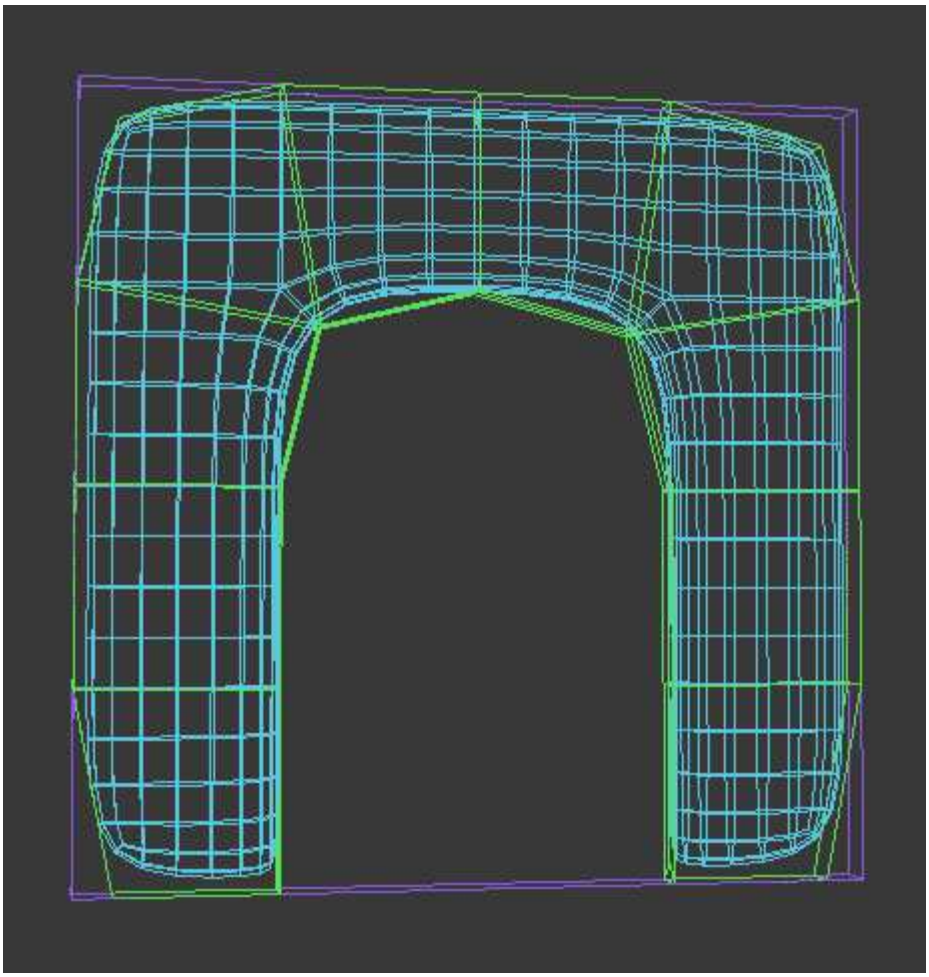

Figure 27: Collision mesh example [#]

The Unreal Engine 3 uses the FBX file type for its static meshes. FBX is a filetype designed by Autodesk [77]. It includes capabilities for 2D, 3D, audio, video, and properties. It is designed to provide interoperability between different content creation applications. The file type is used widely in several different applications because of its ability to work with multiple software applications. Unfortunately MeshLab did not adopt the FBX file type, and instead chose the Universal 3D (U3D) file format [78]. Because of this, the optimized mesh must be ran through another program to convert it into the FBX file needed for Unreal Engine 3. In this thesis Blender is used to convert the file to FBX. Blender is a free, open-source 3D content creation suite that is available for many operating systems [79]. To do this conversion the mesh is saved

out of MeshLab in the Collada format (.dae) and imported it into Blender. The mesh could then be exported in the FBX file necessary for Unreal Engine 3.

CHAPTER IV

EXPERIMENTS

For this thesis several experiments were performed on the 3D mapping and rendering system. Each experiment was designed to test different aspects of the system. In this chapter, the details of each experiment run will be discussed, as well as, what aspects were tested, and the results of each of the experiments.

4.1 Accuracy

My first several experiments were geared toward testing the mapping of the system. There were several aspects of the 3D mapping that needed to be tested. The first aspect to test was the accuracy. This experiment was designed to ensure that the measurements in the created map and mesh would be accurate to the measurements in reality. To test this, a 3D map of a room with many known measurements was created. The exported mesh file was then imported into MeshLab. MeshLab has a measuring tool that allows you to find the distance between two points along a mesh. A screenshot of one of the measurements in MeshLab is provided in Figure 28. To perform this experiment a large portion of our lab was mapped, ensuring that the TSDF volume was shifted several times and that there were multiple meshes exported. This allowed us to

Measurement	Actual (m)	Experimental (m)	Difference (m)	Error	Multi-TSDF
Printer	0.419	0.408	-0.011	2.63%	No
Cone	0.457	0.454	-0.003	0.66%	No
Box	0.371	0.342	-0.029	7.82%	No
Drawer	0.406	0.402	-0.004	0.99%	No
Bench Width	1.714	1.654	-0.06	3.50%	Yes
Bench to Desk	2.616	2.51	-0.106	4.05%	Yes
Box to Robot	1.943	1.859	-0.084	4.32%	Yes
Room Width	6.299	6.022	-0.277	4.40%	Yes

Table 1: Experimental results of measuring real objects and the virtually 3D mapped objects.

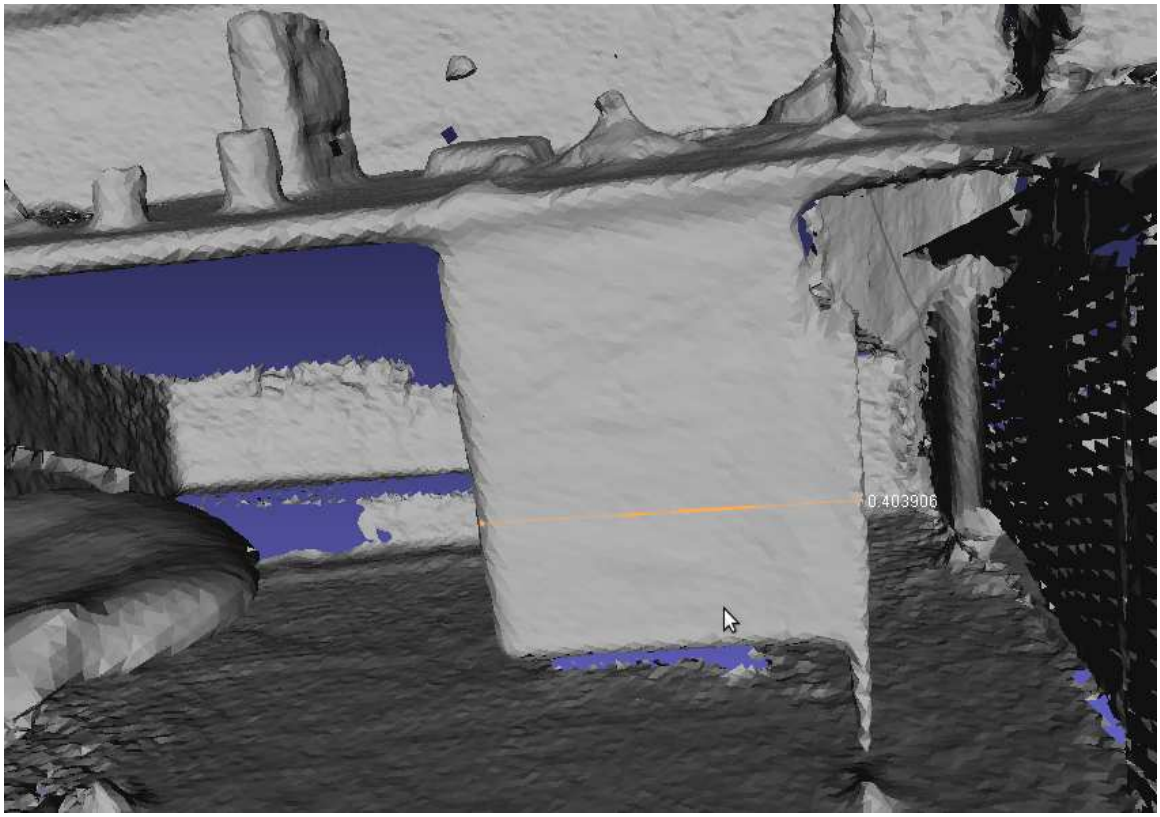

Figure 28: Measuring the drawer width of a 3D Map in MeshLab

measure the accuracy of not only the distances within a TSDF volume, but the accuracy across larger areas. The results of this experiment are shown in Table 1.

The results of the experiment show consistent results for most measurements. The measurements that took place within the same TSDF volume had very accurate results with the exception of the box measurement. This is likely explained by the position of the box. The box was in the corner of the room and did was always near the maximum distance from the depth sensor. Another cause for error is the lower resolution TSDF volume. By decreasing its size from 512 to 256 we lose a small amount of accuracy. Other measurements within the same TSDF provided very accurate results. The drawer and cone measurements both resulted in less than 1% error. The multi-TSDF measurements provided very consistent results. In all four cases the measurements were between 3.5% and 4.4% smaller than the actual measurements. This discrepancy could easily be taken care of by adjusting the transform when converting to Unreal Units. Overall, every experimental measurement was less than the actual measurement. This encourages the transformation to provide more accurate results. By adding a 4% increase to the transformation we can achieve the numbers provided in Table 2.

Measurement	**Actual (m)**	**Experimental (m)**	**Difference (m)**	**Error**	**Multi-TSDF**
Printer	0.419	0.42432	0.00532	1.27%	No
Cone	0.457	0.47216	0.01516	3.32%	No
Box	0.371	0.35568	-0.01532	4.13%	No
Drawer	0.406	0.41808	0.01208	2.98%	No
Bench Width	1.714	1.72016	0.00616	0.36%	Yes
Bench to Desk	2.616	2.6104	-0.0056	0.21%	Yes
Box to Robot	1.943	1.93336	-0.00964	0.50%	Yes
Room Width	6.299	6.26288	-0.03612	0.57%	Yes

Table 2: Adjusting the transformation provides more accurate and consistent results.

By adjusting the transformation, more consistent results were achieved. On the measurements within a single TSDF volume, we did not achieve the level of accuracy as before, but also did not have as large of range of error. However, on the multi-TSDF volume measurements, we have decreased our error to a range from 0.21% - 0.57%. This is a very high level of accuracy. As mentioned previously, this thesis in designed for scanning in larger areas. Therefore we are more concerned with the accuracy in larger measurements than the accuracy of measuring small details. These results are well within the desired accuracy for this thesis.

4.2 Processing Speed

In the previous work section we discussed several similar works that utilized RGB features to assist in finding their transformations. The limitations of these methods was that it required extra processing time and would decrease the overall frame rate to only a few frames per second (FPS) and in some cases below one FPS. It is important to provide the user with a frame rate that allows for fluid, natural motion and that does not appear to have a high latency. Ideally the software should be able to run at a minimum of 10 FPS.

In this experiment, the software was run for several minutes scanning in a large environment and recorded the frames per second. By scanning in a large area it could be determined if the overall memory size would cause the frame rate to drop. The lab room was mapped very carefully for over 10 minutes. It created a world containing over 3.6 million points. The frames per second and elapsed time were tracked. The number of frames per second was calculated by finding the elapsed time per every 33 frames processed. The results of this experiment can be seen in Figure 29. The frame rate stayed around 20 frames per second for most of the mapping process. There were occasional dips down to the 13 FPS range when the TSDF was shifted. This is expected and was one reason why it was necessary to use the 256 voxel size

in the TSDF volume. The overall average for this experiment was 19.94 FPS. This is an adequate frame rate for using this system. There did not appear to be a drop in the average FPS as the time went on. The frame rate is high enough that even with the drops while shifting it still met the requirements put forth for this thesis. For most applications, such as 3D mapping for rendering, it should only take a few minutes to map the entire area. If this software was going to be used in another way, such as robotics, where it would be running for an extended period of time, the decrease in FPS while shifting may need to be addressed.

The *openni_camera* node publishes images at 30 frames per second in ideal conditions. Its frame rate was tested when publishing over the wireless network. The frame rate ranged from 15.6 FPS when frame sizes were averaging over 100 kB to 29.6 FPS when frames were compressed to less than 20 kB. The frame rate was typically in the range of 25 FPS. This experiment was run on the netbook from Section 2.3.1. The limitation to the publishing was the processing power of the netbook. Running an identical test on the workstation from Section 2.4.1 results in a constant rate between 29 and 30 FPS.

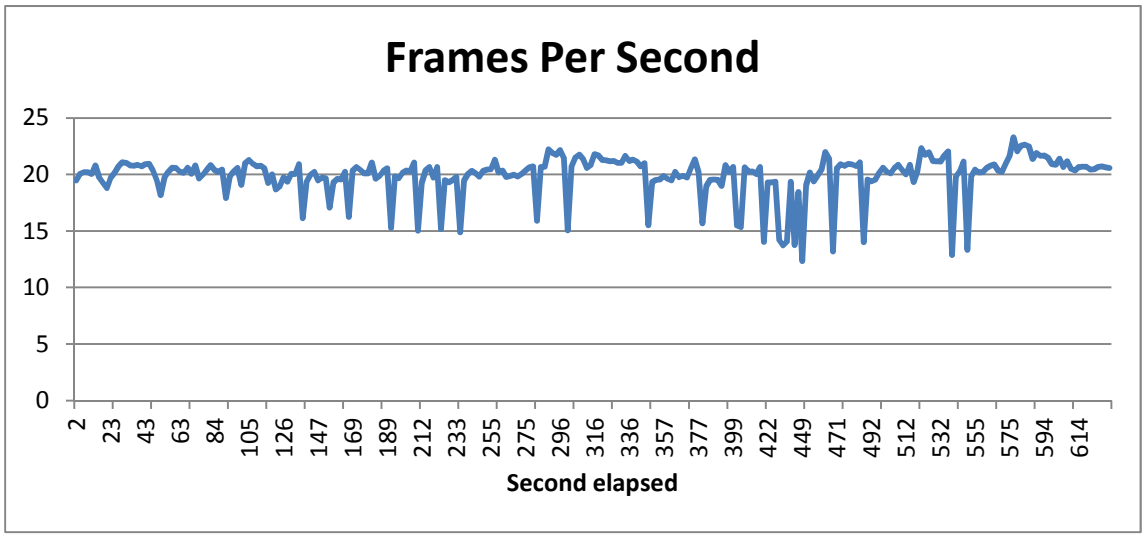

Figure 29 : Frames per second during a 3D mapping session.

4.3 Alignment Reliability

One of the key weaknesses of the Kinfu system is that it struggles to keep track of the camera's location when the depth sensor does not contain enough detail. In areas that are predominantly flat and featureless, the Kinfu algorithm will not be able to detect where the camera is moving. This is one reason why many of the previous work implementations used RGB features to assist in tracking. Currently the algorithm handles getting lost by acting as though it had shifted back to the origin. It will extract everything in the TSDF volume to a point cloud, similar to what it would do during a shift. This prevents having lost data or merging multiple scans in different coordinate systems into one TSDF. Once everything has been shifted out, it will reset the cameras current location to the origin and restart the mapping process. This can result in having misaligned meshes output. It can be recovered by manually translating the misaligned meshes to align them with the others. MeshLab has built in functionality for alignment with minimal user input. While this will allow for Kinfu to work, even when getting lost, it is not an ideal solution. It also does not allow the user to scan areas with few features as the system will constantly be considered lost.

In an effort to test the reliability of the system to properly align the cameras location within the coordinate system, an experiment was devised to test it under different conditionsThe 3D mapping system was run in three different environments, and the number of times the system was considered to be lost was recorded. The first environment is ATRC 304. It is a typical sized room with many features that should be relatively easy for the algorithm to align. The second environment is the ATRC hallway. This environment has very few features and the algorithm could struggle to track the movement. The third environment is ATRC 320. This is a large lab room that has many features, but they are spread apart which could prove challenging for the depth image to show. Each environment is mapped exactly one time. This is done because the system should not require users to perform multiple attempts to map one area. A user should be able to successfully map their environment on their first attempt.

Location	Features	Frames	Lost
ATRC 304	Rich	2013	1
ATRC 320	Dispersed	2376	3
Hallway	Few	924	5

Table 3:Results from reliability testing

This experiment behaved similar to the predictions. During the scan of ATRC 304 the software only became lost one time. It was not clear what made it get lost on that frame. ATRC 320 has many features but there are large areas between them. In the areas with lots of features it behaved very well and did not get lost. However, in the areas without features it became lost three times. The algorithm is not designed to track areas outside the TSDF. If you are centered in the 3m x 3m x 3m TSDF volume, anything beyond 1.5 meters in front of you cannot be used for tracking. This was the issue for ATRC 320. While being lost only three times is not devastating, it makes it very difficult for the user to reconstruct their 3D map when they are complete. Finally the hallway was tested. The hallway is nearly completely devoid of features that the depth sensor could pick up on. If you used these features, you could scan small areas while these features are in the frame. However, the mapping of the hallway was unsuccessful due to the large areas where there are no depth features. The depth and RGB images are shown below in Figure 30 so you can see the depth map in each of the different types of environments.

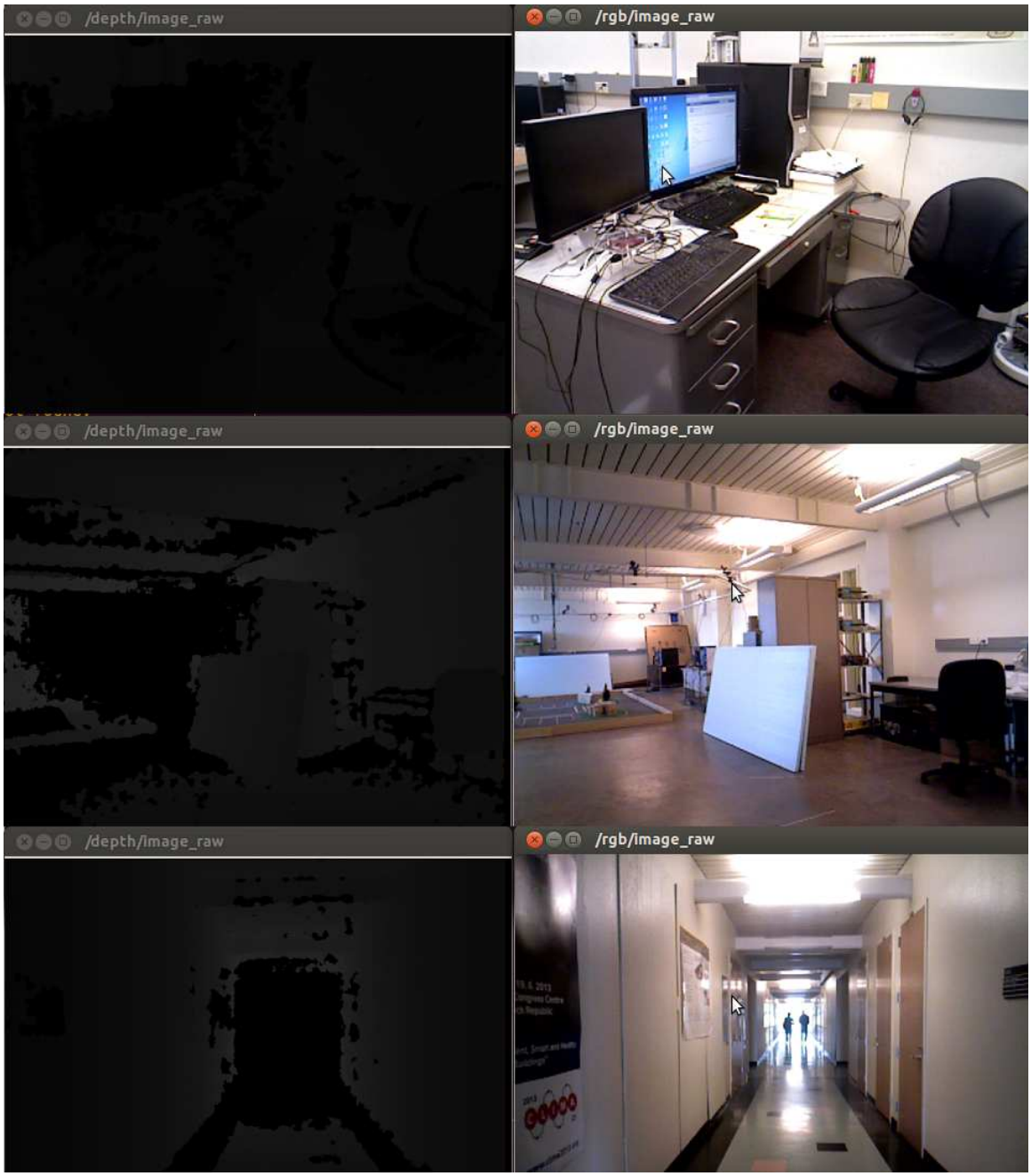

Figure 30: Depth and RGB images for each of the three environments tested. Top: ATRC 304. Middle ATRC 320. Bottom: Hallway.

This experiment clearly shows that the reliability is lacking in large areas. The results can be seen in Table 3. This has been a known issue with KinectFusion since its release. There is discussion within the PCL developers to use the RGB information, if it is available, when the

depth image cannot provide enough features to accurately find the transformation [80]. This solution would help for mapping large environments with empty spaces. There has been research in other areas for improving upon this problem such as using motion capture [58] and IMU information [81]. Also, because this implementation does not utilize a loop closure algorithm as done in [4], drift can occur over time. An example of this problem can be clearly seen in Figure 31.

Figure 31: An example of drift occuring during a scanin a rectangular room.

4.4 Voxel Size

For the third experiment on the point cloud acquisition, the goal was to gather some information on using different voxel sizes for the TSDF volume. As mentioned previously, for this thesis a 256 x 256 x 256 TSDF representing a 3 meter x 3 meter x 3 meter cube is used. This was done to

eliminate much of the data and in turn allow for faster processing. It was done at the cost of increased detail in the 3D map. The goal of this experiment is to validate these two claims by showing the difference in data, speed, and detail in the scene. Since the amount of detail in a scene is not easily quantifiable, images of the results will be shown to allow the reader to make their own judgments.

For this experiment, KinfuLS was compiled to run using either the 256 or 512 voxel setting. ROS was not for this experiment to ensure that the results were not affected by the netbook's processing power or network speed.. Instead the Xtion Pro Live sensor was connected directly to the server computer. The surrounding area was then carefully scanned, using the same path for both settings. The outputs of the two scans were used to compare the differences. The results of this experiment can be seen in Table 4 and Figure 32 and 33.

Voxel Size	256	512	Ratio
Exported Points	245,390	1,947,012	12.60%
PCD File Size (KB)	3,839	30,427	12.62%
Shifting Time (ms)	65	231	28.14%
Average FPS	23.37	17.78	131.44%

Table 4: Results of testing dividing the TSDF volume into 256 or 512 voxels in each dimension.

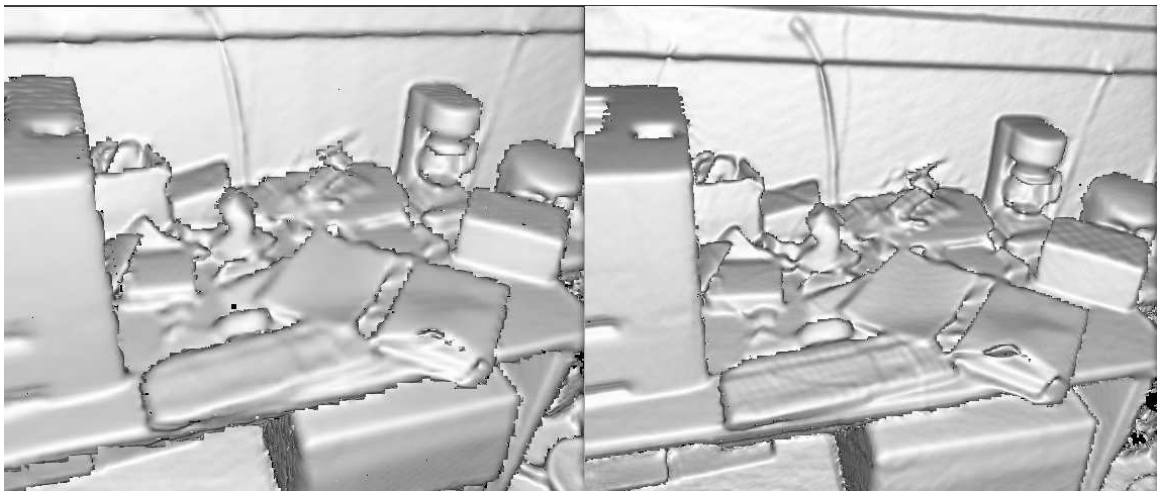

Figure 32: A comparison between dividing the TSDF volume into 256 (left) or 512 (right) voxels along each dimension.

The results of this experiment were very accurate with my predictions set forth earlier in this thesis. The voxel size setting v_s determines the total numbers of voxels by the equation$n_v = {v_s}^3$. Therefore, but cutting the voxel size in half, the number of voxels is decreased to $\left(\frac{1}{2}\right)^3 = \frac{1}{8} = 0.125$. Both the exported number of points and the resulting PCD file size were reduced to nearly this fraction, at 12.60% and 12.62% respectively. This is a substantial saving in memory and hard drive space. By reducing the voxel size we also decreased the shifting time by over 70%. This is important because if the user is moving the camera quickly during a shift, it can cause a long pause between frames. This pause can lead to the camera getting lost and ruining the 3D mapping attempt. A final interesting effect of the voxel size is the average frames per second. In these scans we were able to increase the frames per second by 31% simply by adjusting the voxel size. As covered in Section 4.2, frames per second are vital for the usability of the system.

Looking at Figure 32, you can see that the 512 voxel size setting does provide more detail than the 256 setting. However, the objects in the 256 setting are still easily recognizable. You can clearly make out the keyboard, notebook, and coffee maker. To reiterate, if this project was aimed at mapping smaller areas, a 512 voxel size setting would be recommended. However, with large

environments, the slight sacrifice in detail is worth the difference in file size, frame rate, and shifting time. Figure 33 shows the resulting point cloud from using both settings. They are both looking at the top of the printer in our lab. You can see how the overall shape is represented similarly in both images, but the 256 setting has far less overall points.

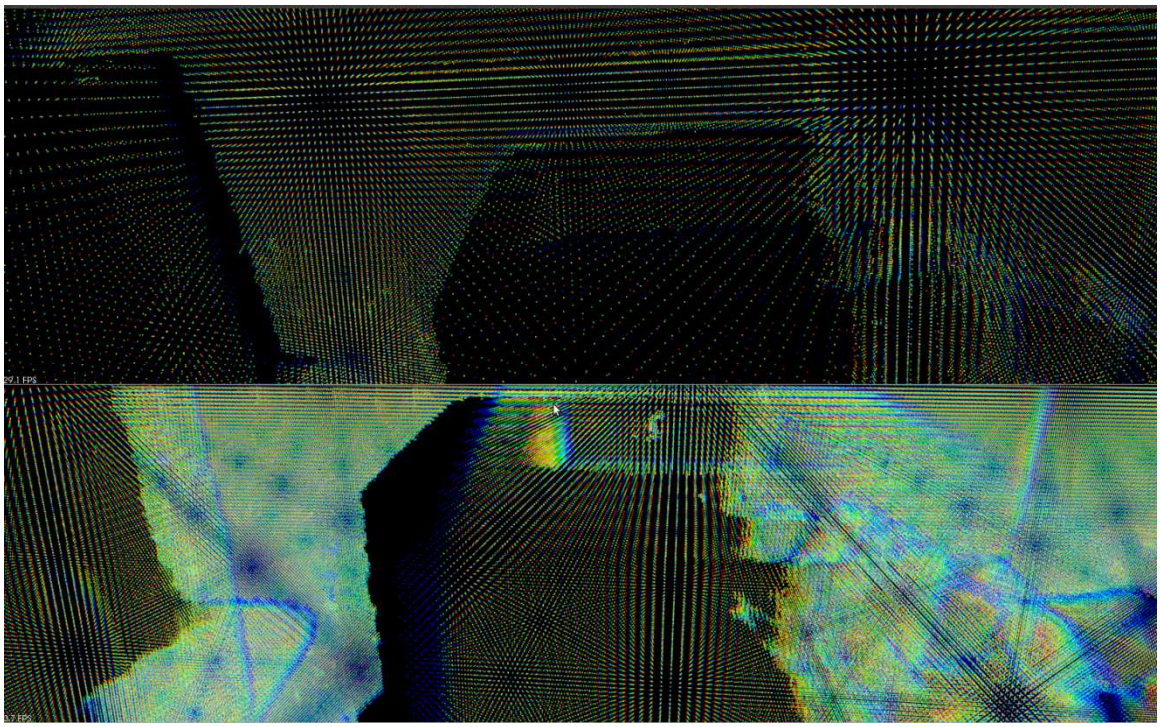

Figure 33: A comparison of the resulting point cloud files from this experiment. Top: Voxel size is 256. Bottom: Voxel size is 512.

4.5 Bandwidth Requirements

In order to utilize the cloud computing service, the user must be able to stream their depth data to the server while downloading the status images from the server. The amount of bandwidth used by these two streams is important to the quality of the scan. The user must be able to communicate quickly with the server to ensure they are able to view what they are doing. For this reason an experiment was created that would test the bandwidth needed to stream both of these

images. In order to test this, the *rostopic* package from ROS was utilized [84]. It provides a service that can measure the bandwidth each topic is using. The bandwidth was recorded during mapping on one topic at a time. It was ensured the topics would publish a variety of images from dark, light, and having a variety of shading. This is important as compression, and therefore the bandwidth, can be affected by the image's content. The results of this experiment can be seen in Table 5.

Compression Type	**/depth/image topic**			**/kinfuLS/status topic**		
	Min (kB/s)	**Average (kB/s)**	**Max (kB/s)**	**Min (kB/s)**	**Average (kB/s)**	**Max (kB/s)**
Raw	4,300	4,738	5,000	11,910	13,328	13,930
Compressed	N/A	N/A	N/A	358	751	1,030
CompressedDepth	1,720	1,932	2,320	N/A	N/A	N/A
Theroa	N/A	N/A	N/A	N/A	N/A	N/A

Table 5: Bandwidth usage for both depth and status images with multiple compression types.

The results of this experiment are slightly interesting. As we would expect the raw, or uncompressed, images have a very high bit rate. These are beyond usable for this thesis. As mentioned earlier, the *compressed* transport method only works on the status image because the depth image is 16 bit and is not supported. The *compressed* transport method achieved good compression, decreasing the bit rate from 13 MB/s to only 751 kB/s. The *compressedDepth* transport did not perform nearly as well. It decreased the bit rate from 4.7 MB/s to 1.9 MB/s. The *compressedDepth* transport uses lossless PNG compression which cannot achieve the results of the lossy JPEG compression used by the *compressed* transport. The *theora* transport could not be tested due to a bug in ROS's newest release, ROS Fuerte, which is required for running this project. It is likely that the *theora* would provide better compression for both topics if it were working. This is because *theora* uses video compression which uses previous frames to help further compress images. This is opposed to PNG or JPEG compression which compresses each

frame individually. The compressed depth and status images have been plotted to show the variance with time. This can be seen in Figure 34. The bandwidth was measured by averaging 100 frames to create a single average. This graph covers roughly 5 minutes of mapping use.

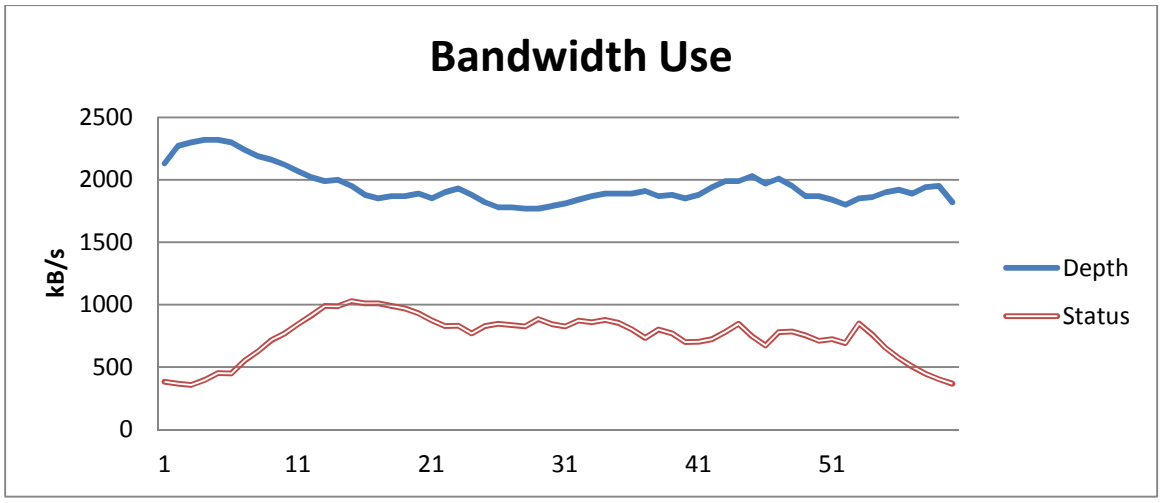

Figure 34 : Bandwidth use for depth and status images over 60 measurements.

4.6 Triangulation

After testing the acquisition of the point clouds, the triangulation of each point cloud was tested. This step is important because it must meet two criteria. The first criterion is that it must be done in a reasonable amount of time. The user should not have to wait very long to get their result from the 3D mapping process. The second criterion is that is must correctly triangulate the point clouds. This is another subjective measurement that is hard to quantify. To test the triangulation point clouds were collected from several scans and triangulated. During the scans it was attempted to create output clouds of differing sizes in order to test the marching cubes algorithm on different levels of data. The amount of time taken to triangulate each cloud was measured. This provided data on the run time for triangulation. This data can be seen in Table 6. This data has also been plotted in Figure 35The meshes were then loaded into MeshLab to inspect the quality of the created surface and took screenshots as seen below in Figure 36.

Mesh	Points	Triangulation time (ms)
1	1,065,831	214
2	949,422	186
3	137,382	80
4	241,350	92
5	75,531	72
6	18,978	66
7	428,238	116
8	162,033	89
9	25,347	71
10	41,838	72

Table 6: Elapsed time for triangulating each point cloud.

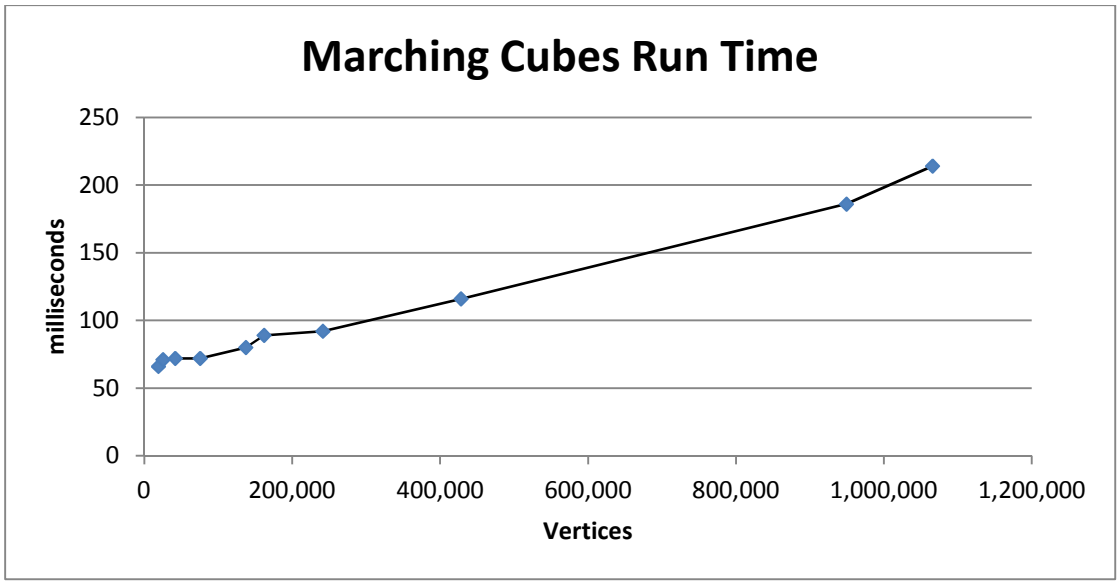

Figure 35 : Run time for the marching cubes algorithm.

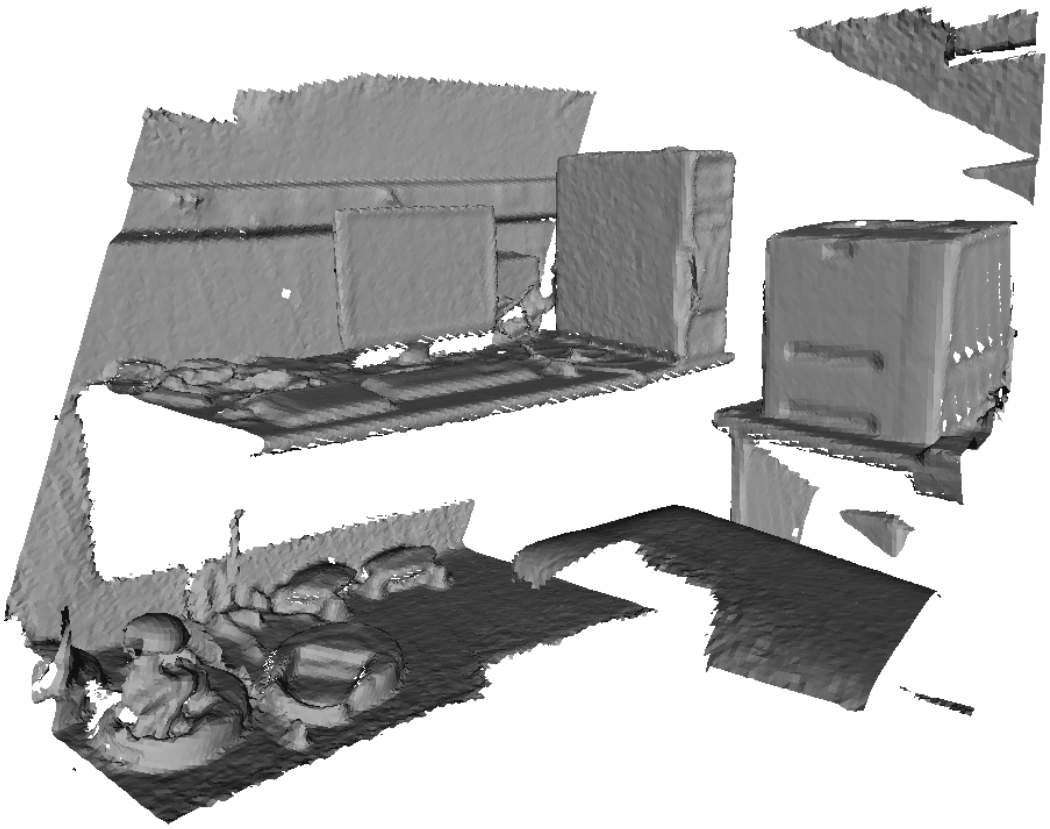

Figure 36: An example of a triangulated mesh from our 3D mapping application.

The results of this experiment are very satisfactory. We were able to triangulate over 3 million vertices in just over 1 second. This experiment shows that extremely large environments can be triangulated in a relatively small amount of time. By keeping all exported point clouds around 250,000 vertices we would be able to minimize the triangulation time to less than 100 milliseconds per mesh. Typically, the number of vertices stays below 400,000 for each exported point cloud. For the experiment a large number of vertices were intentionally exported for testing. Figure 36 shows one of the triangulated meshes. Overall it does a very good job. There are a few holes that have occurred on the right side of the printer. This tends to occur more often on reflective or glossy surfaces. The other areas that are not filled in are caused by “shadows” or areas that are hidden out of the line of site of the depth camera.

4.7 Mesh Optimization

Mesh optimization may be the most important part of this thesis. It takes these very dense meshes and creates highly optimized meshes that are much more ideal for real time rendering. However it must also preserve the shape and not degrade the quality significantly. Once again, this is a highly subjective test and it is difficult to quantify the results. An experiment was run that involves optimizing several meshes from a run of the software developed in this thesis. The level of optimization is determined by the user who must judge how much optimization to apply. The user may preview the level of optimization in order to make their decision. The input vertices, triangles, and file size, the output vertices, triangles, and file size, as well as the run time for each mesh was recorded. The input and output of some of the meshes have been displayed for comparison. The results can be seen in Table 7.

Mesh	Input			Output			Runtime (s)
	File Size (kB)	Vertices	Triangles	File Size (kB)	Vertices	Triangles	
1	10,692	351,372	117,124	439	15,717	20,000	2.742
2	6,173	208,389	69,463	231	8,391	10,404	1.587
3	7,748	234,294	78,098	339	11,968	15,581	1.691
4	1,427	51,939	17,313	63	2,538	2,585	0.408
Total:	26,040	845,994	281,998	1,072	38,614	48,570	6.43

Table 7: Results of quadric mesh optmization on 4 meshes.

The results of the mesh optimization are pretty astounding. The only result that is not impressive is the run time. This optimization takes place on the CPU so it does not benefit from the GPU acceleration. However, as the user optimizes each piece individually, they do not have to wait for the sum of all the run times, just wait for the individual mesh to be optimized. This should keep the user from ever waiting more than 3-4 seconds in the worst case. The other results

were very impressive. The file size was reduced from 26 MB to 1MB. This is a 96% decrease in size. Similarly, the vertex count went down from 846,000 to 38,600. This represents a 95% compression. Finally, the number of triangles was reduced from 282,000 to 48,600. This optimization saves 83% of the triangle count. It is important to note that there are no meshes with over 60,000 vertices. This is the limit for a static mesh to be imported into the Unreal Engine 3. An example of the input and output mesh can be seen in Figure 37. You can see that the mesh is significantly optimized while retaining most of the shapes well. You can also see that the quadric mesh optimization will optimize areas with less detail such as floors, walls, and other flat areas far more than objects that contain more curvature.

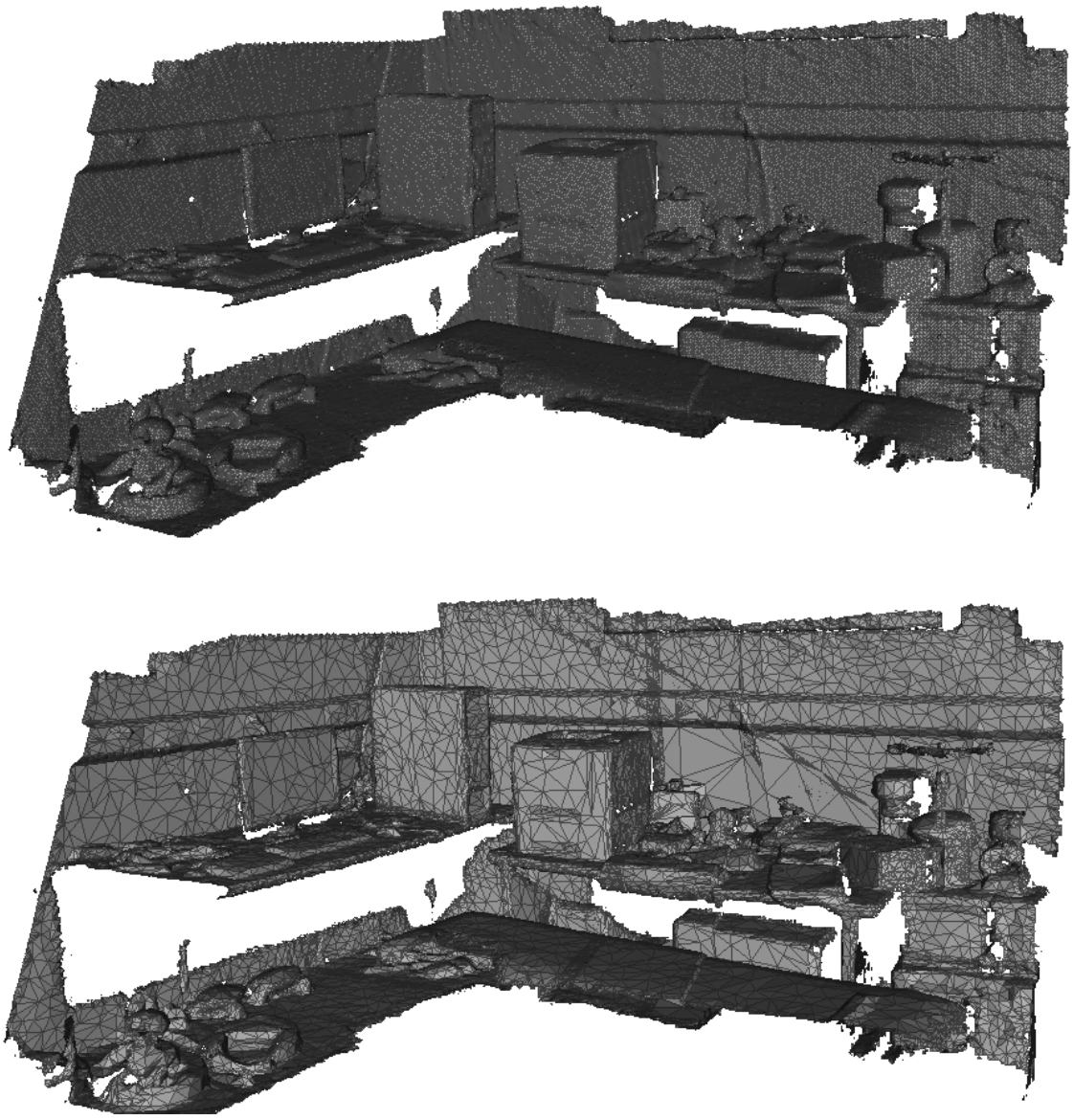

Figure 37: Before and after optimization was applied to several meshes.

4.8 Transformation Publishing

As mentioned earlier in this thesis, the *kinfu* package will publish the translation and rotation of the depth sensor in case it is needed for other applications. While it is not intrinsically

useful in this application, an experiment was still run to ensure that it is working correctly. To test this application the ROS package *rxplot* [73] was used. This package can subscribe to topics and plot their results in real time. It was able to subscribe to the x, y, and z locations published to the */tf* topic. It would then plot the movement of the device in real time. This allowed me to test the translation of the device. For the experiment a small square was made to test the movement. The *rxplot* output can be seen in Figure 38. Another ROS package was used, *rviz* [82]. This package can display the 6 DOF in real time. A screenshot of this part of the experiment can be seen in Figure 39. This was used to test both the rotation and translation of the depth sensor.

Both of these showed that the transformation was being published properly. One thing that seems odd is that the output is in the depth sensor's own coordinate frame. The *tf* ROS package mentioned earlier is designed to handle many different coordinate frames. However, it seems counter intuitive that moving vertically would not be directly along an axis. One reason for this is that the depth sensor has no way of knowing which way is up or down. This is an issue that arises when importing meshes into the UDK which will be addressed in the next section.

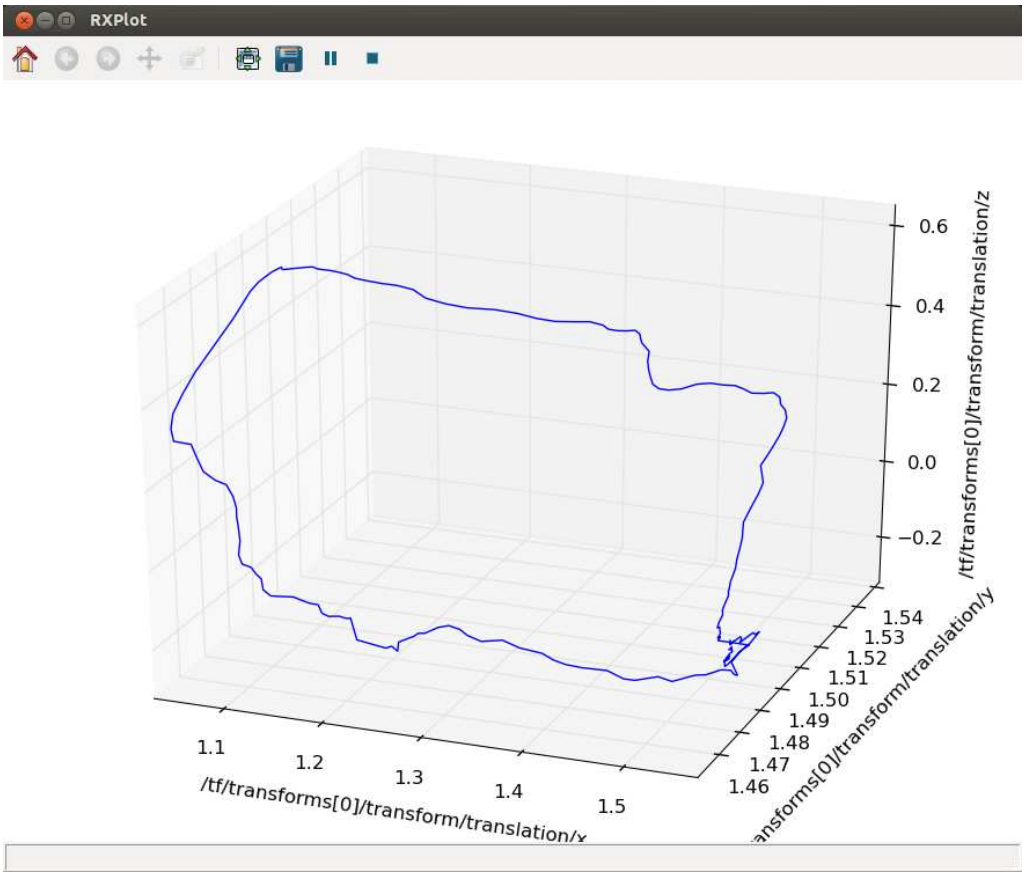

Figure 38: *Rxplot* plotting the depth sensors movement.

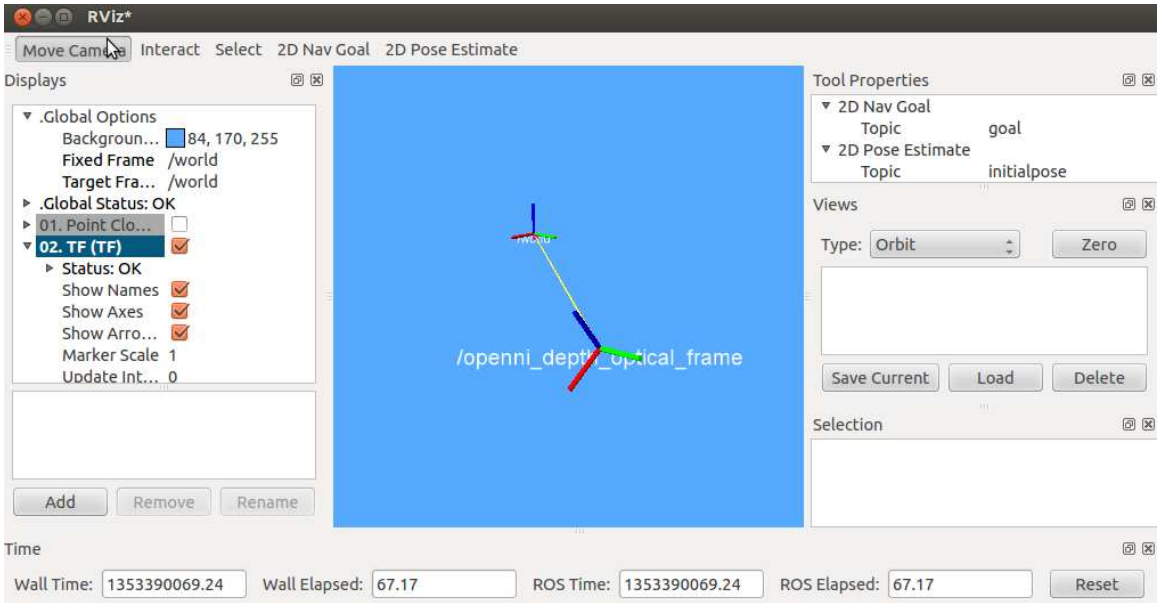

Figure 39 : *Rviz* vizualizing 6DOF output from *kinfu* in real time.

4.9 RGB-D SLAM Comparison

In this section, the work presented in this thesis is compared to the RGB-D SLAM package in ROS [83] which is based on the work in [9]. RGB-D SLAM is currently used in ROS as a 3D mapping and localization package. A brief experiment was performed to measure and compare the performance. Due to memory restrictions, only a small area was able to be mapped with RGB-D SLAM. The same area was then mapped using the method presented in this thesis.

The map created by the RGB-D SLAM program used 6.5 million points to represent the map. It was able to map at around one frame per second. The map created using this thesis used 1.3 million points and 450,000 triangles before optimization. After optimization, the map was represented using 70,000 vertices and 114,000 triangles. A visual comparison of their outputs may be seen in Figure 40.

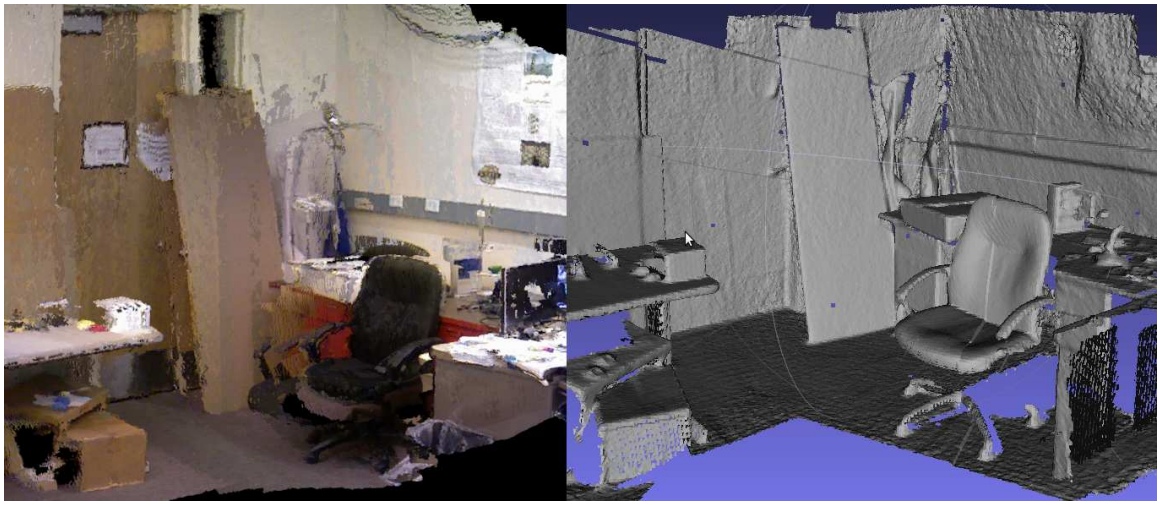

Figure 40: A comparison between RGB-D SLAM and this thesis. Left: RGB-D SLAM Output. Right: The output from this thesis

.

4.10 Overall Experiment

The end goal of this thesis is to create realistic renderings and visual displays of our optimized mesh from our 3D maps. The Unreal Engine 3 was chosen as the tool of choice for this rendering process. The details of why it was chosen and what has been done to allow the mesh to be imported have been discussed in Section 3.5. After these previous experiments, we are now ready to map a 3D environment and import it into the engine. For this process, ATRC 304 was chosen as the area to be mapped. This room was discussed in Section 4.3. This experiment was run from the beginning and did not use any outputs from the previous experiments.

To run this experiment, first the ATRC 304 lab was mapped. This produced a series of meshes which needed to be optimized. These meshes were then imported into MeshLab and optimized with the quadric mesh optimization function. They were then exported as Collada DAE files and imported to Blender. Blender was used to apply a smoothing group. This is necessary because it significantly decreases the number of vertices in a model, and also provides a smoothed model that looks better when rendered. Blender then exported each file in the FBX file type necessary for the Unreal Engine 3. Unreal Development Kit's level editor was then used to import the files. While the files are all correctly imported with respect to size and relative location, they are not rotationally oriented to the Unreal Engine's coordinate system. This is due to the fact that the *kinfu* package does not know which way is up and down. Due to this fact, the rotate tool was used to align the meshes into the game world.

At this point you are ready to drop into the engine and explore in first person mode. The UDK comes with a default "game mode" that allows your character to walk around with the standard WASD keys. There are several more features of the UDK that can be used to enrich this experience. One such option is to apply collision meshes to the models when they are imported. There are three options for collision meshes. You can either use a pre-modeled mesh, which we

do not have. The other two options are to use the actual graphical mesh or have UDK compute a collision mesh for you. Either of these methods will work. Using the graphical mesh is much slower, but provides much more accurate results. The UDK generated collision mesh generally provides adequate collision meshes, but it can struggle with concave shapes.

There are dozens of other UDK features that could be applied to this project but are not explored thoroughly. A brief list of these features is ambient noise, atmospheric lighting, time of day lighting, and running on a mobile phone. These are a few of many, many features the UDK has to offer.

The results of this experiment are given below. The 3D mapping process produced 27 total files. A screenshot of the dense point clouds and triangulated mesh before optimization can be seen in Figure 41 along with a picture of the ATRC 304 lab for comparison. The triangulated meshes were then optimized using MeshLab. They were then exported to the FBX file type and imported into the Unreal Engine 3. The meshes were then placed into the level and rotated into place. Table 8 shows the properties of several of these files throughout the experiment. You may note that the FBX and DAE file types are capable of storing more properties than the PLY file, and will be slightly larger when storing the same mesh. Also note the difference between the optimized and UDK triangle and vertex counts are due to the way the different engines interpret the mesh. Figure 42 shows the optimized meshes and a screenshot of the default UDK game mode being played on a level composed of our optimized, smoothed meshes. Figure 43 shows the entire mapped mesh of ATRC 304.

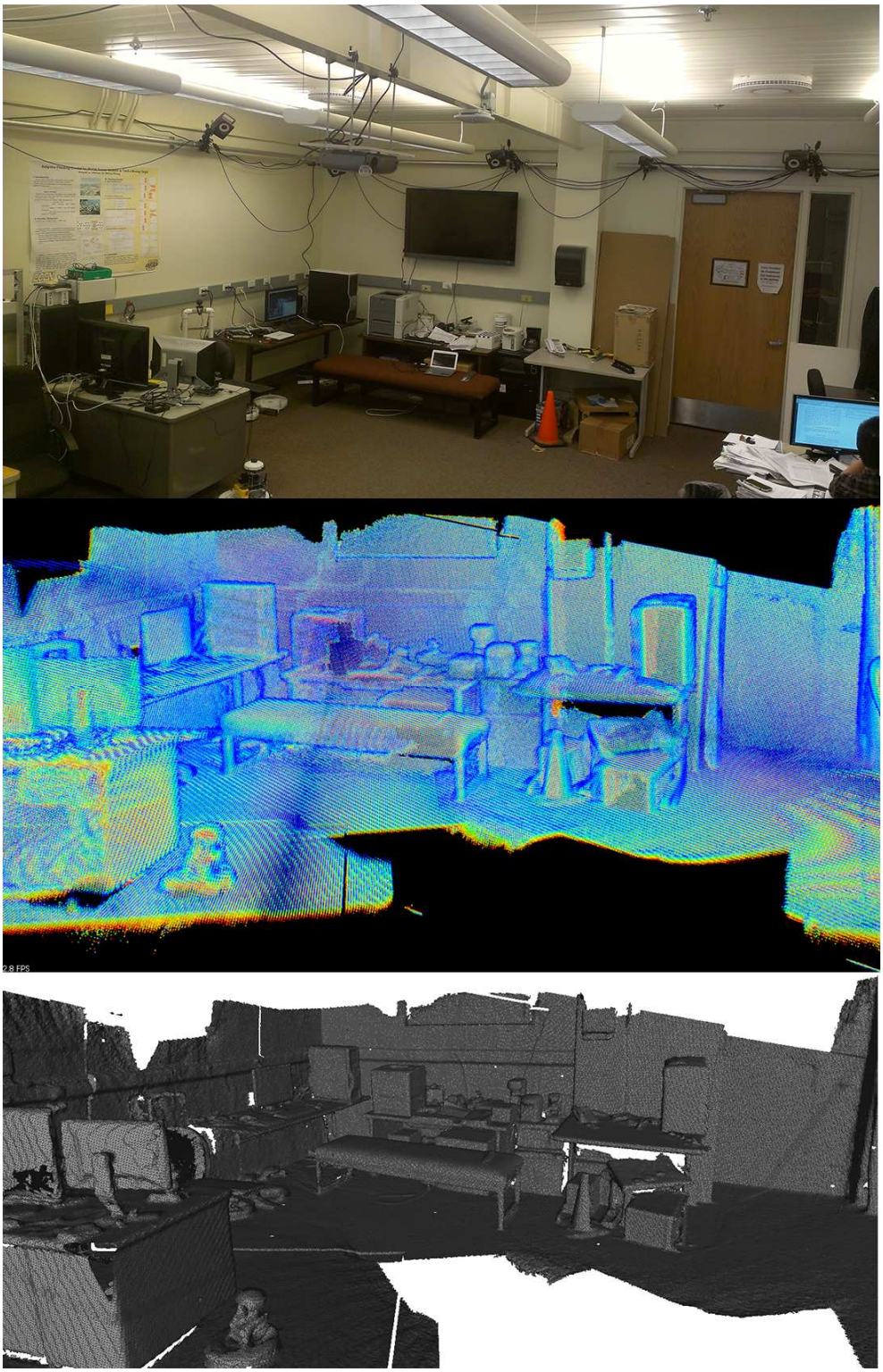

Figure 41:Top: Picture of ATRC 304 Lab. Center: Point clouds representing ATRC 304 lab. Bottom: The 3D map of the same area.

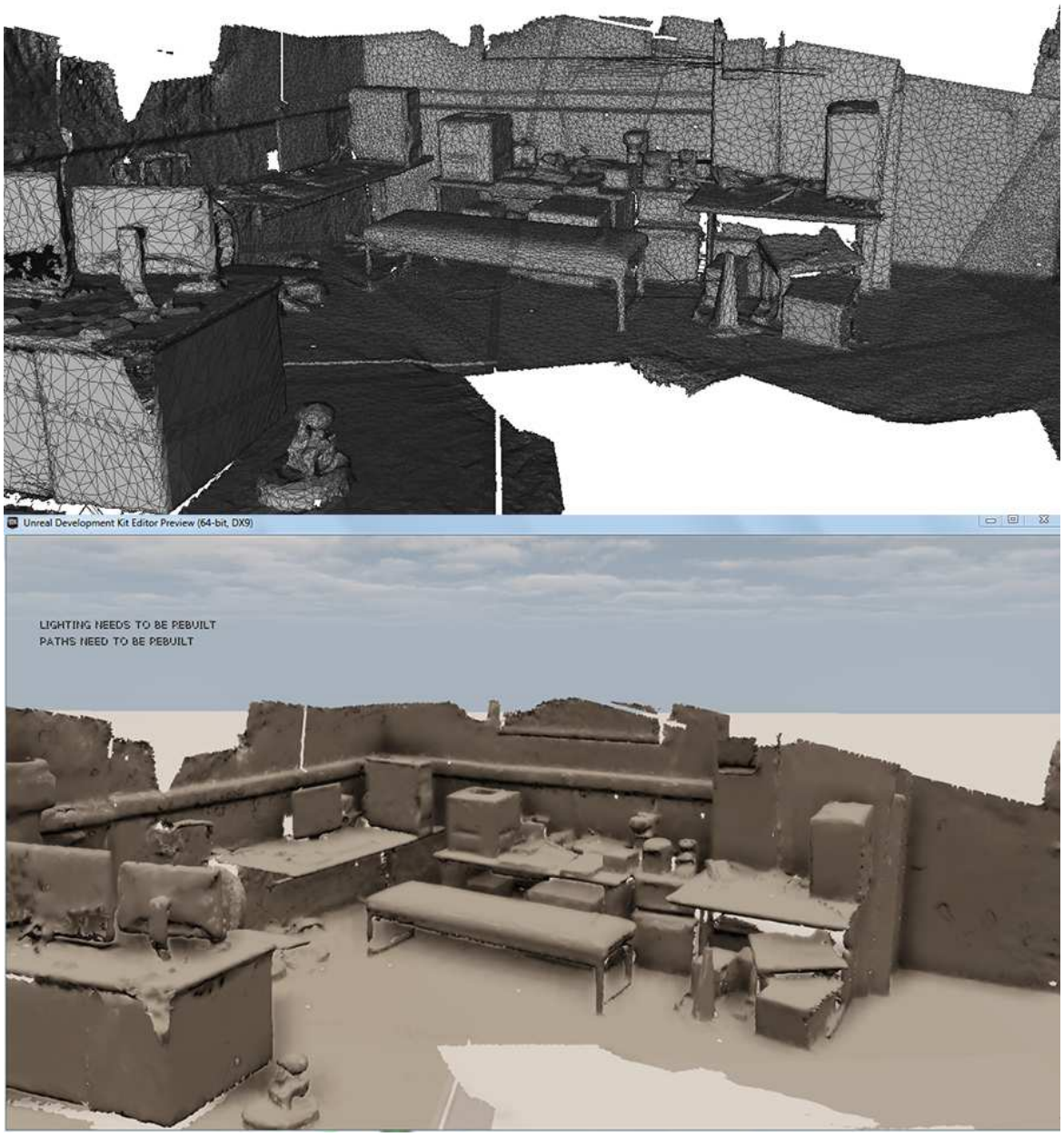

Figure 42: Top: Optimized meshes in MeshLab. Bottom: In game screenshot of ATRC 304 lab.

Mesh	3D Mapping Output			Optimized			UDK		
	Vertices	Triangles	PLY File Size (kB)	Vertices	Triangles	DAE File Size (kB)	Vertices	Triangles	FBX File Size (kB)
1	850,020	283,340	26,281	39,299	56,556	4,410	27,446	42,721	4,557
2	996,483	332,161	31,476	46,934	66,394	5,241	32,366	47,037	5,380
3	123,213	41,071	3,570	6,572	10,252	732	4,898	8,000	804
4	273,576	91,192	8,408	7,804	11,382	840	5,669	9,043	940
5	2,253	751	64	159	187	16	117	126	92
6	384,972	128,324	11,331	24,478	32,003	2,478	15,093	20,985	2,764
7	315,687	105,229	9,047	17,883	26,244	1,969	12,942	18,626	2,076
8	464,289	154,763	13,446	26,378	38,606	2,937	18,923	28,835	3,037
9	339,165	113,055	9,979	21,285	28,194	2,177	13,585	18,756	2,421
10	10,212	3,404	282	630	847	61	573	760	142

Table 8: The optimization of 10 meshes from the output of the mapping process through the rendering in the Unreal Engine 3.

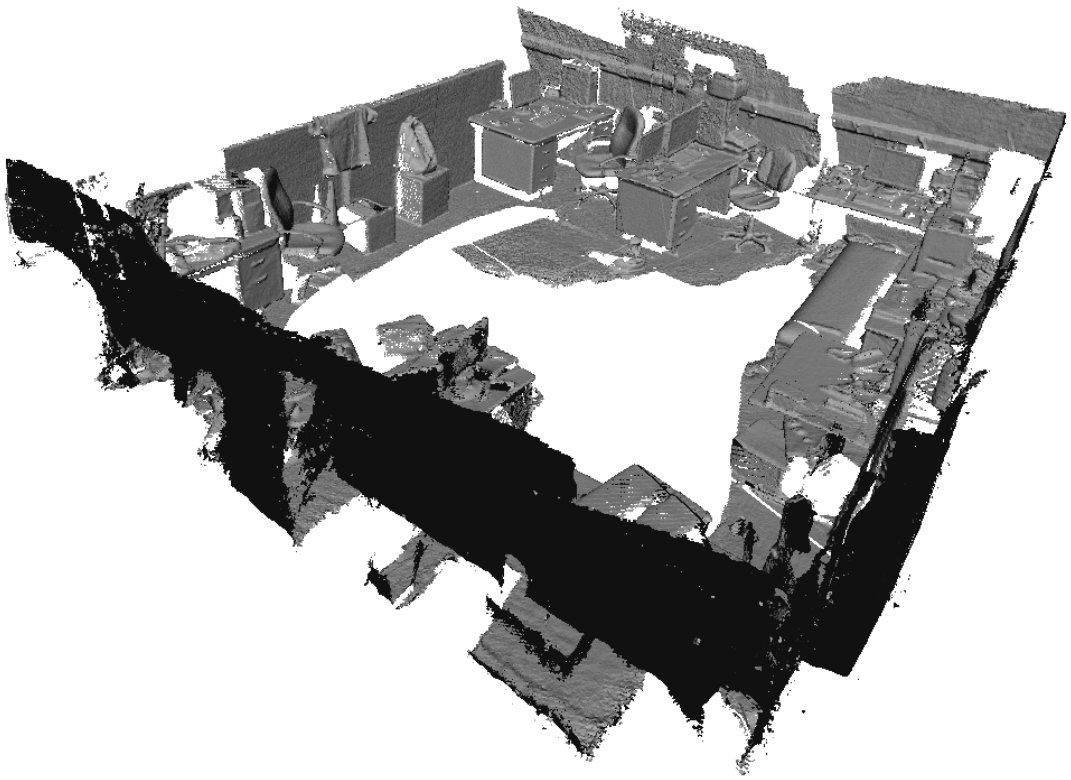

Figure 43 : ATRC 304 triangulated mesh

4.11 Problems

This thesis provides an implementation for creating 3D maps using a portable scanner and a depth map. However, there are several problems that have been faced which could improve the overall quality. These issues will be briefly discussed in this section to provide a better understanding of the issues.

One issue that this implementation suffers from is the reliability of the tracking. Areas which do not provide enough features in the depth image are difficult to track. Tracking also only uses the information currently in the TSDF volume to estimate the transformation. This means if

there are valid features, but they lie outside of the 3 meter cube, they are not used. This requires a relatively tightly packed environment for a complete scan.

A second issue is the lack of a loop closure algorithm to prevent drift. While the TSDF tracking is accurate, it still suffers from accumulated drift. This problem was clearly shown in Figure 31. Even a small amount of drift with each shifting of the TSDF can accumulate into large errors when mapping a larger environment.

While the image compression used in this implementation provided adequate bandwidth for our testing, it would be ideal to further compress the depth map. Utilizing either video compression or new research on depth image compression would decrease the uploading requirements that may prohibit some users from using the current implementation.

Another issue faced is the scanning of reflective or bright surfaces. Objects which are very reflective or are brightly lit are difficult to scan because the infrared pattern will be either reflected or diminished. This will often times result in an incomplete or incorrect map being created.

A final issue is the use of ROS for networking. ROS requires a static IP address and several ports for communication. This is not ideal for many applications as it requires extensive setup and cannot use networks which cannot forward the ports. For a more practical application, ROS could be replaced by transmitting the images over a socket. The compression could be done using OpenCV's libraries [85].

CHAPTER V

CONCLUSIONS AND FUTURE WORK

This thesis provides a method for generating 3D maps using a portable scanning device and cloud computing. It also provides a method for the triangulation, optimization, and rendering of the 3D map. Chapter 1 provided an introduction to the 3D mapping process and discussed several related works. In chapter 2 the hardware setup used for this thesis was discussed and also a few alternatives for the portable scanning device were mentioned. Chapter 3 covered the software used in this thesis. This includes the point cloud acquisition through PCL's KinfuLS, triangulation using the marching cubes algorithm, and optimization using the quadric mesh optimization algorithm. It then discussed using the Unreal Engine 3 for the rendering of this optimized 3D map in a game engine to provide an easy to use method for exploring the map. In chapter 4 the results of 9 experiments were provided to help validate the 3D mapping process and provide a quantitative insight into the results.

There are still several areas that could be further researched to improve upon this thesis. One of these areas is the inclusion of RGB data into the process. RGB information could be used to help track features in the images in which depth data does not provide adequate information to

accurately estimate the transformation. It could also be used to create more realistic renderings. To use add color to the optimized meshes in the Unreal Engine would require the UV mapping of the meshes. This cannot currently be automated and is manually performed. If the UV mapping process were automated, it would enable the “baking” process to be added into this thesis. Baking allows you to take a high resolution model and create normal maps, displacement maps, and color textures for use with the optimized mesh. This would allow for much more realistic rendering.

The second key areas for future work would be to correct the drift over time issue. This could be done using the loop closure method from [4]. This would allow for much more accurate results when scanning in larger areas. Many applications for this 3D mapping system would rely on having accurate maps, so correcting the drift issue would be very important.

Future work could be used to apply this thesis to many different fields. One area would be to use robots to map areas using this method. 3D maps could be created that would be easy for both the robots and humans to interpret. Collaborative mapping could also be used to allow multiple robots to generate their own map. Then when the server detects that robots maps overlap, they can be combined to create a unified larger map.

REFERENCES

[1] http://gizmodo.com/5563148/microsoft-xbox-360-kinect-launches-november-4

[2] http://aidsanity.deviantart.com/art/Interior-Design-wireframe-258796505

[3] http://www.softree.com/articles/LiDARWorkshop.pdf

[4] Henry, Peter, et al. "RGB-D mapping: Using depth cameras for dense 3D modeling of indoor environments." *the 12th International Symposium on Experimental Robotics (ISER)*. Vol. 20. 2010.

[5] Lowe, David G. "Object recognition from local scale-invariant features." *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*. Vol. 2. Ieee, 1999.

[6] Zhang, Zhengyou. "Iterative point matching for registration of free-form curves and surfaces." *International journal of computer vision* 13.2 (1994): 119-152.

[7] Fischler, Martin A., and Robert C. Bolles. "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography." *Communications of the ACM* 24.6 (1981): 381-395.

[8] Huang, Albert S., et al. "Visual odometry and mapping for autonomous flight using an RGB-D camera." *International Symposium on Robotics Research (ISRR)*. 2011.

[9] Engelhard, Nikolas, et al. "Real-time 3D visual SLAM with a hand-held RGB-D camera." *Proc. of the RGB-D Workshop on 3D Perception in Robotics at the European Robotics Forum, Vasteras, Sweden*. Vol. 2011. 2011.

[10] Bay, Herbert, Tinne Tuytelaars, and Luc Van Gool. "Surf: Speeded up robust features." *Computer Vision–ECCV 2006* (2006): 404-417.

[11] Grisetti, Giorgio, et al. "Hierarchical optimization on manifolds for online 2D and 3D mapping." *Robotics and Automation (ICRA), 2010 IEEE International Conference on*. IEEE, 2010.

[12] Rosten, Edward, and Tom Drummond. "Machine learning for high-speed corner detection." *Computer Vision–ECCV 2006* (2006): 430-443.

[13] Rublee, Ethan, et al. "ORB: an efficient alternative to SIFT or SURF." *Computer Vision (ICCV), 2011 IEEE International Conference on*. IEEE, 2011.

[14] Hartmann, Jan, et al. "Real-Time Visual SLAM Using FastSLAM and the Microsoft Kinect Camera." *Robotics; Proceedings of ROBOTIK 2012; 7th German Conference on*. VDE, 2012.

[15] Endres, Felix, et al. "An evaluation of the RGB-D SLAM system." *Robotics and Automation (ICRA), 2012 IEEE International Conference on*. IEEE, 2012.

[16] Wurm, Kai M., et al. "OctoMap: A probabilistic, flexible, and compact 3D map representation for robotic systems." *Proc. of the ICRA 2010 workshop on best practice in 3D perception and modeling for mobile manipulation*. Vol. 2. 2010.

[17] Zou, Yuhua, et al. "Indoor localization and 3D scene reconstruction for mobile robots using the Microsoft Kinect sensor." *Industrial Informatics (INDIN), 2012 10th IEEE International Conference on*. IEEE, 2012.

[18] Shi, Jianbo, and Carlo Tomasi. "Good features to track." *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR'94., 1994 IEEE Computer Society Conference on*. IEEE, 1994.+

[19] Izadi, Shahram, et al. "KinectFusion: real-time 3D reconstruction and interaction using a moving depth camera." *Proceedings of the 24th annual ACM symposium on User interface software and technology*. ACM, 2011.
[20] Newcombe, Richard A., et al. "KinectFusion: Real-time dense surface mapping and tracking." *Mixed and Augmented Reality (ISMAR), 2011 10th IEEE International Symposium on*. IEEE, 2011.
[21] Whelan, Thomas, et al. "Kintinuous: Spatially Extended KinectFusion." (2012).
[22] Roth, Henry, and Marsette Vona. "Moving Volume KinectFusion." *British Machine Vision Conf.(BMVC),(Surrey, UK)*. 2012.
[23] Owen, Steven J. "A survey of unstructured mesh generation technology." *7th International Meshing Roundtable*. Vol. 3. No. 6. 1998.
[24] Yerry, Mark A., and Mark S. Shephard. "Automatic three-dimensional mesh generation by the modified-octree technique." *International Journal for Numerical Methods in Engineering* 20.11 (2005): 1965-1990.
[25] Watson, David F. "Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes." *The computer journal* 24.2 (1981): 167-172.
[26] Löhner, Rainald. "Progress in grid generation via the advancing front technique." *Engineering with Computers* 12.3 (1996): 186-210.
[27] Weatherill, Nigel P., and Oubay Hassan. "Efficient three-dimensional Delaunay triangulation with automatic point creation and imposed boundary constraints." *International Journal for Numerical Methods in Engineering* 37.12 (1994): 2005-2039.
[28] Lorensen, William E., and Harvey E. Cline. "Marching cubes: A high resolution 3D surface construction algorithm." *ACM Siggraph Computer Graphics*. Vol. 21. No. 4. ACM, 1987.
[29] Johansson, Gunnar, and Hamish Carr. "Accelerating marching cubes with graphics hardware." 2006.
[30] Cignoni, Paolo, Claudio Montani, and Roberto Scopigno. "A comparison of mesh simplification algorithms." *Computers & Graphics* 22.1 (1998): 37-54.
[31] Garland, Michael, and Paul S. Heckbert. "Surface simplification using quadric error metrics." *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. ACM Press/Addison-Wesley Publishing Co., 1997.
[32] Trenholme, David, and Shamus P. Smith. "Computer game engines for developing first-person virtual environments." *Virtual reality* 12.3 (2008): 181-187.
[33] http://www.crytek.com/cryengine
[34] http://www.idsoftware.com/business/history/
[35] http://source.valvesoftware.com/
[36] http://www.unrealengine.com/
[37] http://unity3d.com/
[38] http://www.newegg.com/Product/Product.aspx?Item=N82E16826785047
[39] http://www.newegg.com/Product/Product.aspx?Item=N82E16834215243
[40] http://www.xbox.com/en-US/KINECT
[41] http://openkinect.org/wiki/Main_Page
[42] http://www.asus.com/Multimedia/Motion_Sensor/Xtion_PRO/
[43] http://www.asus.com/Multimedia/Motion_Sensor/Xtion_PRO_LIVE/
[44] http://www.openni.org/
[45] http://www.csd.toshiba.com/cgi-bin/tais/support/jsp/modelContent.jsp?ct=SB&os=&category=&moid=2523338&rpn=PLL3FU&modelFilter=NB305-N310&selCategory=2756709&selFamily=2336267
[46] http://www.fit-pc.com/web/fit-pc/fit-pc2i-specifications/
[47] http://www.arduino.cc/

[48] http://beagleboard.org/
[49] http://www.raspberrypi.org/
[50] http://www.gartner.com/it/page.jsp?id=1980115
[51] http://www.gartner.com/it/page.jsp?id=1924314
[52] http://pointclouds.org/blog/nvcs/raymondlo84/index.php
[53] http://raymondlo84.blogspot.com/2012/09/using-accelerometer-with-kinectxtion-on.html
[54] https://developer.nvidia.com/what-cuda
[55] http://www.visualization.hpc.mil/wiki/CUDA_Programming_Model
[56] Mell, Peter, and Timothy Grance. "The NIST definition of cloud computing (draft)." *NIST special publication* 800 (2011): 145.
[57] Rusu, Radu Bogdan, and Steve Cousins. "3d is here: Point cloud library (pcl)." *Robotics and Automation (ICRA), 2011 IEEE International Conference on*. IEEE, 2011.
[58] Du, Jianhao, Ou, Yongsheng and Weihua Sheng. “Improving 3D Indoor Mapping with Motion Data.” I*EEE International Conference on Robotics and Biomimetics.* IEEE, 2012.
[59] Curless, Brian, and Marc Levoy. "A volumetric method for building complex models from range images." *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. ACM, 1996.
[60] http://www.pointclouds.org/blog/srcs/fheredia/all.php
[61] http://en.wikipedia.org/wiki/Triangle_mesh
[62] http://meshlab.sourceforge.net/
[63] http://www.ros.org/wiki/
[64] http://www.ros.org/news/2011/11/celebrating-the-fourth-anniversary-of-ros-and-the-first-roscon-2012.html
[65] http://ros.org/wiki/openni_camera
[66] http://www.ros.org/doc/api/sensor_msgs/html/msg/CameraInfo.html
[67] http://www.ros.org/wiki/image_transport
[68] http://www.theora.org/
[69] http://www.ros.org/wiki/image_view
[70] http://www.ros.org/wiki/roscore
[71] http://ros.org/wiki/tf
[72] http://www.ros.org/wiki/rxgraph
[73] http://www.ros.org/wiki/rxplot
[74] http://www.unrealengine.com/en/udk/downloads/
[75] http://udn.epicgames.com/Three/UnrealUnits.html
[76] http://udn.epicgames.com/Three/FBXStaticMeshPipeline.html
[77] http://usa.autodesk.com/fbx/
[78] Ecma International, Universal 3D File Format, 3rd Edition, June, 2006, ECMA 363
[79] http://www.blender.org/
[80] http://www.pcl-developers.org/Kinfu-Improvements-td5135752.html
[81] des Bouvrie, Bas. "Improving RGBD Indoor Mapping with IMU data." (2011).
[82] http://ros.org/wiki/rviz
[83] http://www.ros.org/wiki/rgbdslam
[84] http://www.ros.org/wiki/rostopic
[85] http://opencv.willowgarage.com/wiki/

APPENDICES

ROS Commands used for this thesis:

To run the mapping software:

rosrun kinfu kinfuLS _image_transport:=compressedDepth

To view the status image:

rosrun image_view image_view ige:=/kinfuLS/status compressed

To plot 6DOF transformations in real time:

rxplot -M 3d /tf/transforms[0]/transform/translation/x:y:z

To view real time 6DOF pose:

rosrun rviz rviz

To run the roscore:

roscore

VITA

CRAIG MOUSER

Candidate for the Degree of

Master of Science

Thesis: REALTIME 3D MAPPING, OPTIMIZATION, AND RENDERING BASED ON A DEPTH SENSOR

Major Field: Electrical and Computer Engineering

Biographical:

Education:

Completed the requirements for the Master of Science in Electrical and Computer Engineering at Oklahoma State University, Stillwater, Oklahoma in December 2012.

Completed the requirements for the Bachelor of Science in Electrical and Computer Engineering at Kansas State University, Manhattan, Kansas in May 2011.