VOXEL OCTREE INTERSECTION BASED 3D SCANNING

JOEL BENNETT
Bachelor of Science, University of Lethbridge, 2008

A Thesis
Submitted to the School of Graduate Studies
of the University of Lethbridge
in Partial Fulfillment of the
Requirements for the Degree

MASTER OF SCIENCE

Department of Mathematics and Computer Science
University of Lethbridge
LETHBRIDGE, ALBERTA, CANADA

VOXEL OCTREE INTERSECTION BASED 3D SCANNING

JOEL BENNETT

Approved:

Signature *Date*

Co-Supervisor: Dr. Stephen Wismath

Co-Supervisor: Dr. Kevin Grant

Committee Member: Denton Fredrickson

Committee Member: Dr. David Kaminski

Chair, Thesis Examination Committee: Dr. Howard Cheng

Dedication

To those who keep trying. Don’t ever give up.

Abstract

Recent developments in the field of three dimensional (3D) printing have resulted in widely available low-cost 3D printers. These printers require 3D models, which are traditionally created in 3D modeling software or are created from 3D scans of existing objects. To be printable, these models must exhibit the property of being watertight. In this thesis, a technique is developed which, in combination with a custom built low-cost 3D scanner, produces watertight 3D models. Models produced by this technique – the voxel octree intersection technique – do not require any additional processing prior to 3D printing. Results from using this technique with the custom built scanner are examined, along with the effects of changing various parameters to the technique.

Acknowledgments

First, I'd like to thank my immediate family – both my wife and my parents in supporting me along the way. To my dad, thank you for exposing me to all sorts of interesting science videos when I was young, and for having a computer in the home. Who would have thought that you would end up having a child who would be a computer programmer?

Next, I'd like to thank my co-supervisors – Stephen Wismath and Kevin Grant. You've been crazy enough to take me on as a graduate student, and giving me the freedom to pursue some pretty wild ideas. This thesis is the tip of an iceberg of awesomeness that was birthed in the HCI lab.

To my former lab-mate Chris Sanden, thank you for the interesting conversations and showing me that success in both academia and the private sector is possible. Bouncing ideas off each other in the lab was an absolute blast, and a time I'll never forget. David Fox – thank you for helping me with the stepper motor wiring. To Kevin and Catherine in the Arts department – thank you for letting me use your equipment and your time.

I'd also like to offer a special thank you to the Government of Alberta and federal government of Canada for their respective funding. Thank you for remembering that the future starts now, and thank you for being willing to invest in it.

To Stanford University – thank you for the bunny and dragon models. There's something ironic about a rabbit that has been scanned, printed, then re-scanned and re-printed.

I am sure that there are so very many others that I could thank. All of you, in your own individual way, have had an effect on me. Here's to hoping that I'll be able to make my own 'dent' in the universe, and have a positive effect on others along the way...

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Recent developments in the field of three dimensional (3D) printing have lowered the cost of commercially available 3D printers. For example, a low-cost 3D printer can be purchased for less than \$600 [12], while only three decades ago the cost of a similarly capable printer could easily have been ten times that amount [11]. As a result of these recent developments, 3D printers and 3D printing services are now reaching a larger audience than ever before.

The dropping costs and commercial availability of 3D printers and 3D printing services have led to a revolution in personal manufacturing. A single individual with a low-cost 3D printer can now prototype complex objects with minimal cost in minimal time. Not only can an individual quickly revise a design, but the prerequisites for manufacturing have also been lowered. Instead of needing complex and expensive machinery, all that is need to produce an item is a computer, a 3D printer, raw printing materials, and a design. When working with 3D printing and computer numerical control (CNC) machining, these designs come in the form of a 3D model.

In order to create a 3D print or CNC machined object, a 3D model of the object is a requirement. These models are traditionally constructed using computer aided design (CAD) software, or may be created from a 3D scan of an existing object. Both approaches have advantages and disadvantages. Regardless of how these 3D models are constructed, they must exhibit the property of being *watertight*. This means that there cannot be any holes, gaps, or open seams along the surface of the object. If an object were filled with water, it should not leak. Objects which are 3D printed or CNC machined must be watertight [7].

The process of 3D printing or CNC machining must be able to determine which regions of an object represent the inside, outside, and surface of the object being created. If it is impossible to make this distinction, it becomes impossible to create an accurate re-creation of the object in question.

In this thesis, a technique is developed that uses a custom built, low-cost 3D scanner and simple scanning technique to create watertight 3D scans. By using commercially available, off-the-shelf components in a unique hardware arrangement, we are able to construct a simple low-cost 3D scanner. By processing data from this scanner in a series of specific steps, we are able to generate a watertight 3D model which does not require any additional processing prior to 3D printing. This particular technique for generating a watertight 3D model is known as the voxel octree intersection technique.

1.1 3D Printing

3D printing is the process of creating a three dimensional physical model from a digital source. Similar to how a traditional inkjet or laser printer creates a two dimensional image on a piece of paper, a 3D printer creates a three dimensional object. Commercially available 3D printers generally use one of the following methods to create a solid print:

Fused deposition modeling With fused deposition modeling, a printer melts and ejects a continuous strand of filament. A computer guided nozzle moves around the printing area, ejecting the melted filament. This molten filament hardens, and forms a single layer of the object. When a layer is complete, distance between the completed layer and the printing head is increased, and the next layer is deposited on top of the previously printed layer. This process is continued until the entire object has been constructed, layer by layer. Common filament materials include PLA (polylactic acid) and ABS (acrylonitrile butadiene styrene) plastic.

Stereolithography Stereolithography relies on curing a photosensitive liquid using a beam of light. A computer guides the light onto areas where solid material is desired. When all desired areas of a single layer have been exposed to the light, the current layer is submerged in the photosensitive liquid, covering the recently exposed layer in more liquid. This layer of fresh liquid is then cured in the same way as the previous layer. The process is repeated until all layers of the object have been cured.

Sintering With sintering, a heat source such as a heat gun or laser is used to melt the shape of the desired object into a layer of powered substrate. When the desired shape is melted, more substrate is added on top, covering the previously created shape. The next shape is melted into this new layer of substrate, fusing it to the material of the previous layer. This process is repeated until all layers of the object have been created, after which the remaining unfused substrate is then removed leaving only the desired object. The advantage of this technique over fused deposition modeling and stereolithography is that the substrate acts as a support while the object is being printed. This makes it possible to print complex objects without the need for adding additional supports to the object during the printing process. A modified version of this technique may apply a binding agent rather than a heat source to bind together particles of the substrate.

1.1.1 Uses of 3D Printing

Although 3D printing has been around for several decades [30], recent developments in low-cost 3D printing have sparked a renewed interest in 3D printing technologies. Recent uses for 3D printing have included printing personal firearms [49], bone scaffolding [15], oral surgery implants [14], replacement car parts [33] and aerospace components [38]. Because of its ability to create complex components, 3D printing is quickly becoming a choice manufacturing technology for prototyping objects when compared to traditional manufacturing technologies.

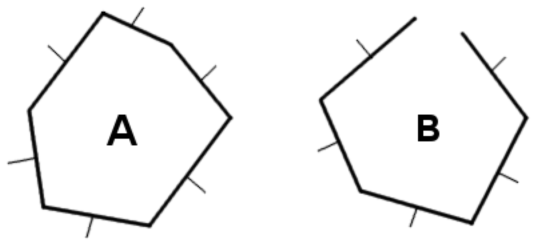


Figure 1.1: An example of an open and closed shape.

1.2 Watertight Geometry

All of the previously mentioned 3D printing methods create the final printed object layer by layer. To do this, the printer must have a set of instructions on how to construct each layer. This is done by using computer software to slice the original 3D model, creating a series of cross-sections along the height of the object. The software then instructs the printer on how to move the print head to re-create each of these slices. In each of these slices, areas are marked as being either inside, outside, or along the contour of the surface of the object being printed. Because the 3D printing process requires knowing these distinct areas, printing problems can arise if a model is not watertight.

Figure 1.1 shows an example of the cross-section of two shapes. Thick lines denote the surface of the object and thin lines indicate the direction of surface normals. If a simple flood-fill algorithm was used to color the interior of shape A, only the interior of shape A would be filled with color. If the same flood-fill algorithm was used to color the interior of shape B, nearly the entire image would be filled with color with the exception of the interior of shape A. When 3D printing, in order to distinguish which areas need to be printed, there must be no ambiguity as to whether or not a particular region belongs to the interior, exterior, or contour of an object. Such ambiguity may result in regions being filled by the 3D printer which should not otherwise be printed. For example, in Figure 1.1, if objects A and B were printed, the entire layer would be marked as being either the contour or the interior of an object, and would be printed as such, resulting in an undesirable print.

1.3 3D Scanning

Three dimensional scanning is the process of collecting information from a scene or object, and using that information to digitally reconstruct the scene or object being scanned. Information collected may include 3D positions, surface normals, and color information about specific points in 3D space. Current 3D scanning methods can be placed into one of two categories: those that require physical contact between the scanning device and object being scanned, and those that do not.

1.3.1 Contact-Based Scanning

Contact-based scanning systems use a physical probe that contacts the surface of the object being scanned. For example, a high-end Computer Numerical Control (CNC) machine may use a Coordinate Measuring Machine (CMM) attachment to determine the dimensions of a workpiece prior to machining. Other physical contact 3D scanners use a series of linkages and joints. The orientations of the joints are tracked, and are used to located the probe end of the device in three dimensions. Destructive scanning methods are also included in this category, as they require the object being scanned to be cut or abraded into a series of slices. Measurements are taken at each slice, and are used to re-create the object.

Although highly accurate [46], able to scan reflective surfaces, and often used in the machining industry, these types of scanners do have some drawbacks. The speed at which they can operate are physically limited, as they are often only able to measure a single point at a time. They also require physical contact between the scanner and object at each scanned point, which may be seen as a drawback when attempting to scan objects that are of a fragile or biological nature.

1.3.2 Contactless Scanning

Most contactless scanning methods work by emitting a beam or pattern of light which is bounced off the surface of the object being scanned. One or more properties of the reflected light are analyzed and used to calculate the distance between the emitter and returned beam.

These measurements are then used to reconstruct the object being scanned.

Triangulation

With triangulation based scanning, a laser point or line is projected into the scene. The laser light bounces off the object being scanned, and is picked up by an optical sensor, such as a CCD (charge coupled device) or CMOS (complementary metal oxide semiconductor) sensor. The laser emitter and sensor are held a fixed distance from each other, with the laser light being emitted at a known angle. Based on data coming from the optical sensor, it is possible to create a triangle using the position of the optical sensor, laser, and sensed laser dot or line. Because the distance between the laser emitter and optical sensor are known, and the angle of the emitted laser light is also known, this can be used to triangulate the position of the reflected laser light. An example of a commercial product that uses this method is the Makerbot Digitizer [23].

Structured Light

Structured light scanners rely on a similar principle to triangulation based scanners, but rather than emitting a single point or beam of light, they use emitters that project a set pattern of light. Discrepencies are noted between the projected pattern and the pattern that is detected with the optical sensor. The distortion in the projected pattern is then used to calculate depth values, which are used to generate a model of the object being scanned [42]. In order to obtain quantifiable depth measurements, structured light scanners must be calibrated prior to use.

Confocal Laser Scanning Microscopy

With confocal laser scanning microscopy, a laser light is passed through a series of narrow apertures and lenses, and is captured on an optical sensor. The amount of diffraction in the received light is measured, and is used to determine the distance to the object which reflected it. Each measurement retrieves a single 3D point. This method is capable of capturing incredibly small objects, such as individual cells or integrated circuit components [44]. Because confocal laser scanning microscopy captures only one point at a time, it is limited in scanning speed.

Photogrammetry

Unlike previously mentioned methods, photogrammetry does not require projecting a light into the scene being scanned. With photogrammetry, two or more 2D images are taken of an object from different positions. Unique features of the object being photographed are identified, and the relationships between these features are used to reconstruct a three dimensional model of the object in question [45]. A recently released software package called 123D Catch by Autodesk© uses this technique to generate 3D models based on a series of 2D images. Although this method does potentially allow reflective objects to be scanned, there is some doubt as to whether it produces better results than laser based triangulation methods [6].

Volumetric

As with photogrammetry, volumetric scanning methods do not rely on projected light. These scanning methods include Computer Tomography (CT) and Magnetic Resonance Imaging (MRI) scans. These methods rely on capturing a series of two dimensional images along a specified axis. These 2D images are then compiled into a volumetric model of the object. The volumetric models produced by these methods are often rendered using ray-tracing [18], or are converted to polygonal models [31] and rendered in a traditional manner.

Time-of-Flight

Time-of-flight based scanners emit measured pulses of light. These pulses of light reflect off the surface being scanned, and bounce back toward a high-speed sensor. The sensor is able to measure the delay between when the light was emitted and when it was sensed.

This, combined with the speed of light, is used to calculate the distance to the object which reflected the light. Time-of-flight sensors include most LIDAR systems, as well as the Microsoft Kinect 2.0. Although these types of sensors are capable of many measurements per second, they may suffer from considerable noise [16]. Like other contactless based scanning methods which rely on projected light, they are also unable to scan surfaces which are highly reflective, refractive, or which absorb the particular wavelength of emitted light.

1.3.3 Uses of 3D Scanning

Some of the many uses of 3D scanning include digitally re-creating race tracks for video games [21], preserving sculptures and museum exhibits of historical importance [29], replicating movie sets for use with visual effects [1], reverse engineering, prosthetics fitting, and medical diagnostics.

1.4 Objectives

This thesis describes a technique for taking data captured from a Microsoft Kinect depth sensor and converting that data into a 3D watertight mesh. Because the end result is watertight, the mesh is immediately suitable for 3D printing or CNC machining without the need for any additional processing. This technique was developed with the following restrictions in mind:

- To be able to generate a watertight 3D model without the need for any post-processing.
- To be capable of scaling detail, as needed.
- To not require video memory or additional GPU (Graphics Processing Unit) hardware during processing.
- To utilize only off-the-shelf, widely available, low-cost hardware components.

With these particular restrictions in place, it was necessary to develop a new technique for handling this specific situation. This technique takes a voxel-based approach, and uses an underlying tree data structure to accomplish these goals.

1.5 Structure of this Document

Chapter 2 looks at related work and describes in detail the workings of the Microsoft Kinect sensor. Chapter 3 provides a detailed description of the voxel octree intersection technique. Chapter 4 shows the results of using this technique, including the effects of altering various parameters to the technique. Chapter 5 discusses areas of potential future work which may be used to increase the efficiency and effectiveness of the voxel octree intersection technique. The appendices include the hardware design of the 3D scanning rig used to obtain the results, and show an approximate cost breakdown of the equipment used.

Chapter 2

Related Work and the Microsoft Kinect

2.1 Related Work

The field of surface reconstruction from point data is not a new problem, nor is it a simple problem [26]. Various algorithms have been developed that convert point cloud data into polygonal models [34]. Depending on the nature of the data being used, a particular algorithm may make use of structured or unstructured data, and create a polygonal mesh, spline-based surface, or volumetric model. Some methods take into account local or global fitting, including fitting points against known shapes [8]. Additionally, some methods produce watertight meshes while others do not. A sampling of these methods, including those that relate more closely to the voxel octree intersection technique are described in detail as follows.

2.1.1 Ball-Pivot Algorithm

The ball-pivot algorithm reconstructs a polygonal mesh based on a dense set of unorganized point data [9]. The algorithm does so by starting with a single seed triangle and a ball (sphere). The ball is placed on the edge of the seed triangle, and is rotated on the edge until it contacts another point in the point cloud. When contact is made, the edge upon which the ball has been rotating and the contacted point are used to form a triangle. This triangle is added to the output mesh, and the process is repeated along the edges belonging to the perimeter of the mesh. When no more contacts are encountered, a new seed triangle is selected, and the process is repeated. When the process yields no more contacts between any seed triangles or perimeter edges, the process is complete.

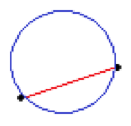


Figure 2.1: (a)

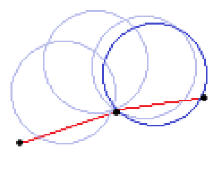


Figure 2.2: (b)

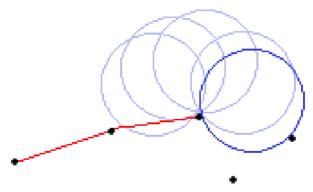


Figure 2.3: (c)

A two-dimensional representation of this process is shown in Figure 2.1. In the 2D case, the process begins by creating a seed edge between two points. A ball is created, and rotated around a single point. When the rotated ball contacts a point, an edge is created between the point of rotation and the contacted point.

The Ball-Pivot algorithm produces a polygonal mesh which is not guaranteed to be watertight. Although the algorithm is capable of generating a mesh from an unorganized point cloud, points must be regularly spaced in order to give ideal results. If too small of a ball is chosen or if points are irregularily spaced, the resulting mesh will contain holes. Similar issues happen when attempting to re-create acute surface features. When rotating the ball around an edge, the ball may contact the opposing side of an acute feature prior to contacting the inner portion of the feature. If this occurs, the resulting surface spans the gap between opposing sides of the feature, rather than creating an acute feature, as shown by example in Figure 2.3. In this example, the rotated ball failed to contact the lower point prior to contacting the rightmost point, resulting in a loss of detail of the final surface.

2.1.2 Surface Nets

The Surface Nets algorithm is designed to provide a smooth three dimensional model from data obtained through medical scanning procedures such as MRI or CT scans [20]. Volumetric data coming from medical scanners often comes in the form of a series of 2D slices. The resolution of this data may vary in different dimensions. For example, the distance between adjacent pixels in a single image may be considerably smaller than the distance between the same pixel between two adjacent slices. The Surface Nets algorithm is able to recreate a smoothed polygonal model of the surface that does not suffer from aliasing that other methods produce when dealing with such data [17].

The Surface Nets algorithm works by defining a set of cubes that have at least one corner belonging to the surface of the object being re-created. In each of these cubes, a node is placed in the center of the cube and adjacent nodes are connected with an edge. A function is then applied which relaxes the position of the node within the cube, with the restriction that the node must be kept inside the bounds of the cube. This same function attempts to minimize the distance between all edges in the connected net. Once the net has been relaxed, and the distances between the connected nodes have been minimized, a 3D polygonal surface is constructed by linking neighboring nodes with a set of polygons. Although the resulting mesh may be free from aliasing and topologically smooth when compared to other methods [28], there is no guarantee that the resulting mesh is watertight.

2.1.3 Power Crust

Unlike the previously mentioned algorithms, the Power Crust algorithm generates a watertight mesh [4]. It does this by calculating an approximate medial axis transform (MAT) of a set of points. The MAT can be thought of as the union of an infinite number of maximum-sized balls that are contained in a given shape. To calculate the MAT, the input points are first used to create a special weighted Voronoi diagram called the power diagram. A series of balls are created and applied to the power diagram. Each ball is expanded to its maximal size. Those balls which are able to expand indefinitely are considered to be outside the shape while those that are unable to expand make up the interior of the shape. The border between these two types of balls is used to establish the contour of the object being re-created.

By adjusting the radius of the interior balls, a thin crust of the object can be created.

This crust is then converted into two sets of triangular faces – those that border the inside of the object and those that border the outside of the object. The end result is a watertight set of geometry that forms a thin crust along the surface of the object. The accuracy of the re-created object depends heavily on the density of the original set of points. Although the algorithm is able to re-create a polygonal mesh from a set of unorganized point data, it relies on calculating the Delaunay triangulation of the set of points. The Delaunay triangulation has a worst case complexity of $O(n^2)$, although average cases are closer to being linear in time [4].

2.1.4 Poisson

Poisson surface reconstruction algorithms require point cloud data where each point in the point cloud has an associated normal [24]. The normal is used to calculate a gradient field, which is then used to solve a Poisson equation. To generate a polygonal mesh, the original surface points and results from the Poisson equation are considered. A modified version of the Marching Cubes algorithm is used to generate the resulting polygonal mesh. The particular Poisson based approach presented by [24] is resistant to noise, handles areas of both high and low detail, and produces a watertight mesh. A parallelized Poisson surface reconstruction algorithm has also been developed to work with particularly large datasets [10].

Although Poisson based reconstruction methods may produce watertight meshes with high detail, they may introduce some error into the generated mesh. If two groups of depth readings are separated by an empty region where no depth readings are available, the algorithm may fill the gap between these two regions, resulting in an incorrect topology.

2.1.5 Marching Cubes and Marching Tetrahedrons

One of the most common isosurface extraction algorithms is that of Marching Cubes. Although not explicitly a surface reconstruction algorithm, it is used by many other surface reconstruction algorithms to generate the resulting polygonal mesh. It does this by tracking

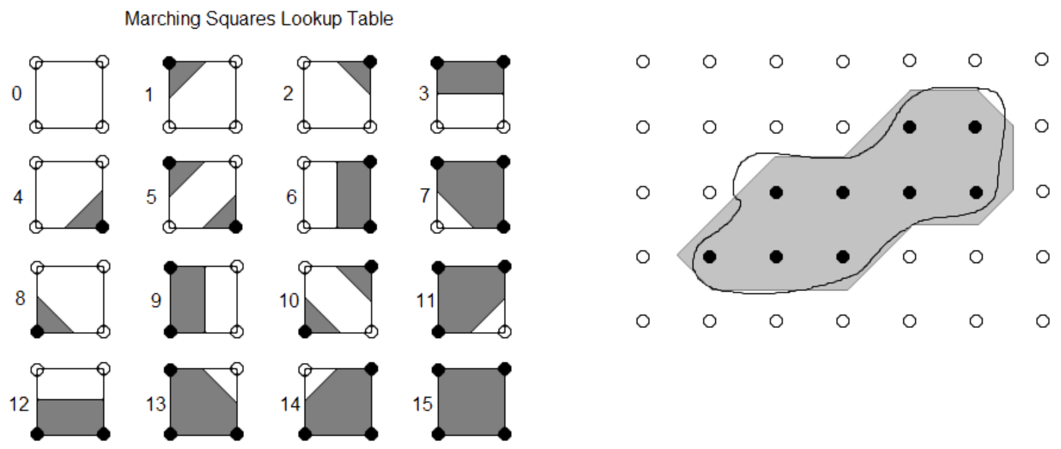


Figure 2.4: An example of applying the Marching Squares algorithm.

values at the corners of a grid of cubes. These corners are marked as either being inside or outside the intersecting surface. As there are 8 corners to a cube, there are 2^8 possible combinations of corners being inside or outside the surface. The 8 values for a single cube can be tracked using an 8-bit number, which is checked against a lookup table to find the particular set of geometry that best represents the surface intersecting the cube in that manner. This lookup is performed for each cube in the grid, and the resulting geometry is added to the final mesh. The lookup table used in the Marching Cubes algorithm can be simplified by recognizing that many entries in the table have rotational or mirrored equivalents. The original work by Lorensen and Cline simplified this into a table with only 15 entries.

A two-dimensional example of the algorithm, known as Marching Squares, is shown in Figure 2.4. In this figure, the lookup table for Marching Squares is shown on the left. On the right, the original isosurface is shown by the thin black line. Points which are considered to be inside the surface are black, while points outside the surface are white. The resulting shape from running the Marching Squares algorithm is shown in gray.

The accuracy of a surface created by Marching Cubes can be improved by either increasing the resolution of the underling grid or by altering the points at which the geometry intersects the edges of each cube. Rather than forcing each intersection to happen at the exact mid-point between two corners, the intersection point can be interpolated based on the values at each corner. This requires tracking more than just a binary inside/outside flag for each corner, but results in a much better fitting surface.

The original Marching Cubes algorithm suffers from some ambigious cases – situations in which the algorithm is unable to determine a topologically correct set of geometry based solely on the corner values of the cube. For example, Case 5 in the lookup table in Figure 2.4 shows two points as being inside the surface and two points outside the surface. The ambiguity comes from not knowing whether or not the isosurface simply passes through each respective side of the square or whether the isosurface creates a passage between the two points. As a result of this ambiguity, there is no guarantee that the original Marching Cubes algorithm will produce a watertight mesh. Additional lookup tables have been created to overcome this issue [13] [40].

A similar, but alternate method to Marching Cubes is Marching Tetrahedrons [47], or Marching Tetrahedra. Rather than operating on a cube, Marching Tetrahedrons operates on a single tetrahedron. As a result, the lookup table only needs to contain 2^4 elements, and the resulting geometry does not suffer from the same ambiguity as Marching Cubes.

2.1.6 KinectFusion

The KinectFusion project uses a Microsoft Kinect sensor to capture a 3D scene [39]. It does this by using a global model to track the overall scene, which comes in the form of a signed 512^3 volumetric grid. Changes in the position of the Kinect sensor are identified using the iterative closest-point algorithm. As the sensor is moved around the scene, the global model is updated to reflect newly captured data. The final visual of the scanned scene is rendered using raycasting. This expensive process is carried out on the GPU (graphics processing unit) in order to keep the global model updated at a near real-time speed of up to 10 Hz. Although the visual results from Kinect Fusion are impressive, there is no guarantee of a watertight mesh being generated. This approach also relies on a significant amount of

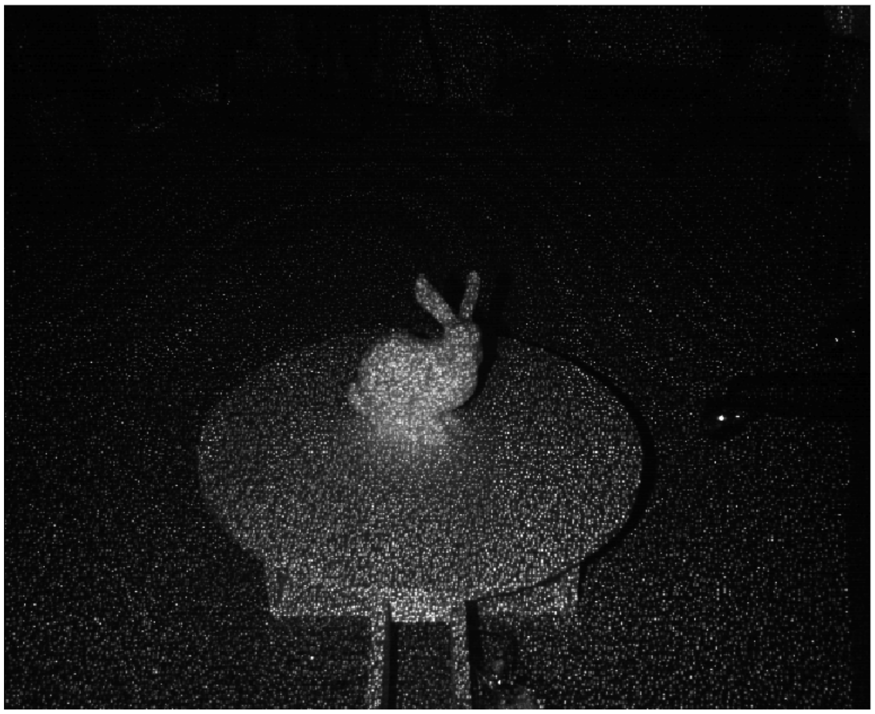


Figure 2.5: Infrared depth image taken from a Microsoft Kinect.

GPU processing and is limited in resolution because of the maximum size of the internal model. Some work has been done to adapt this approach to use an octree, with favourable results [48].

2.2 Microsoft Kinect

The Microsoft Kinect is a commercially available 3D camera system that was released by Microsoft in 2010. The sensor works by emitting a pseudo-random pattern of infrared light, which is reflected off a scene and captured on a separate CMOS sensor [5]. Disturbances in the infrared light are used to measure depth via triangulation, and are captured at a resolution of up to 640x480 at a rate of 30 times per second [25]. Figure 2.5 shows an example of an infrared image captured by the Kinect infrared sensor. A model of the Stanford bunny is faintly visible in the center of the image.

Two versions of the Kinect sensor are available – one as strictly a peripheral for the Xbox 360 gaming console, and the other for use with computers running Microsoft Windows. The Kinect for Windows sensor has a configuration setting which allows it to read depth values between 0.4 m – 3.6 m, while the Xbox 360 version of the sensor is able to read values between 0.8 m and 4.0 m [35]. Methods in the provided Software Developer Kit (SDK) provide depth information in the form of a one-dimensional array of 16-bit values. Out of the 16 bits, 3 bits are reserved for the player index – a value used to determine which parts of the image are inhabited by a particular player. The remaining 13 bits contain the captured depth value for a particular pixel.

The Kinect sensor has a horizontal field of view of 57° and a vertical field of view of 43°. At a distance from the sensor of 1.0 m, both the horizontal and vertical resolution (spacing between captured points) is 1.6 mm. It has been shown that the accuracy of values coming from the Kinect sensor fall off in a non-linear fashion – meaning that depth values measured closer to the sensor are more accurate than those measured further away from the sensor [5]. In order to obtain accurate and high-resolution depth values, the Kinect sensor must be kept as close as possible to the object being scanned while still being within the valid scanning range. If the object being scanned is brought too close to the sensor, the sensor is unable to obtain valid readings. Appendix A shows the hardware design used in this work, which holds the Kinect sensor at a fixed position relative to the object being scanned.

The Kinect also contains a color camera, microphone array, accelerometer, and tilt sensor. Only the infrared sensor and tilt sensor are used in this work.

Chapter 3

The Voxel Octree Intersection Technique

As outlined in Chapter 1, the goal of this thesis is to create a system that uses low-cost, commercially available hardware to produce a 3D watertight model of a scanned object. With that general goal in mind, several restrictions were added:

- The technique must produce a watertight mesh, which should not require any additional processing prior to 3D printing.
- The technique must not use GPU processing.
- The technique must not use more than 2 GB of random access memory (RAM) when performing and processing a reasonable sized scan..

The first restriction stems from the recent surge in popularity of 3D printing and the descending prices of commercially available 3D printers. In order to 3D print an object from a scan, the model produced from the scanner must be watertight. Many of the currently available 3D scanning methods do not produce watertight meshes, and as a result, require additional processing to fill holes or gaps in the model prior to 3D printing. This restriction eliminates the need for additional post-scan processing, simplifying the overall process for the end-user.

The second restriction eliminates any reliance on GPU based processing. Although GPUs are becoming increasingly common, this restriction lowers the cost of the computer needed to run the algorithm by removing the need for a GPU. If running the algorithm in a cloud-computing environment, this restriction also allows for a greater selection of cloud computing hosts, as cloud-based GPU processing is still reasonably uncommon.

The restriction on the amount of RAM allows the technique to run on systems that have comparatively little system memory or that are running on older 32-bit operating systems. Also, the particular implementation of the technique created for this thesis makes use of the .Net based Microsoft XNA graphics library, which is a 32-bit library. Because of this, the final program must be compiled as a 32-bit binary which cannot access more than 2 GB of RAM. The XNA library was chosen because of its ease of use and robust vector and matrix functions. If an alternate implementation of the technique makes use of a different vector and matrix library, this restriction may be removed.

In order to accomplish the goal of creating a watertight 3D mesh with these particular restrictions in place, several different approaches were considered. The voxel octree intersection technique takes the following general approach:

1. An object is placed on a rotatable scanning platform.

2. A depth image (snapshot) is recorded using the Kinect sensor. This snapshot is processed and stored to disk.

3. The scanning platform is rotated by a configured amount and the previous step is repeated. This is done until a configured number of snapshots have been taken.

4. After all the necessary snapshots have been acquired, each snapshot is read back from disk. For each snapshot, valid adjacent depth pixels in the image are processed into a set of triangles. Each triangle is tested against a tree structure, where nodes of the tree are created and marked as being intersected by one or more triangles.

5. Once all triangles for all snapshots have been intersected into the tree, a collection of all leaf nodes is retrieved from the tree.

6. Each of these leaf nodes is used as a corner in a 3D cube. For each cube, an algorithm checks the values of the corners of the cube and creates geometry that approximates the intersecting surface.

7. The resulting geometry is written out to a stereolithography (STL) file.

After the scanner is initially calibrated and configured, the entire scanning process only requires a single interaction from a user to initiate a scan, which results in a watertight 3D model.

The voxel octree intersection technique makes use of a particular type of tree data structure to track regions of 3D space which are considered to be solid. An assumption is made about the depth data coming from the Kinect sensor, which is processed in such a way that adjacency information is used when entering the scanned data into the tree. The scanning process adds additional information to this tree with each subsequent scan. After all scanned data has been entered into the tree, the leaf nodes of the tree are extracted, and a set of geometry is created for each leaf node. The resulting geometry forms a hollow, watertight re-creation of the scanned object. This geometry is saved as an STL file – a file format commonly used by 3D printers.

By using a tree-based data structure, processing times and memory consumption are kept low. Redundant scanned data from subsequent scans is also handled without increasing memory consumption, and the final resolution of the resulting mesh can easily be scaled. This entire process is carried out on the CPU, so as to not require any GPU processing. Although the idea of using octrees for storing spatial data is not a new idea, the voxel octree intersection technique is unique. Alternate approaches rely on complex mathematical models, require higher quality scanned data, or are unable to generate watertight 3D meshes. The approach taken by the voxel octree intersection technique avoids these issues while still generating a watertight 3D mesh.

The design of the scanning hardware setup is found in Appendix A. A breakdown of costs for the hardware design is found in Appendix B.

3.1 Object Placement and Data Filtering

To begin a scan, the object being scanned is placed on the scanning platform. The specific placement of the object on the platform is inconsequential as all scanned data is transformed into a common coordinate space, however, the orientation of the object does matter. Any portions of the object which cannot be *seen* by the scanner will be assumed to be hollow. For example, if a solid cube is placed on the scanning platform and scanned, the top, left, right, front, and rear of the cube will be included in the generated model. The bottom of the cube will not be present in the generated model, as there is no way for the scanner to see the bottom of the cube without changing the cube's orientation relative to the scanning platform. The voxel octree intersection technique relies on the assumption that regions of 3D space are empty unless shown otherwise by data coming from the scanner. In the example of a scanned cube, the resulting cube will have a hollow interior, as there is no way for the scanner to know the contents of the interior of the cube.

Once the object has been placed on the scanning platform, the scanning process is initiated. In this process, a series of snapshots of depth data are recorded, with the platform being rotated between snapshots. As mentioned in Chapter 2, data coming from the Kinect sensor comes in the form of a one-dimensional array of 16-bit values. If the sensor is unable to determine the depth of an object at a given location, a value of 0 is returned. Figure 3.1 shows the result of converting this one-dimensional array into a 640x480 grayscale image. The model of the Stanford bunny is present in the center of the figure, and is sitting on the center of the scanning platform. Regions where the scanner is unable to obtain a valid depth values are marked in black. Lighter shades of gray represent depth values which are closer to the sensor. With the Kinect sensor, the infrared emitter and infrared sensor are offset by several centimeters. As a result, all objects in a scene have a shadowed edge, as the sensor is unable to determine valid depth values where no infrared light falls. This shadow is apparent in the figure along the left edge of the scanning platform and left edge of the Stanford bunny.

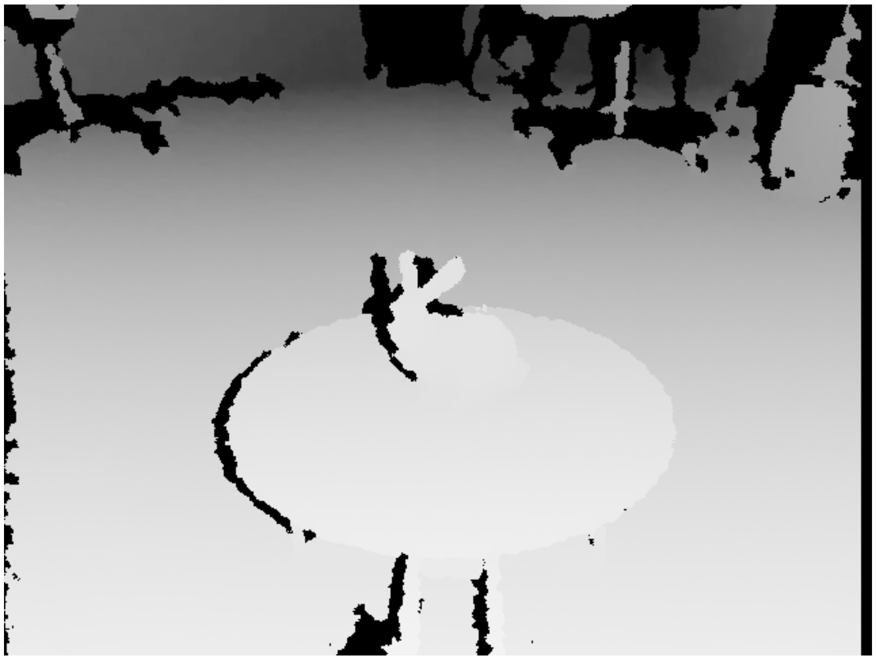


Figure 3.1: Greyscale depth values from the Kinect sensor.

What is not readily apparent in a single image is the amount of noise present in the captured depth values. Depending on the material type of the object being scanned and angle of the surface being contacted by the infrared light, the Kinect sensor may return inconsistent values. For materials that absorb infrared light or that are reflective, subsequent frames may alternate between returning valid and invalid values for a particular pixel.

Figure 3.2 shows the same scene as Figure 3.1, but regions where at least one invalid depth value in the past ten captured snapshots are colored in red. The luminance of the red

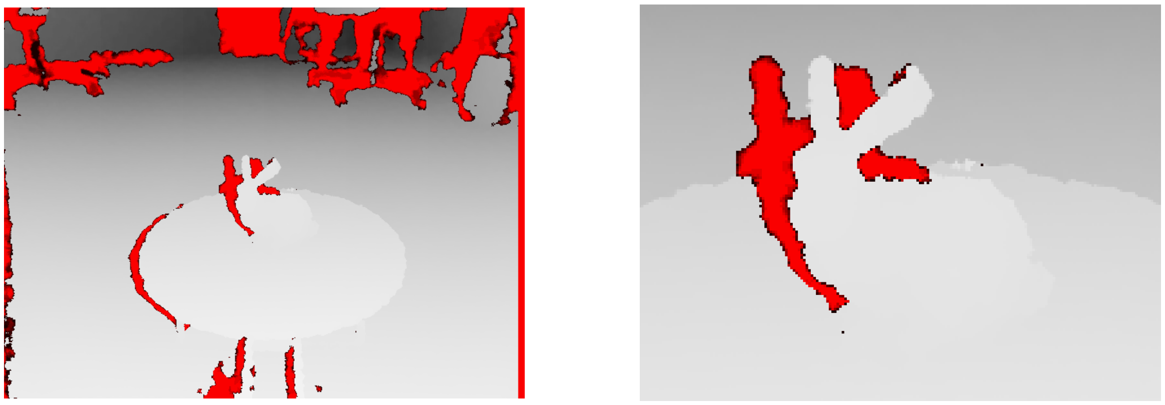


Figure 3.2: Invalid depth values over a ten frame average.

values correspond to the number of invalid values over a ten snapshot average – the brighter the red, the more invalid values for that particular pixel over the ten snapshot average. This is particularly visible as a soft (antialiased) edge along the border between the black and red regions. The image on the right shows center of the image on the left, but zoomed in by 500%.

Because of the potentially noisy nature of the data, it may be desirable to apply a filter to the depth data prior to saving it to disk. For example, a filter can be applied that will average valid depth values for a given pixel over ten snapshots. A comparison of using different data filters is shown in Chapter 4.

After the depth values are filtered, they are then converted into an array of world space coordinates. The software developer kit provided by Microsoft contains a method to convert the one dimensional array of values to a one dimensional array of world space coordinates relative to the position of the depth sensor. Invalid depth values, or those that have a value of 0, are converted to a world space coordinate of $(0,0,0)$.

Next, a series of transforms is applied to the array of world space coordinates. This is done to bring all captured snapshots into the same common coordinate space. For these transforms, the Y axis points up from the scanning platform, the Z axis points in the direction from the upright portion of the scanner to the scanning platform, and the X axis points to the right of the Kinect, as if positioned behind it.

These transforms are done by first applying a rotation on the X axis to counteract the tilt of the Kinect sensor. This rotation is necessary, as in the default scanning position, the Kinect sensor is tilted downard to best capture the contents of the scanning platform. Depending on the size of the object being scanned, the sensor tilt can be adjusted. To determine the value of this rotation, the tilt angle is obtained from the Kinect sensor via the Software Developer Kit. Next, a translation is applied based on the distance between the infrared sensor on the Kinect and center of the scanning platform. The value of this translation is obtained from a previously configured setting. It is critical that the distance between the center of the scanning platform and position of the infrared sensor on the Kinect is accurately measured for all three axes. Any discrepency between the configured and actual distance will create mis-alignments of the combined snapshots, resulting in an inaccurate scan. Finally, a rotation on the Y axis is applied based on the current rotation of the scanning platform. These transformations may be skipped if the point being operating on has a value of $(0,0,0)$.

3.2 Clipping and Storing Captured Data

The clipping process limits the amount of data used in later steps. It does this by setting any values that fall outside a specified range to zero. The minimum and maximum bounding distances are optionally configured by the user prior to starting a scan. This allows the later steps to only work with values that pertain to the object being scanned. For example, without setting the minimum bounding value on the Y axis, the resulting model would also include the scanning platform and the floor around the base of the scanner. As this is undesirable, the minimum bounding value on the Y axis is set so that any values below the height of the scanning platform are set to zero.

Once the transforms have been applied and the data has been clipped, the array containing the common coordinates space values is written to disk. It is important that this array contains the same number of elements as the original one-dimensional array. Writing the data to disk at this point allows the later portions of the process to be re-run with different parameters without needing to re-capture the scanned data.

The process of capturing depth data, filtering it, converting it to a common coordinate space, clipping it, and storing it to disk is repeated each time the scanning platform is rotated. The process advances to the next step after the scanning platform has made one full rotation.

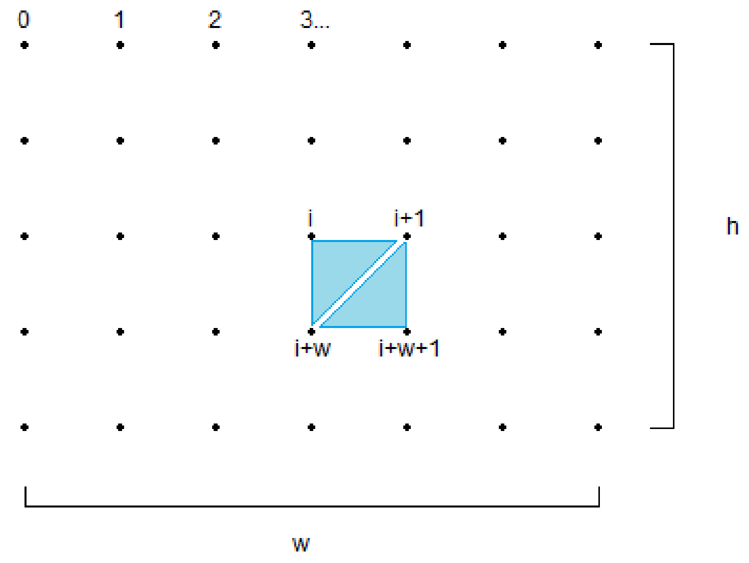


Figure 3.3: Two dimensional array of points used to create a series of triangles.

3.3 Triangulation and Thresholding

After the scanning platform has made one full rotation, each recorded snapshot is read back from disk and is used to generate a set of triangles. Because we store the world-space coordinates in the same order as the original one-dimensional array, we are able to make certain assumptions based on the adjacency of elements in the array. We can visualize the stored one-dimensional array in a similar manner to the originally captured depth image. By linking three adjacent elements, we can create a triangle in world-space coordinates. We can visualize the original depth image as a two-dimensional array of dimensions $w * h$, where w is the width of the original depth image and h is the height of the original depth image. Figure 3.3 shows a scaled-down version of this array.

For each element in the array, two triangles are created. Supposing that i is our current position in the array, the first triangle is formed using elements at i, $i+1$, and $i+w$. The second triangle is formed using elements at $i+1$, $i+w+1$, and $i+w$. There are several special cases where no triangles should be formed. These cases happen when:

- i mod *width* is equal to $(w-1)$ or when i/w is equal to $(h-1)$. When visualizing

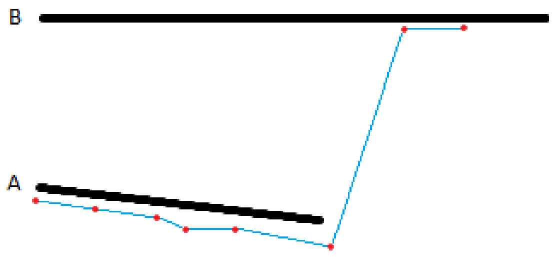


Figure 3.4: Consecutive depth points accidentally linking separate surfaces.

the data as a two dimensional array, no triangles should be generated when *i* is in the last row or last column. Any attempt at doing so will result in an invalid triangle or an attempt at reading outside the bounds of the array.

- Any of the three points of the triangle have a value of (0, 0, 0). This indicates that one or more of the values in the original array was an invalid depth value or was removed in the clipping process. Such values should not be used.

- The squared length of any side of the triangle exceeds a specified length, or threshold value, indicating that at least one of the points of the triangle is on a different surface.

The triangle edge length check is performed to prevent triangles being formed which would link unrelated geometry. Figure 3.4 depicts this situation, but in two dimensions. In the 2D case, adjacent points are linked with lines. In the figure, lines A and B represent the original surface being scanned. The series of points are recorded depth values, and the thin lines connecting the points are the lines created by the triangulation step. The first few points are connected as they follow along Surface A. The second-to-last line creates an accidental bridge, connecting Surface A to Surface B, while in reality no such connection exists. If this adjacent connection were included, it would create an inaccurate topology of the surface being scanned. For this reason a threshold value is applied. For the 3D case, if the squared length of any side of the triangle exceeds the threshold value, the triangle is discarded. Comparing the squared length to the threshold value saves performing a square

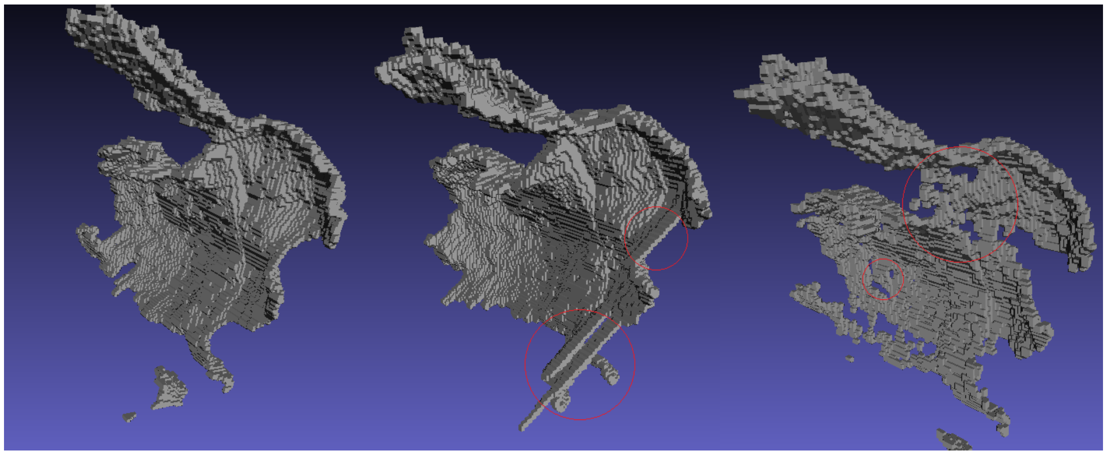


Figure 3.5: Similar scans with different threshold values applied.

root operation for each edge of every potential triangle. Although a single square root operation is inexpensive, this test must be performed for every edge of every potential triangle. From experimentation, a value of 0.0001 was found to be a suitable threshold value. This value may be configured by the user prior to scanning, depending on the nature of the object being scanned.

Figure 3.5 shows the effect of altering the threshold value. The scan on the left was performed with an appropriate threshold value of 0.0001. The scan in the middle had too large of a threshold value (0.001), resulting in bridges between unrelated geometry. The scan on the right used too small of a threshold value (0.00001), resulting in missing geometry in the final model. Particular problem areas have been circled in the figure.

The product of the triangulation step is a set of triangles which follow the approximate contour of the surface of the object being scanned. This set of triangles is then passed to the octree intersection step.

3.4 Octree Intersections

As previously mentioned, the voxel octree intersection technique uses an underlying tree structure to track regions of 3D space that are marked as being solid. These individual regions of space are known as voxels, or volumetric pixels. To track which voxels are marked as solid, triangles from the previous step are tested to see if they intersect any portion of the tree.

As this process is somewhat involved, it will first be described in 2D. To begin, a tree is created. In the 2D case, a quadtree is used – a tree whose root forms an axis-aligned square, and each child of a node occupies exactly one quarter of the area of a parent node. Each line from the previous step is run through a recursive process that begins at the root node. A line is tested against the root to see if any portion of the root is intersected by the line, or whether the line lies entirely within the root. If either case is found to be true, a set of temporary child nodes are created for the root node. These temporary child nodes are also tested to see if they intersect or contain the line. If they do, they are added as true children to the root node, and the process recurses on these true children. At each step of the process, prior to the intersection test, the size of the current node is tested to see if it is less than or equal to a configured size. If the node size is equal to or below the configured size, no additional tests are performed and a value is assigned to the node. The process is then repeated for each line created by the triangulation step. The end result is a tree whose nodes are intersected by or contain at least part of a line.

Figure 3.6 depicts this process happening in 2D. Figure A shows the root node and a line generated from the triangulation step, which is being intersection tested against the root node. Figure B shows the temporary child nodes created during the intersection test of the root. As the bottom two nodes do not contain or intersect the line, processing continues on only the top two nodes. The process recurses on these child nodes until child nodes of a minimum size are reached. At this point, in Figure F, the leaf nodes of the tree represent those regions of 2D space that were found to contain or intersect a line. These leaf nodes are assigned a value which is used in the next step.

The 3D case works in the same manner as the 2D case. Rather than using a quadtree, an octree is used – a tree in which each node can contain up to eight child nodes, and each child contains exactly one eighth the volume of the parent node. In this particular

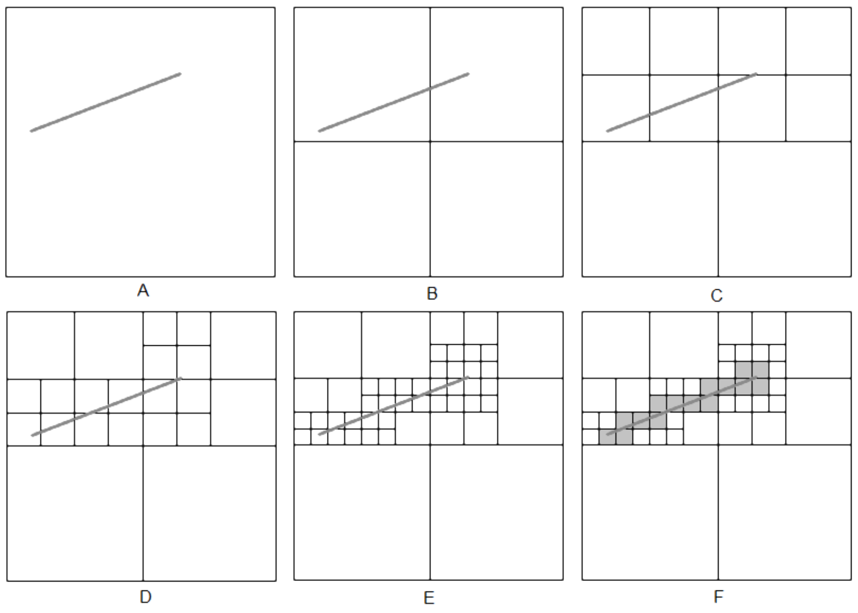


Figure 3.6: A quadtree being intersected by a line.

implementation, each node in the tree contains references to eight potential children. These references may be null, indicating that there is currently no data occupying that portion of the tree. As intersection tests are performed and temporary child nodes are added to the tree, these references are updated to point to the newly created temporary child nodes.

Like the 2D case, the 3D case defines each node by using an axis-aligned bounding box. This allows each node to be constructed using only two 3D points – one to track the minimum corner of the box and one to track the maximum corner of the box. The root of the tree is also restricted such that all sides of the root node are of equal length. This same restriction is applied to all nodes in the tree to simplify calculations when working nodes at various points in the algorithm and to save memory. As with the 2D case, the initial minimum and maximum corners of the root node can be determined beforehand by examining the scanned data or can be set to a predetermined value.

Once the root node has been created, each triangle from the triangulation step is intersection tested against the root of the tree. This is done using a very efficient triangle to axis-aligned bounding box intersection test [3]. This triangle to axis-aligned bounding box intersection test works by first testing the axis-aligned bounding box against the axis-aligned bounding box of the triangle. If these two intersect, it then attempts to find a separating axis between the bounding box and the triangle. If no separating axis can be found, the triangle and the axis-aligned bounding box are intersecting. This test will also give a positive result if the triangle is contained entirely with in the axis-aligned bounding box.

If the root node and triangle intersect, the same intersection test is performed for each child node of the root. If a child is null at this point, a temporary child node is created and the intersection test is performed against the temporary child. If the child node intersects the triangle, the null reference in the parent node is updated to point to this newly created temporary child node. If no intersection is detected, the temporary child node is discarded. This process of performing intersection tests and creating temporary child nodes is recursively continued until child nodes of or below a specified size are reached, or until no additional intersections are found. For example, if working in world units of metres, a minimum child node size of 0.001 would result in the process being recursively repeated until child nodes with a size equal to or below 1 mm are reached. The result is a tree where all leaf nodes in the tree are 1 mm in size or smaller, and any node in the tree is intersected by at least one triangle.

A pseudocode listing of the recursive intersect function is found in Algorithm 1. It operates on a leaf node and takes in as arguments a triangle to intersect and the size of a final leaf node. For each node in the tree, at most 9 intersection tests will need to be performed – one to determine if the parent node intersects, and eight tests to determine if each of the children intersect. If the parent node intersection test is moved outside the recursive function, the number of tests can be reduced to eight tests per node. If this is done, it becomes necessary to perform an initial intersection test to see if the triangle in question intersects the root node. When the recursive function reaches a leaf node, the node value is set. To save memory, each leaf node is assigned only a single value rather than a value

Algorithm 1 Recursive Intersection Test

```
function INTERSECT(Triangle t, float leafSize)
    if nodeSize > leafSize then
        Perform current node axis-aligned bounding box test against t
        if current node intersects t then
            for i = 0, i < 8 do
                Get child node at position i
                if node is null then Create temporary child node
                end if
                if Child node intersects t then
                    Assign child node to child node at position i
                    Child node -> INTERSECT(t, leafSize)
                end if
            end for
        end if
    else
        Set nodeValue = 1.0
    end if
end function
```

at each corner of the leaf. This value is used by the next step of the technique to generate the polygonal mesh. In this case, all leaf nodes have a value of 1.0. Chapter 4 discusses the effects of changing this value.

The particular nature of the octree allows the algorithm to quickly determine which leaf nodes are intersected. For example, if wanting to achieve a leaf node size of 1.0 mm or less and beginning with a root node that is 1.0 m in each dimension, it would take an octree with a maximum height of 11. Table 3.1 shows the relationship between the tree height and leaf node size for a tree with a root node size of 1.0 m.

This particular octree implementation also offers a sigificant advantage in terms of memory consumption over simply allocating a 3D array of voxels. For a 3D array of voxels to cover an area of 1 cubic meter and still have a resolution of 1 mm, it would be necessary to allocate an array with 1024^3 – or just over one billion elements. In order to achieve this while still remaining within the 2 GB memory restriction, each element would be restricted in size to approximately 2 bits or less. The tree structure allows us to cover a large volume with significant resolution and still allow a more detailed value for each region of

Table 3.1: Relationship between tree height and node size for a 1.0 m root node.

Tree height	Leaf size (in m)
0	1
1	0.5
2	0.25
3	0.125
4	0.0625
5	0.03125
6	0.015625
7	0.0078125
8	0.00390625
9	0.001953125
10	0.000976563

space while working within tight memory limitations. This assumes that not every region of space within the cubic region is scanned and considered solid.

The octree has a performance advantage over a 3D array of voxels when performing other operations on the tree. For example, when gathering all regions of space which have been marked as solid from a 3D array of voxels, it is necessary to visit all elements in the array. To perform this same operation on the octree, we recursively walk from the root through the non-null references to the child nodes. By doing so, we reach the leaf nodes of the tree, which are then returned. The functions to perform such operations are reasonably simple.

The octree also makes it possible to easily scale the resolution of the final generated model. As an input to the next step of the voxel octree intersection technique, a set of nodes is required. Under normal circumstances this is the set of all leaf nodes from the tree. If we desire to lower the resolution of the generated model, we can instead pass in a set of non-leaf nodes which are all of the same height. If h_L is the height of a leaf node in the tree, we can effectively halve the resolution of the final generated model by returning all leaf nodes of $h_L - 1$ height. Obtaining such a set of nodes from the tree is a trivial operation. Performing a similar operation on a 3D array of voxels requires sampling all of the voxel data.

3.5 Isosurface Generation

As mentioned in Chapter 2, one of the most common algorithms for extracting a polygonal surface from a voxel grid is the Marching Cubes algorithm. Although Marching Cubes is a popular choice, it suffers from ambiguious cases where the algorithm is unable to determine the correct intersection of the surface through the cube. If these ambiguous cases are not handled correctly, the resulting polygonal mesh will not be watertight. This ambiguity can be resolved by either taking additional samples within the cube, or by consulting additional lookup tables.

To avoid the need for re-sampling the scanned object or requiring additional lookup tables, a variant of the Marching Cubes algorithm is used. This algorithm, known as Marching Tetrahedrons or Marching Tetrahedra, divides a cube into six tetrahedrons. Values at the four corners of each tetrahedron are obtained and checked against a lookup table to determine which geometry should be created. The Marching Tetrahedrons algorithm does not suffer from the same ambiguity as Marching Cubes, and as a result, always produces a watertight polygonal mesh. The downside to the Marching Tetrahedrons algorithm is that it may produce a final mesh containing a larger number of triangles than results obtained from the Marching Cubes algorithm. Because our end goal is a watertight mesh for the purpose of 3D printing or CNC machining, the number of triangles in the final mesh is not a large concern.

Because we are working with a tree structure rather than a regularly-spaced voxel grid, it is necessary to modify the Marching Tetrahedrons algorithm to work with this scenario. These modifications are also necessary because each leaf in the tree only stores a single value rather than eight values – one at each corner of the leaf. Although Marching Tetrahedrons works only a single tetrahedron at a time, it is still necessary to know all eight corner values, as we still need to process all six tetrahedrons for each cube.

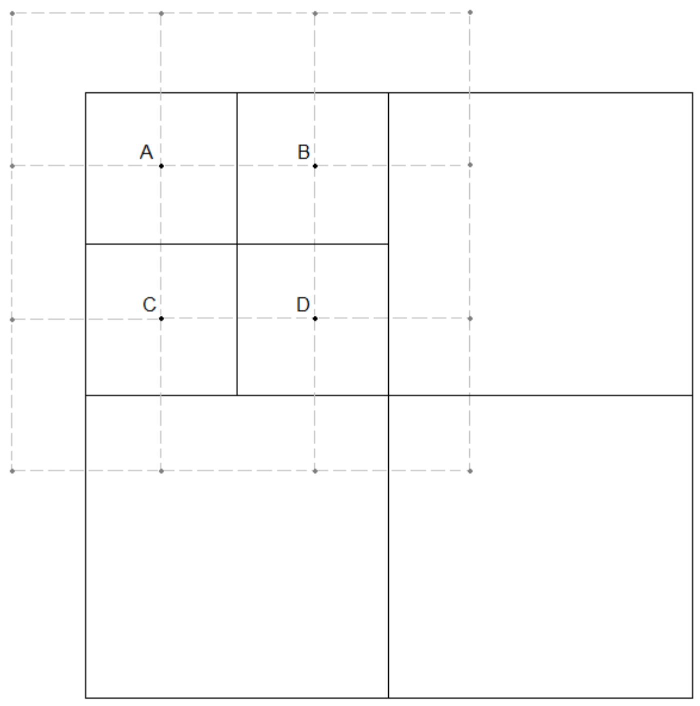


Figure 3.7: Gathering values for isosurface generation from neighboring nodes.

To get around the issue of only having a single value per leaf, we use values from adjacent leaves and create a pseudo-voxel – a temporary voxel whose eight corners lie in the center of eight different but adjacent nodes. For each node that is passed into this step of the voxel octree intersection technique, we create eight pseudo-voxels. We then run the Marching Tetrahedrons algorithm on these pseudo-voxels to obtain our final geometry.

Figure 3.7 depicts this process in 2D. In the figure, vertices A, B, C, and D are the centers of four leaf nodes in the tree. For each leaf node in the tree, four pseudo-voxels are created which use the center of the leaf node as one corner of the pseudo-voxel. For example, for leaf node A, the pseudo-voxels, as represented by dashed lines, appear to the upper right, upper left, lower right, and lower left of A. A recursive function is used to obtain the value for each corner of the pseudo-voxel. If the corner of a pseudo-voxel lands outside the tree or is not contained within a leaf node, a value of 0 is assigned to that corner. When all the corner values for a pseudo-voxel have been obtained, the Marching

Tetrahedrons algorithm is run and the resulting geometry is added to the final mesh.

To avoid creating duplicate geometry, the centers of each pseudo-voxel are tracked in a hash table. Prior to creating a pseudo-voxel, the hash table is checked to see if the center of that potential pseudo-voxel already exists in the hash table. If it does, the process moves on to work with the next pseudo-voxel. This prevents creating duplicate pseudo-voxels, which in turn prevents creating duplicate geometry.

After the Marching Tetrahedrons algorithm has been run on all pseudo-voxels, the resulting geometry is written to disk in the format of a Stereolithography (STL) file. This file format is a commonly used file format for 3D printers, modeling software, and even graph drawing software [2].

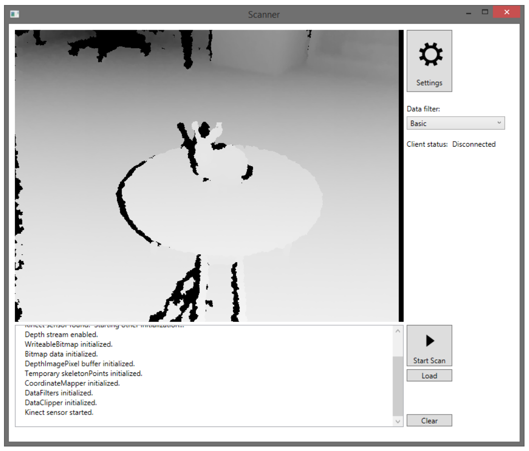
Chapter 4

Implementation and Results

4.1 Implementation

An implementation of the voxel octree intersection technique, along with the scanning hardware described in Appendix A was created for this work. The software created consists of several components – the main application, the client application which runs on the client computer, and a separate application for viewing captured data and configuring the scanner. Figure 4.1 shows the main application and calibration application. All software was written in C#, which runs on the .Net framework. In order to run the C# client application on the Rasberry Pi, which uses a Linux based operating system, it is necessary to install Mono on the Raspberry Pi. Mono provides an implementation of the .Net framework for Linux based operating systems.

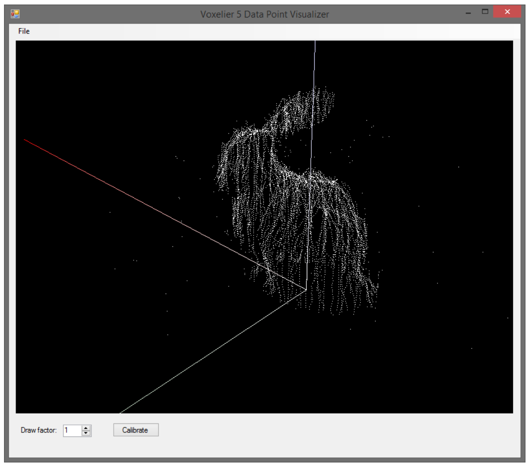
The main application and client application on the Raspberry Pi communicate with each other via the open-source Lidgren networking library. After the main application is started, the client application is also started, and connects over the network to the main application. When the main application requires the scanning platform to turn a given amount, it sends a network message to the connected client indicating how far the platform should be rotated. When the platform is finished rotating, the client application sends a message back to the main application indicating that the platform rotation is complete, and that scanning can continue.



Data filter: Basic

Client status: Disconnected

```
Kinect sensor found. Starting other initialization...
Depth stream enabled.
WriteableBitmap initialized.
Bitmap data initialized.
DepthImagePixel buffer initialized.
Temporary skeletonPoints initialized.
CoordinateMapper initialized.
DataFilters initialized.
DataClipper initialized.
Kinect sensor started.
```



Draw factor: 1

Figure 4.1: The main scanner application and data viewing application.

4.1.1 Configuring the Scanner and Performing a Scan

Prior to beginning a scan, the scanning system must be properly calibrated and configured. Values for the triangulation threshold and the number of degrees per rotation are set and the elevation angle of the Kinect is read from the sensor. When configuring the scanner for first use, the distance between the center of the scanning platform and the center of the Kinect's infrared sensor is measured. This is done by placing a small object in the center of the scanning platform. The calibration application is opened, and the user adjusts the X, Y, and Z values so that the center of the object falls onto the Y axis, and the base of the object is aligned with the X/Z plane. These X, Y, and Z values are recorded by the user and entered into the settings of the main application. Values defining the minimum and maximum clipping space and final scanning resolution may also be configured at this stage.

It is critical that the distance between the center of the scanning platform and the Kinect be measured as accurately as possible. If off even by a few mm, the mis-calibration will result in undesirable artifacts in the resulting scan. Figure 4.2 shows the difference between a properly and improperly calibrated scan. Both scans show the underside of a scanned Stanford bunny model. The scan on the left is poorly calibrated while the scan on the right is properly calibrated. With the poorly calibrated scan, it is possible to distinguish the different snapshots of data, and the mis-alignment results in visible spurs along the contour

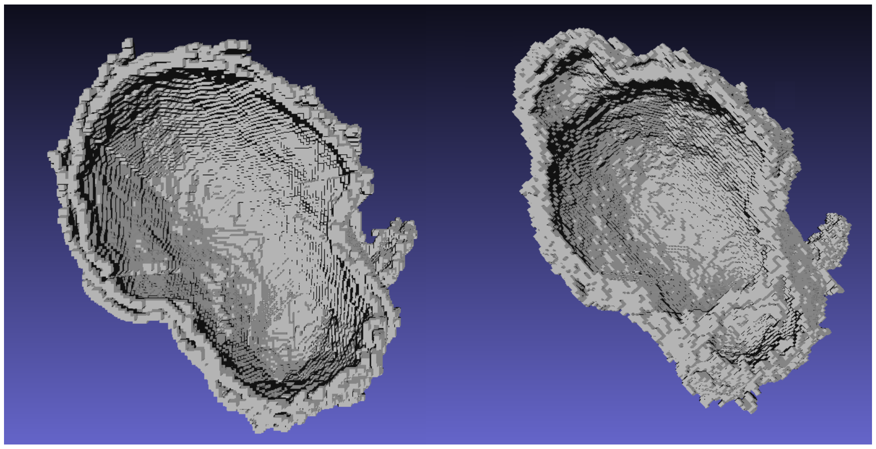


Figure 4.2: A poorly calibrated and a well calibrated scan.

of the object. If scanning artifacts are present in a scan, the X, Y, and Z distance values should be adjusted and the object re-scanned.

To set up the scanner prior to performing a scan the main application is first started, then the client application is started on the Raspberry Pi. The user then places an object on the scanning platform, and clicks the *Start Scan* button. When the scan is complete, the results are placed in the *Scan.stl* file located in the same directory as the main application executable.

4.2 Analysis of Introduced Error

There are several sources of error to consider when using the combination of the described scanning hardware and voxel octree intersection technique. These sources of error include:

- Error in depth values recorded by the sensor.
- Error as a result of poor scanner calibration.
- Error introduced by the voxel octree intersection technique.

The first source of error to consider is error introduced by the Kinect sensor in the captured depth values. Research has shown that values coming from the Kinect sensor tend to oscillate between several different states [25], and that the accuracy of values rapidly falls off with distance [5]. Despite this, and the Kinect's limited depth resolution, it is possible to capture a set of values that reasonably represent the surface of the object being scanned. The quality of the scanned data can be improved by filtering the data prior to passing it in to the voxel octree intersection technique, as discussed in section 4.3.

The particular hardware setup used may introduce error as a result of mis-calibration. For example, if the scanning platform and Kinect platform are not aligned in the same plane on the X/Z axis, seams will be visible between captured snapshots when the mis-aligned scans are combined. Also, if the various components of the scanner are not kept in a fixed alignment with each other during the scanning process, the resulting data will not be aligned after being combined into the common coordinate space. These sources of error can be mitigated by ensuring that the scanner is properly calibrated prior to performing a scan, and ensuring that the scanner is not moved during a scan. A different hardware configuration may also help to reduce or eliminate such errors.

The voxel octree intersection technique will introduce some error into the final geometry of the generated model. This error may be a result of:

- The threshold value being set too low or too high.
- Intersection tests marking entire leaf nodes as being intersected when the underlying surface only intersects a small portion of the leaf node.
- Aliasing as a result of regularly spaced leaf nodes.
- An inability to generate especially thin features because of a minimum resulting thickness in the final model.

Figure 4.3 depicts a simple 2D object being scanned from a single side, and the results of running the voxel octree intersection technique on the scan. In the figure on the left, the thick, light grey colored line represents the original surface being scanned. The small

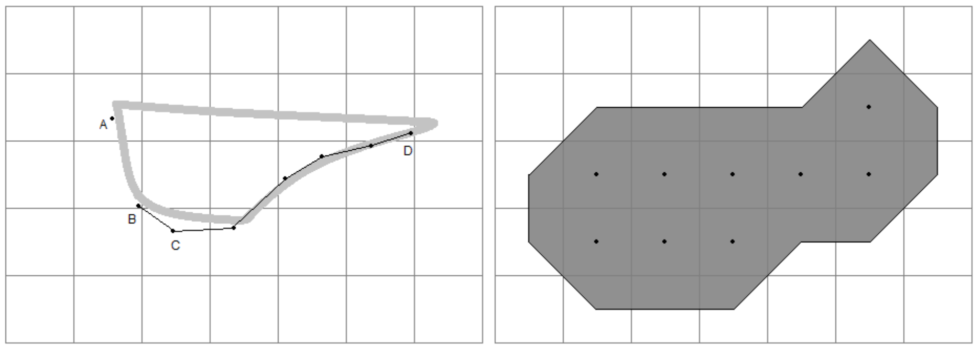


Figure 4.3: Introduced error in two dimensions.

dark points represent individual points captured by the scanner, and the thin lines linking them together represent the lines created from the triangulation step of the technique. In the figure on the right, the thin black lines represent the re-created surface which encapsulates the grey area, creating a watertight shape. Although the voxel octree intersection technique is based on a tree, regularly spaced leaf nodes in the diagram are shown as a grid for the sake of clarity.

The lack of a line going from Point A to Point B in the figure is a result of the threshold value being applied. Despite the underlying surface being present, if the length of the line between the two points is greater than the threshold value, the line is not intersected against the tree. As a result, the leaf node containing Point A is not marked as solid, despite the data from the scanner potentially saying otherwise. In this sense, the data coming from the scanner is somewhat ambiguous – there is no way of determining whether or not two adjacent scanned points are actually connected on the real object without taking further depth samples between the two points. With the voxel octree intersection technique, we make the assumption that adjacent points are connected – up until the length between them exceeds the threshold value. By doing this, we may be rejecting valid situations where two adjacent points are indeed connected, and we may also be linking adjacent points which are portions of two separate and distinct surfaces. By making such an assumption and making a decision based on a configured threshold value, some error is introduced into the final model. An exact measurement of this amount of error is dependent on the particular data being obtained, and the threshold value used.

In Figure 4.3 we also see that nodes surrounding Point B have been marked as solid. This is because the line connecting from Point B to Point C passes through the corners of the four nodes nearest to A. Although the line from B to C passes through an infinitely small portion of these four nodes, the entire nodes are marked as solid. This does not necessarily give an accurate depiction of the underlying surface. As a result, the final model is considerably larger in that region.

In the same figure, Point C is a result of a scanning inaccuracy. The depth points captured by the Kinect sensor are limited in accuracy. As a result, the generated connecting lines may not follow the exact surface of the object being scanned. This can be somewhat mitigated by properly filtering the data during scanning, and discarding ambiguous or invalid data. Despite this, any inaccuracies in the filtered data will still have an effect on the generated model. Although not a particular problem with the voxel octree intersection technique, this source of error is visible in some of the results displayed later in this chapter.

In Figure 4.3, the underlying surface extends to the right beyond Point D, but there are no captured points to the right of Point D. This may be a result of the next point being eliminated due to thresholding, or an invalid reading because of the angle of the surface of the object being scanned. The end result is that the generated model is missing some features of the original scanned object.

The resulting model may also lack fine surface features of the original object. This limitation can be explained in terms of the Nyquist rate. In order to be able to re-create details of a particular surface, the surface must be sampled at twice the resolution of the desired feature size [27]. For example, at a distance of 1.0 m, the horizontal spacing between captured depth points from the Kinect sensor is 1.6 mm. Because of the Nyquist rate, the finest horizontal feature size that we can hope to obtain at a distance of 1.0 m is 3.2 mm in size.

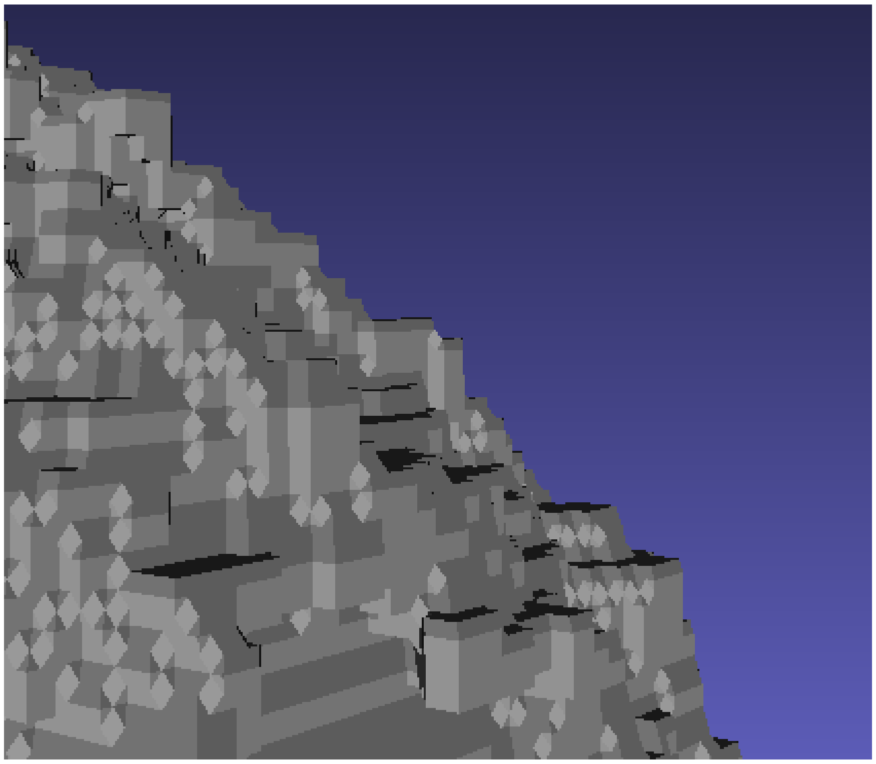


Figure 4.4: A portion of a scan showing aliasing.

4.2.1 Aliasing

One of the consequences of using a grid based approach is the introduction of aliasing – the visible staircase effect that occurs when a continuous variable is mapped to a discrete grid. Although the voxel octree intersection method is based on a tree, it suffers from the same aliasing artifacts as a grid based approach because of two reasons:

1. It uses regularly spaced leaf nodes to generate the final geometry.

2. It uses Marching Tetrahedrons with input values that are either 0.0 or 1.0.

Aliasing is especially apparent along the outer edges of the generated mesh. Figure 4.4 shows an example of aliasing on the edge of a scanned model.

With the voxel octree intersection technique, all leaf nodes in the tree are of equal size. When we are generating the final geometry, we consider only the leaf nodes and their adjacent neighbors. This is essentially the same as operating on a 3D grid, the exception being how we access values in adjacent leaves. As a result, we suffer from the same sort of aliasing that is apparent when working with a grid-based approach.

When working with the Marching Cubes or Marching Tetrahedrons algorithms, the specific values at each corner of the cube or tetrahedron may be considered when creating the final geometry. If we consider the corner values, we can interpolate the position of the generated geometry. For each edge in the cube or tetrahedron, we interpolate the values between two corners, and place the generated geometry at the interpolated position along the edge. This results in a considerably better fitted surface. Figure 4.5 shows two shapes constructed with the Marching Squares algorithm. The figure on the left shows a shape constructed by the Marching Squares algorithm where all edges are intersected in the exact mid-point between two corners. The figure on the right shows a better fitted surface as a result of moving the edge intersection points based on interpolating between corner values.

Because we only store a single value for each leaf node in the tree, and that value is either 1.0 or 0.0, the geometry created from the Marching Tetrahedrons algorithm is often blocky looking. As we are using cube shaped nodes in the tree to designate areas of 3D space that are occupied, the border areas between regions of solid and non-solid space occur on right angles. Without taking into consideration values from surrounding nodes, there is no way to interpolate between these regions in a way that does not produce geometry on either 90° or 45° angles. If we use a node value other than 1.0 for nodes marked as solid, we still have this same problem, as we are still interpolating between zero and non-zero values.

Another way of reducing aliasing is to decrease the minimum size of the leaf nodes in the tree. This is the equivalent of increasing the resolution of the underying grid when using a grid-based approach. Results of changing the resolution are discussed in section 4.4. The effects of aliasing may also be hidden by applying a surface smoothing operation such as the Surface Nets algorithm to the generated mesh. This would give the resulting mesh a much smoother appearance, but is beyond the scope of this work at this time.

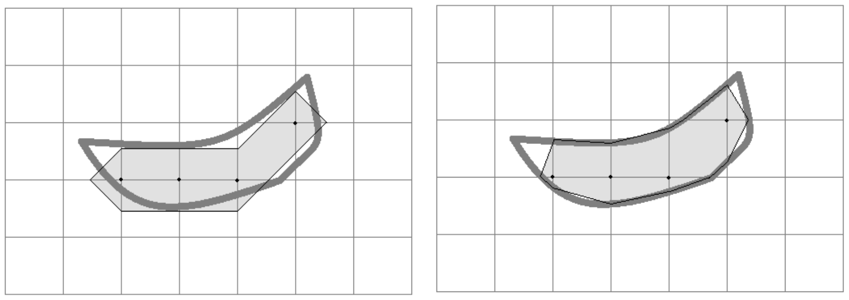


Figure 4.5: Marching squares without and with interpolation.

4.2.2 Worst Case and Minimum Thickness

To calculate the difference between the actual surface of the object being scanned and the re-created surface, we consider the worst case. In this case, a single pseudo-voxel is considered which has only a single corner marked as solid. The Marching Tetrahedrons algorithm then divides this cube into six tetrahedrons. Because we only set corner values to 1.0 or 0, the calculated mid-points between the corners of each tetrahedron will always be exactly half way between the solid and non-solid corners. The greatest distance from the solid corner to any generated geometry is half the distance between opposing corners on the cube. This distance is the euclidean distance between two points, and can be calculated as $\sqrt{(w/2)^2+(h/2)^2+(d/2)^2}$, where w is the width of the leaf node, h is the height of the leaf node, and d is the depth of the leaf node. As all side lengths of the cube are equal, this can be simplified to $\sqrt{3(s/2)^2}$, where s is the length of any side of the cube.

The voxel octree intersection technique will always create a mesh of a minimum thickness, where the thickness of the mesh is the distance between two opposing outer-facing surfaces. Figure 4.6 depicts the thinnest possible generated geometry in 2D. Leaf nodes marked as solid are indicated in grey. Pseudo-voxels are represented by dashed lines, and leaf node centers are indicated by black points. In this case, a line has been intersected against the leaf nodes, and a single series of leaf nodes have been marked as solid. Because the values of the leaf nodes are set to 1.0 when they are intersected, the Marching Tetrahedrons algorithm will create geometry on the mid-point between the solid and non-

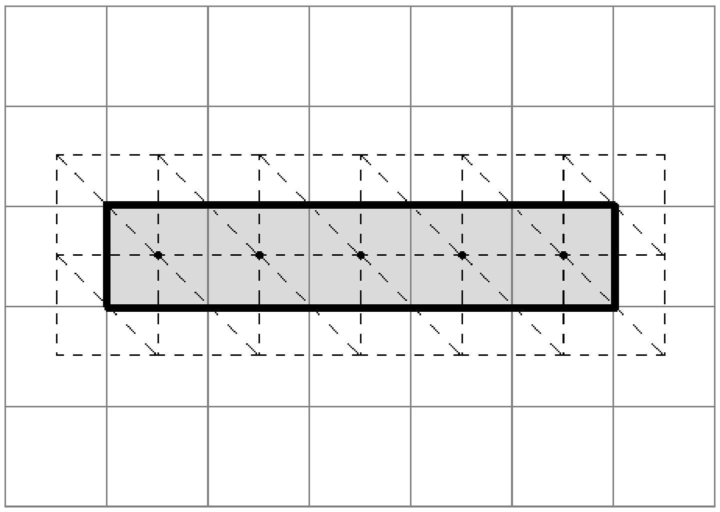


Figure 4.6: A 2D mesh of minimum thickness created by Marching Tetrahedrons.

solid corner of the pseudo-voxel. This occurs with the pseudo-voxels on both sides of the solid leaf nodes, resulting in a surface that is exactly the thickness of one leaf node. This same principle applies in 3D – any captured feature will be represented with a minimum thickness of one leaf node.

4.3 Comparison of Data Filters

Three selectable filters were created to process the depth data coming from the Kinect sensor. These three filters are:

- Basic: Captures a single frame of data from the sensor without making any changes to the data.
- Average: Captures multiple frames of data, and uses the average of these frames. Invalid depth values are ignored, and average depth values are calculated using remaining valid values.
- Average-Discard: Captures multiple frames of data and uses the average of these frames. For any particular depth pixel, if any of the frames contain invalid values for that pixel, all values for that pixel are discarded.

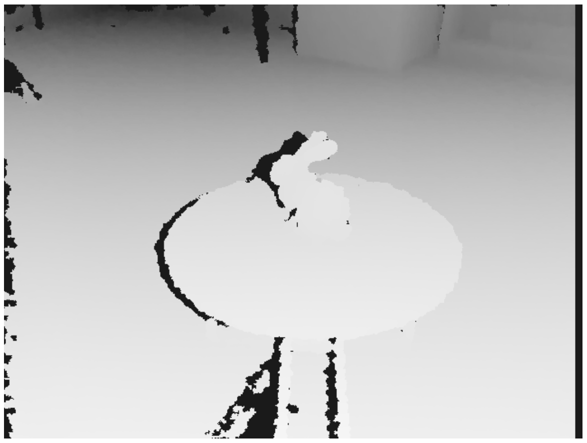


Figure 4.7: Data from the Basic data filter converted to a grayscale image.

The intent of the Average and Average-Discard data filters is to provide more accurate depth readings over the surface and edges of the object being scanned. By taking an average depth reading, the Average data filter removes oscilations between different depth values, resulting in depth values that are more consistent with the surface of the object being scanned. The Average-Discard data filter takes this one step further by eliminating any potentially questionable depth values. Figure 4.7 shows values coming from the Basic data filter converted to a grayscale image.

Data coming from the Average and Average-Discard data filters may appear to be very similar to data coming from the Basic data filter, but a closer inspection reveals small but significant differences. Figures 4.8, 4.9 and 4.10 highlight these differences. Figure 4.8 shows the differences between the Basic and Average data filter over the course of ten frames. These differences are highlighted in red, and show the oscillation in depth values, especially as the distance from the sensor increases. This happens due to the quality of depth data falling off with increasing distance on the Kinect sensor. Figure 4.9 shows the differences between the Average and Average-Discard data filters over the course of ten frames. Figure 4.10 also shows the differences between the Average and Average-Discard data filters, but over the course of 250 frames. By averaging over a larger number of frames, a greater number of questionable depth values are rejected. Note that the regions marked in red in Figure 4.10 are somewhat thicker than in Figure 4.9.

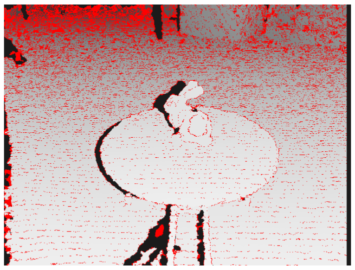


Figure 4.8: (a)

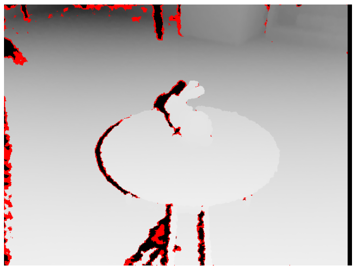


Figure 4.9: (b)

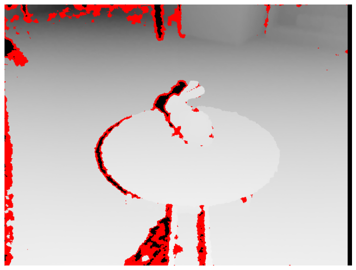


Figure 4.10: (c)

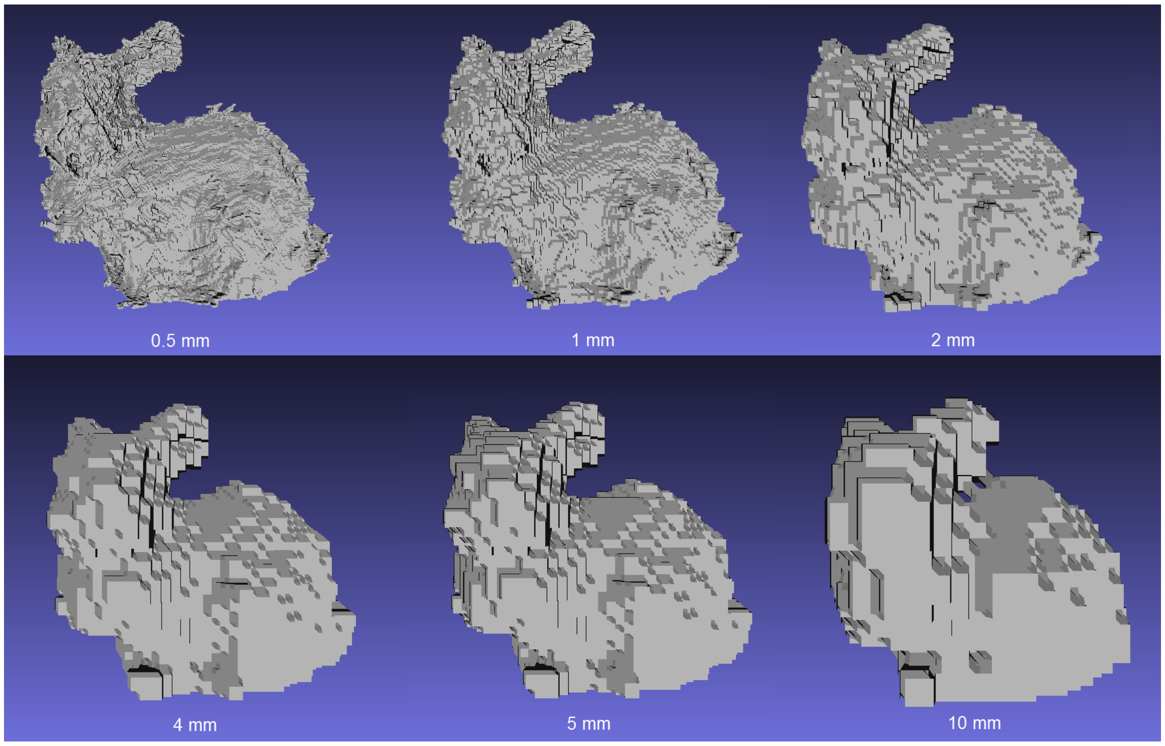


Figure 4.11: Scans completed using various leaf node sizes.

4.4 Comparison of Resolutions

The detail of the final model is heavily influenced by the size of the leaf nodes chosen during the intersection process. By keeping the size of the root node constant and changing the size of the leaf nodes of the tree prior to begining the intersection tests, the overall effect is that of changing the resolution of the final model. This is demonstrated in Figure 4.11.

In the figure, the original root node was given the dimensions of 1.0 in the X, Y, and Z axis, and eight data snapshots were captured during the scan. The values listed below each scan are the minimum leaf size in the tree. By decreasing the minimum leaf size, the effective resolution is increased, allowing the final model to have considerably more detail. This is especially visible in the scan completed with the 0.5 mm leaf node size – small features and fine noise are visible. As the minimum leaf size is increased, the amount of detail and noise visible in the model decreases, as does the processing time and final polygon count of the model. Table 4.1 shows the processing time, peak memory consumption, and final polygon count of each processed scan.

Table 4.1: Processing times, memory consumption and polygon count with increasing leaf size.

Minimum leaf size (mm)	Processing time (s)	Memory (MB)	Polygon count
0.5	117.1	546.5	5142120
1	34.9	169.8	922916
2	22.7	94.7	205196
4	18.5	88.6	51408
5	18.5	84.5	51408
10	15.9	70.4	13712

An interesting effect is visible when comparing the polygon counts of the scans completed with a 4 mm and 5 mm minimum leaf node size. Although the minimum leaf node size has changed, the effective resolution and final polygon counts remain identical. This is visible in both Figure 4.11 and in the table. This is due to a combination of the original root size and depth of the tree that will achieve such a minimum leaf node size. For example, if a tree has a root node of size 10.0, and a desired minimum leaf size of 2.0, the tree must reach a depth of 4. At this depth, the actual leaf size is 1.25. At each depth in the tree, the child node size is half that of the parent node size. This means that any value chosen for the minimum leaf node size between 1.25 and 2.5 will result in a tree of the same depth. In the case of our 4 mm and 5 mm scans, both values fell within the range of values that gave the same tree depth, essentially generating the same model. The difference in memory consumption can be attributed to garbage collection in the .Net framework running at different times.

4.5 Analysis of Space and Time Efficiency

As shown in Table 4.1, memory consumption and final polygon counts increase exponentially in relation to the depth of the tree. In the worst case, for a given depth of a tree d, the number of leaf nodes in the tree is $(2^{d-1})^3$, where a tree of $d = 1$ has a single node. The maximum number of nodes in a tree d, is the sum of the maximum number of child nodes for all depths of the tree up to d. Such a worst case is incredibly unlikely and even physically impossible when working with trees with a non-trivial depth. The voxel octree intersection technique makes the assumption that space is empty unless otherwise indicated by data coming from the scanner. In order to achieve the worst case, the data would need to indicate that all regions of a given space are solid. For this to happen, either invalid data would need to be passed in from the scanner or the type of scanner used would need be able to acquire depth measurements through solid material. Such data would indicate that all space within the root node is solid, in which case using the voxel octree intersection technique would give less than ideal results.

The total time necessary to process a scan depends on several factors: the number of data points gathered from the scanner, clipping values on all three axes, the number of triangles generated in the triangulation step, and the minimum leaf size for the tree. A general formula for the time to complete a scan is:

$$(S*n)+(i*t)+M \tag{4.1}$$

In this formula, S is the time it takes to capture and filter a single snapshot, and n is the number of snapshots in the total scan. The next pair of variables, i and t, are the time it takes to intersect a single triangle and the total number of triangles to be intersected. The specific value for i is dependent on the root node size, minimum leaf node size, and current contents of the tree. The value for t is dependent on the number of triangles generated by the captured data. The last variable, M, is the time it takes to generate the pseudo-voxels and perform the Marching Tetrahedrons algorithm on them. This is also heavily dependent on the contents of the tree. A general estimation can be created by noting the time taken by previous similar scans, and substituting in specific values as desired.

From experimentation, it was found that one significant performance limiting factor is the number of triangle intersection tests performed for a given triangle. Early experiments altered the structure of the tree so that each node could have up to 3x3x3 children. This flattened the tree, but also greatly increased the number of intersection tests necessary when processing a single triangle. Although each intersection test takes a fraction of a millisecond to perform on decent hardware, the drastic increase in the number of intersection tests resulted in remarkably worse performance.

Another major performance limiting factor is the time needed to generate geometry for each pseudo-voxel. Prior to implementing a caching system, geometry was generated for all neighbors of a given leaf node. This resulted in duplicate geometry and excessively large STL files. A simple caching system was implemented to track the centers of each pseudo-voxel which had already been processed. This significantly increased performance, but still takes a large portion of the overall time when generating a scanned model. For example, when creating a scan from eight snapshots with a minimum leaf size of 0.001, the total time to generate a model was 35.0 seconds, with 8.1 of those seconds being spent processing the pseudo-voxels and writing the resulting file to disk.

4.6 Final Print Comparison

Figure 4.12 shows two models and the results of scanning those models using the voxel octree intersection technique and hardware as described in Appendix A. The original models are on the left while their scanned counterparts are on the right. A 30 cm ruler is included in the figure for scale. All models were printed on a MakerBot Replicator 2 3D printer. The Stanford Bunny model was scanned using the Average data filter with a 2 mm minimum leaf node size while the dragon model was scanned using the Average-Discard data filter and a 1 mm minimum leaf node size. The top portion of the dragon model was not printed

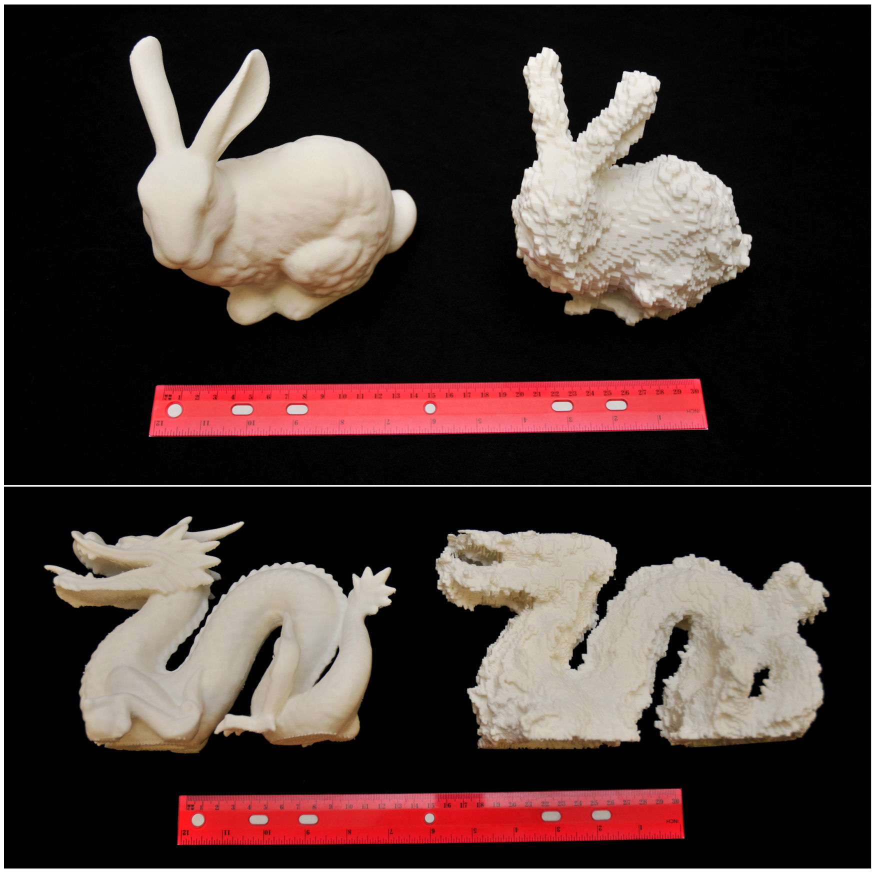


Figure 4.12: Original models and their scanned equivalents.

due to the printer running out of filament near the end of the print job. No modifications were made to the scanned models prior to their printing. As visible in the figure, both models bear a reasonable likeness to their original counterparts.

Details from the Stanford Bunny model are shown in Figure 4.13. A small protrusion is visible on the side of the Stanford Bunny scan, as indicated by the red arrow. Such a protrusion is the result of a single errant depth value. The hollow interior of the model is also visible along the bottom of the figure.

Despite the reasonable resemblance to the original models, the scanned models exhibit a very rough surface which suffers from severe aliasing. As a result, the usefulness of such models is somewhat limited when compared to meshes produced by alternative methods.

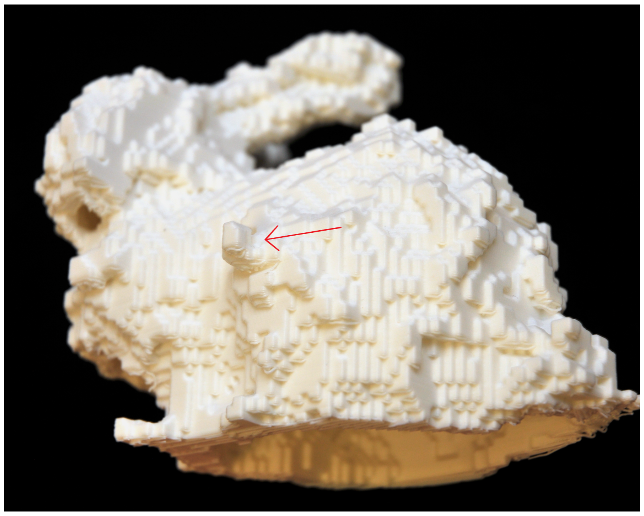


Figure 4.13: A protrusion on a scanned model.

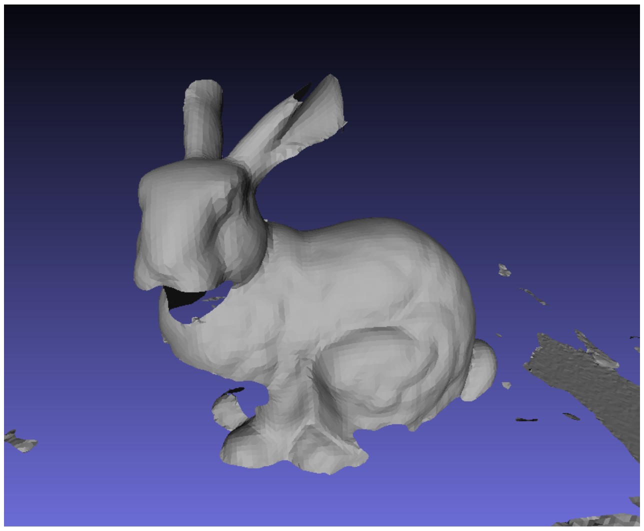


Figure 4.14: A partial scan of the Stanford Bunny.

4.7 Comparison to a Commercially Available Scanner

Figure 4.14 shows a partial scan of the Stanford Bunny model taken by a Creaform Go!SCAN 3D scanner, which is capable of a 0.5 mm resolution. The Go!SCAN scanner, which is a patterned light scanner, produces noticeably smoother looking results when compared to the voxel octree intersection technique. It also captures at near real-time rates. Although the Go!SCAN is neither the least nor most expensive 3D scanner, its $15,000 price is significantly higher than that of the hardware setup described in Appendix B.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

The voxel octree intersection method provides a way of using data from a commercially available low-cost sensor to generate a watertight 3D mesh. First, data coming from the sensor is filtered to ensure that following steps are working with valid data. Next, adjacency information is extracted from the filtered data and is used to generate a set of triangles. These triangles are then efficiently tested for intersections against the contents of a specially designed tree data structure. The underlying tree data structure allows the algorithm to mark regions of 3D space as being occupied in a memory efficient manner. By taking the leaf nodes from the tree and finding their adjacent neighbors, we can use the Marching Tetrahedrons algorithm to generate a set of watertight geometry. We can also easily scale the detail in the resulting mesh by using all nodes from the tree which are of a given height. The resulting geometry is watertight, and is immediately suitable for 3D printing or CNC machining without the need for any additional processing.

Although the voxel octree intersection method produces watertight results while using little memory, resulting meshes are not smooth, and suffer from considerable aliasing. Resulting meshes also have high polygon counts, and thus, are generally unsuitable for real-time rendering when compared to meshes generated by alternative methods. Memory consumption is also heavily tied to resolution – doubling the resolution requires eight times the amount of memory.

Despite these drawbacks, the voxel octree intersection method provides a different approach to generating meshes which are guaranteed to be watertight, using commercially available low-cost hardware.

5.2 Future Work

Although the voxel octree intersection technique provides satisfactory results as shown in Chapter 4, there is still room for improvement. The following areas of research may prove to be valuable in increasing the performance and accuracy of the technique.

5.2.1 Improved Data Filters

Depth values coming from the Kinect sensor, particularly along acute object edges or boundaries between distant objects have been shown to be quite noisy. For this reason, some basic filters were developed and applied to help improve the quality of the data. Although these filters did help, there is still room for improvement in increasing the general qualty of the captured depth data. Rather than relying on a simple filter, a filter could use a predictive model to get an even better estimate of values along object boundaries. Predictive methods such as Kalman filtering have been used in analyzing depth values [32]. A predictive filter would be able to assist in obtaining depth values along object edges. When re-constructing objects that contain a significant number of distinct edges, a predictive filter could greatly improve the quality of the resulting generated mesh.

5.2.2 Alternate Scanning Hardware and Cloud Based Processing

Even at its highest resolution, the quality of depth data provided by a Kinect depth sensor is questionable when scanning small objects. In addition to having a relatively low spatial resolution at a distance of 1 m, the depth data suffers from considerable noise. The next generation of the Kinect depth sensor was revealed with the Xbox One gaming console in May 2013, with plans to release a software developer kit for Windows announced in March 2014 [37]. The newer Kinect sensor uses a time-of-flight based sensor which uses timed pulses of infrared light to determine distances, and is able to capture depth images at a higher 1920x1080 resolution. It also features a wider field of view, and is able to acquire valid depth values from objects that are closer to the sensor than the previous version sensor [36]. Replacing the existing Kinect sensor with this newer Kinect sensor could have a significant impact on the quality of captured data.

Rather than using a Kinect sensor, another option would be to acquire depth data from a low-cost laser scanner. The accuracy of the laser scanner is dependent on several criteria, including the laser line width, laser line quality, and resolution and quality of the optical sensor [19]. A low-cost laser scanner could be constructed from a laser line diode and camera module for the Raspberry Pi. With the Raspberry Pi's camera module having a maximum resolution of 2592x1944 [41], it should be able to give results similar to those seen when using a DSLR camera [22]. By switching over to using data collected from a laser scanner rather than a Kinect, the triangulation step would need to be altered to create triangles using data from successive adjacent scans.

Using a laser scanner based solely on the Raspberry Pi could also eliminate the need for both the computer running the Kinect sensor and the Kinect sensor itself. This would lower the overall cost of the scanning setup, but would still require some external computing power to perform the more computational and memory expensive operations. The developed algorithm is quite CPU and memory intensive, but this work could be offloaded to a cloud-computing system, with the results being sent back to the Raspberry Pi. If this approach were taken, it would be wise to reduce the amount of data being stored to disk and sent to the cloud. To do this, rather than storing the entire captured depth array, only the list of valid triangles resulting from the triangulation step would be stored to disk and sent across the network for processing. Although this would lower the amount of required disk space and network traffic, it would also make it impossible to re-process the same data using different triangulation thresholds.

5.2.3 Parallelized Implementation

Performance of the voxel octree intersection technique is currently limited to running on a single thread. This greatly limits performance, and does not make good use of computing resources, especially in a multi-core and multi-threaded environment.

The simplest way of running the algorithm in parallel is to split the octree, and have one core process the contents of half the tree. For example, if running on a computer capable of running four simultaneous threads, the octree would be split into four – each thread processing two adjacent child nodes of the root of the octree. Any triangles spanning these root child nodes would need to be duplicated, so that each respective thread would have a copy of the triangle. Because these triangles are duplicated, and because the way the neighbour checks are done when generating the final geometry, there is no need to reconcile the data between adjacent child nodes so long as the portion of code generating the final geometry has access to all resulting portions of the octree.

Another option when attempting to increase the performance of the algorithm through parallelism is running the algorithm on the GPU. It this sense, it would have similar system requirements to KinectFusion. Work has shown that an octree based KinectFusion is possible [48], and that GPU based parallel computing when working with octrees is also possible [43].

5.2.4 Caching Systems

A caching scheme for the triangle intersection step of the technique may greatly increase performance. When a triangle is intersection tested against the tree, the intersection tests begin at the root of the tree and end at one or more leaf nodes. This requires intersection tests to be performed at each level of the tree for each child at that level. Suppose the initial dimension of the root node are 1.0 m in the X, Y, and Z dimensions and that the minimum size of a leaf node is 1 mm. This means that each leaf node will be $\frac{1}{1024}$ m in size. If the root of the tree is considered to be at a height of 0, the leaf nodes will be at height 10. In order to determine which nodes are intersecting a given triangle, all 11 levels of the tree will need to be tested. As the intersection test is capable of adding children to a given node, all 8 children must be tested for intersections. This means that at a minimum, $8 * 11$ intersection tests will need to be performed for any given triangle, plus an additional intersection test to see if the triangle initially falls within the bounds of the root node.

Because of the initial structure to the scanned data, subsequent triangles being intersection tested are likely to be in a similar local area. A combination of a small cache and hashing scheme could be implemented to speed up these intersection tests. A hash of the maximum axis-aligned bounding box of the triangle would be tested against the resulting hash of the last few nodes. If the hashes match, indicating that they both fall within the same approximate dimensions within the tree, the intersection tests would begin at the hashed node.

Rather than using a small cache, another way to improve performance would be to look at the last leaf node accessed by the algorithm. Rather than starting at the root of the tree, the algorithm would start at the last used leaf, and walk up the tree until it finds a node that entirely encompasses the given triangle. The process would then recurse toward the leaf nodes, as it currently does. This also takes advantage of the proximity of subsequent triangles, and would only improve performance so long as triangles were in proximity of each other. This means that when testing triangles found along the bounds of a snapshot of data, performance may be worse, but the overall general case would still be better. Some analysis into whether this method of walking up the tree, then back down may show an improvement in performance.

5.2.5 Virtualized Scanning for Large Areas

The voxel octree intersection technique may be adapted for use in autonomous aircraft used in mapping indoor and outdoor environments. Rather than using a single octree to track a specified region of space, the technique could be modified to cover a larger area by using multiple octrees. Each tree would still track data which falls between its minimum and maximum bounds. Data falling outside the bounds of one tree would be tracked in neighboring trees. This would still allow each respective area to be scanned in detail while only tracking areas where depth information is found.

Bibliography

[1] The hobbit: Weta returns to middle-earth. `http://www.fxguide.com/featured/the-hobbit-weta/`. Accessed: 2014-05-14.

[2] 21st international symposium on graph drawing, gd2013. In *21st International Symposium on Graph Drawing, GD2013*, volume 8242 of *Lecture Notes in Computer Science*, Berlin, Germany, 2013. Springer.

[3] Tomas Akenine-Möller. Fast 3d triangle-box overlap testing. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[4] Nina Amenta, Sunghee Choi, and Ravi Krishna Kolluri. The power crust. In *Proceedings of the sixth ACM symposium on Solid modeling and applications*, pages 249–266. ACM, 2001.

[5] M.R Andersen, T. Jensen, P. Lisouski, A. K. Mortensen, M. K. Hansen, T. Gregersen, and P. Ahrendt. Kinect depth sensor evaluation for computer vision applications. Technical Report ECE-TR-6, Department of Engineering - Electrical and Computer Engineering, Aarhus University, feb 2012.

[6] Emmanuel P. Baltsavias. A comparison between photogrammetry and laser scanning. *ISPRS Journal of Photogrammetry and Remote Sensing*, 54:83–94, 1999.

[7] Martin Bechthold. Teaching technology: Cad/cam, parametric design and interactivity. In *Predicting the future–Proceedings of the 25th International eCAADe Conference, Frankfurt*, pages 767–775, 2007.

[8] Fausto Bernardini and Chandrajit L Bajaj. Sampling and reconstructing manifolds using alpha-shapes. 1997.

[9] Fausto Bernardini, Joshua Mittleman, Holly Rushmeier, Cláudio Silva, and Gabriel Taubin. The ball-pivoting algorithm for surface reconstruction. *Visualization and Computer Graphics, IEEE Transactions on*, 5(4):349–359, 1999.

[10] Matthew Bolitho, Michael Kazhdan, Randal Burns, and Hugues Hoppe. Parallel poisson surface reconstruction. In *Advances in Visual Computing*, pages 678–689. Springer, 2009.

[11] Simon Bradshaw, Adrian Bowyer, and Patrick Haufe. The intellectual property implications of low-cost 3d printing. *ScriptEd*, 7(1):5–31, 2010.

[12] Glen Bull and James Groves. The democratization of production. *Learning & Leading with Technology*, 37(3):36–37, 2009.

[13] Evgeni V Chernyaev. Marching cubes 33: Construction of topologically correct isosurfaces. *Institute for High Energy Physics, Moscow, Russia, Report CN/95-17*, 42, 1995.

[14] Adir Cohen, Amir Laviv, Phillip Berman, Rizan Nashef, and Jawad Abu-Tair. Mandibular reconstruction using stereolithographic 3-dimensional printing modeling technology. *Oral Surgery, Oral Medicine, Oral Pathology, Oral Radiology, and Endodontology*, 108(5):661 – 666, 2009.

[15] Malcolm N. Cooke, John P. Fisher, David Dean, Clare Rimnac, and Antonios G. Mikos. Use of stereolithography to manufacture critical-sized 3d biodegradable scaffolds for bone ingrowth. *Journal of Biomedical Materials Research Part B: Applied Biomaterials*, 64B(2):65–69, 2003.

[16] Yan Cui, S. Schuon, D. Chan, S. Thrun, and C. Theobalt. 3d shape scanning with a time-of-flight camera. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 1173–1180, June 2010.

[17] Paul W de Bruin, FM Vos, Frits H Post, SF Frisken-Gibson, and Albert M Vossepoel. Improving triangle mesh quality with surfacenets. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2000*, pages 804–813. Springer, 2000.

[18] Klaus Engel, Markus Hadwiger, Joe M Kniss, Christof Rezk-Salama, and Daniel Weiskopf. *Real-time volume graphics*. Ak Peters Natick, 2006.

[19] Hsi-Yung Feng, Yixin Liu, and Fengfeng Xi. Analysis of digitizing errors of a laser scanning system. *Precision Engineering*, 25(3):185 – 191, 2001.

[20] Sarah FF Gibson. Constrained elastic surface nets: Generating smooth surfaces from binary segmented data. In *Medical Image Computing and Computer-Assisted InterventationMICCAI98*, pages 888–898. Springer, 1998.

[21] Tom Greaves. Scanning in the streets. *The American Surveyor*, 5(11), December 2008.

[22] HackADay. 3d scanner with remarkable resolution. `http://hackaday.com/2013/05/15/3d-scanner-with-remarkable-resolution/`. Accessed: 2014-05-01.

[23] MakerBot Industries. Makerbot digitizer. `http://store.makerbot.com/digitizer`.

[24] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. Poisson surface reconstruction. In *Proceedings of the fourth Eurographics symposium on Geometry processing*, 2006.

[25] K. Khoshelham. Accuracy analysis of kinect depth data. *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XXXVIII-5(W12):133–138, aug 2011.

[26] Josef Kohout, Michal Varnuka, and Ivana Kolingerov. Surface reconstruction from large point clouds using virtual shared memory manager. In Marina Gavrilova, Osvaldo Gervasi, Vipin Kumar, C.J.Kenneth Tan, David Taniar, Antonio Lagan, Youngsong Mun, and Hyunseung Choo, editors, *Computational Science and Its Applications - ICCSA 2006*, volume 3980 of *Lecture Notes in Computer Science*, pages 71–80. Springer Berlin Heidelberg, 2006.

[27] H.J. Landau. Sampling, data transmission, and the nyquist rate. *Proceedings of the IEEE*, 55(10):1701–1706, Oct 1967.

[28] Michael E Leventon and Sarah FF Gibson. Model generation from multiple volumes using constrained elastic surfacenets. In *Information Processing in Medical Imaging*, pages 388–393. Springer, 1999.

[29] Marc Levoy, Kari Pulli, Brian Curless, Szymon Rusinkiewicz, David Koller, Lucas Pereira, Matt Ginzton, Sean Anderson, James Davis, Jeremy Ginsberg, Jonathan Shade, and Duane Fulk. The digital michelangelo project: 3d scanning of large statues. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 131–144, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[30] Hod Lipson and Melba Kurman. *Fabricated: The New World of 3D Printing*. Wiley, Indianapolis, IN, 2013.

[31] Claudio Lobos, Yohan Payan, and Nancy Hitschfeld. Techniques for the generation of 3d finite element meshes of human organs. *arXiv preprint arXiv:0911.3884*, 2009.

[32] Larry Matthies, Takeo Kanade, and Richard Szeliski. Kalman filter-based algorithms for estimating depth from image sequences. *International Journal of Computer Vision*, 3(3):209–238, 1989.

[33] Popular Mechanics. Jay leno's 3d printer replaces rusty old parts. `http://www.popularmechanics.com/cars/jay-leno/technology/4320759`. Accessed 2014-05-13.

[34] R. Mencl and H. Muller. Interpolation and approximation of surfaces from three-dimensional scattered data points. In *Scientific Visualization Conference, 1997*, pages 223–223, June 1997.

[35] Microsoft. `http://msdn.microsoft.com/en-us/library/hh973078.aspx#Depth_Ranges`. Accessed: 2014-05-01.

[36] Microsoft. Kinect for windows features. `http://www.microsoft.com/en-us/kinectforwindows/discover/features.aspx`. Accessed: 2014-05-01.

[37] Microsoft. Revealing kinect for windows v2 hardware. `http://blogs.msdn.com/b/kinectforwindows/archive/2014/03/27/revealing-kinect-for-windows-v2-hardware.aspx`. Accessed: 2014-05-01.

[38] NASA. Nasa tests limits of 3-d printing with powerful rocket engine check. `http://www.nasa.gov/exploration/systems/sls/3d-printed-rocket-injector.html`. Accessed: 2014-05-13.

[39] Richard A. Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J. Davison, Pushmeet Kohi, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In *Mixed and Augmented Reality (ISMAR), 2011 10th IEEE International Symposium on*, pages 127–136, Oct 2011.

[40] Gregory M. Nielson and Bernd Hamann. The asymptotic decider: Resolving the ambiguity in marching cubes. In *Proceedings of the 2Nd Conference on Visualization '91*, VIS '91, pages 83–91, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[41] Raspberry Pi. Faqs - raspberry pi. `http://www.raspberrypi.org/help/faqs/#camera`. Accessed: 2014-05-02.

[42] C. Rocchini, P. Cignoni, C. Montani, P. Pingi, and R. Scopigno. A low cost 3d scanner based on structured light, 2001.

[43] Michael Schwarz and Hans-Peter Seidel. Fast parallel surface and solid voxelization on gpus. In *ACM SIGGRAPH Asia 2010 Papers*, SIGGRAPH ASIA '10, pages 179:1–179:10, New York, NY, USA, 2010. ACM.

[44] David M. Shotton. Confocal scanning optical microscopy and its applications for biological specimens. *Journal of Cell Science*, 94:175–206, 1989.

[45] S.N. Sinha, D. Steedly, and R. Szeliski. Piecewise planar stereo for image-based rendering. In *Computer Vision, 2009 IEEE 12th International Conference on*, pages 1881–1888, Sept 2009.

[46] Debbie Sniderman. 3d scanning options. *Desktop Engineering*, pages 20–23, jan 2011.

[47] David F Watson. *Contouring*, volume 5.

[48] Ming Zeng, Fukai Zhao, Jiaxiang Zheng, and Xinguo Liu. Octree-based fusion for realtime 3d reconstruction. *Graphical Models*, 75(3):126 – 136, 2013. Computational Visual Media Conference 2012.

[49] Scott Zimmer. The right to print arms: the effect on civil liberties of government restrictions on computer-aided design files shared on the internet. *Information & Communications Technology Law*, 22(3):251–263, 2013.

Appendix A

Scanner Design

The 3D scanner created for this thesis was designed to use low-cost, commercially available hardware components. All components for the scanner were either purchased online, at local hardware stores, or were 3D printed. The following contains a brief overview of the scanning hardware, electronics, and software created for the scanner.

A.1 Scanner Hardware Design

Figure A.1 shows a general overview of the scanner, excluding the Kinect sensor and electronic components. The scanner is constructed from 1/2 inch (1.27 cm) baltic birch plywood. The plywood portions of the design are fastened using wood screws, unless otherwise indicated.

The Kinect platform is held in place by sliding a pair of 1/4 inch bolts through holes drilled in both the tower and the Kinect platform. These regularly spaced holes along the height of the tower allow the Kinect platform to be moved up or down, depending on the size of the object being scanned. The Kinect is held on to the Kinect platform by a 3D printed bracket, which is bolted to the Kinect platform with a 1/4 inch bolt. This allows the Kinect sensor to be securely attached to the platform without causing damage to the sensor, allowing it to be re-used in other projects. The tower is attached to the rest of the scanner via a pair of 1/4 inch bolts. This allows the tower portion of the scanner to be removed, allowing for easier transport. If either the Kinect platform or tower is removed and replaced, the scanner must be re-calibrated prior to use.

Details of the scanner base are shown in A.2. The scanner base consists of several supports, a small metal plate, a stepper motor, and a 'lazy susan' style bearing. The metal plate contains a round hole in the center, allowing the stepper motor to protrude through it. The stepper motor is attached to the plate with 6 mm bolts, and the plate is then screwed into the rest of the scanner base using wood screws. One side of the stepper motor shaft is keyed to allow it to grip a 3D printed bracket that is attached to the underside of the scanning platform. The 'lazy suzan' style bearing is also secured to the scanner base using wood screws. The 3D printed bracket is attached to the bottom of the scanning platform using self-tapping metal screws. A 1/4 inch hole drilled in the center of the platform allows the scanning platform and 3D printed bracket to be lowered onto the keyed shaft of the stepper motor.

This particular hardware design ensures that the object being scanned and Kinect are held in a fixed position relative to each other. This eliminates the need for registering scans

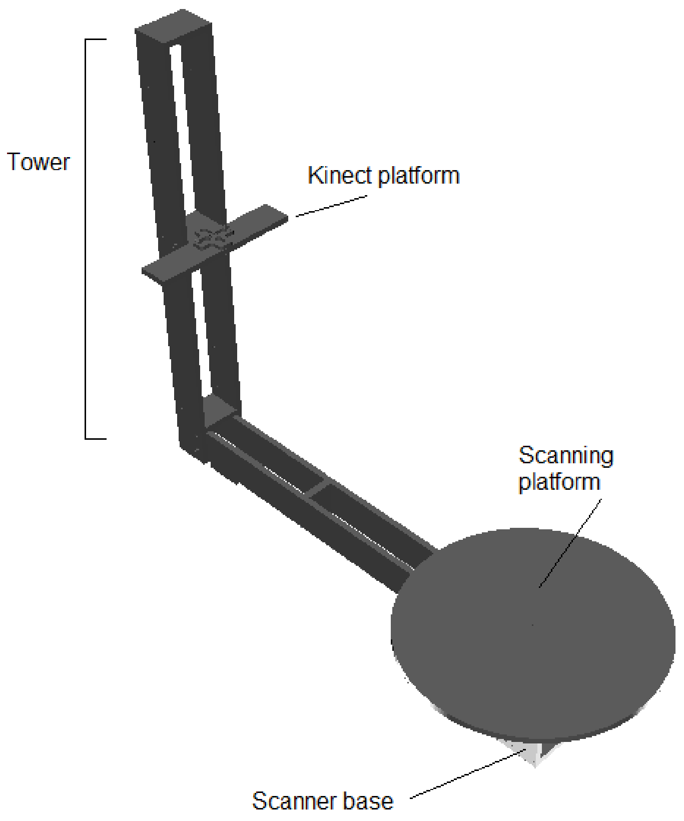


Figure A.1: An overview of the scanning hardware setup.

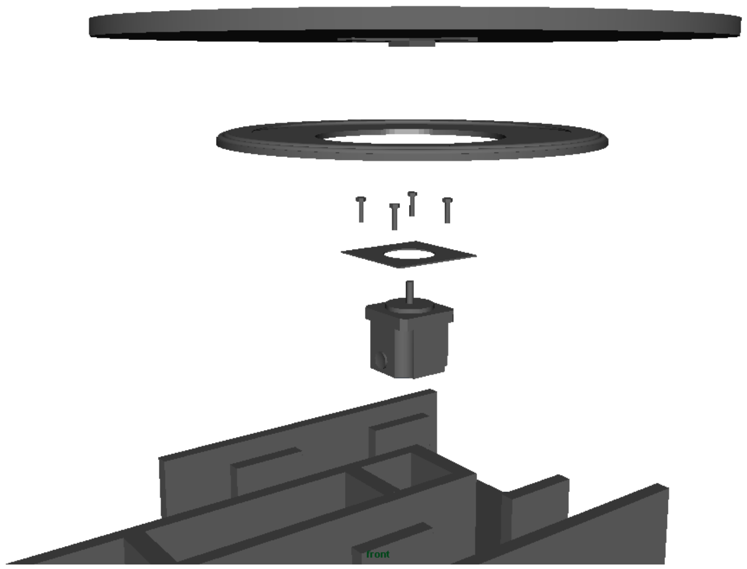


Figure A.2: Expanded details of the scanner base, in exploded view.

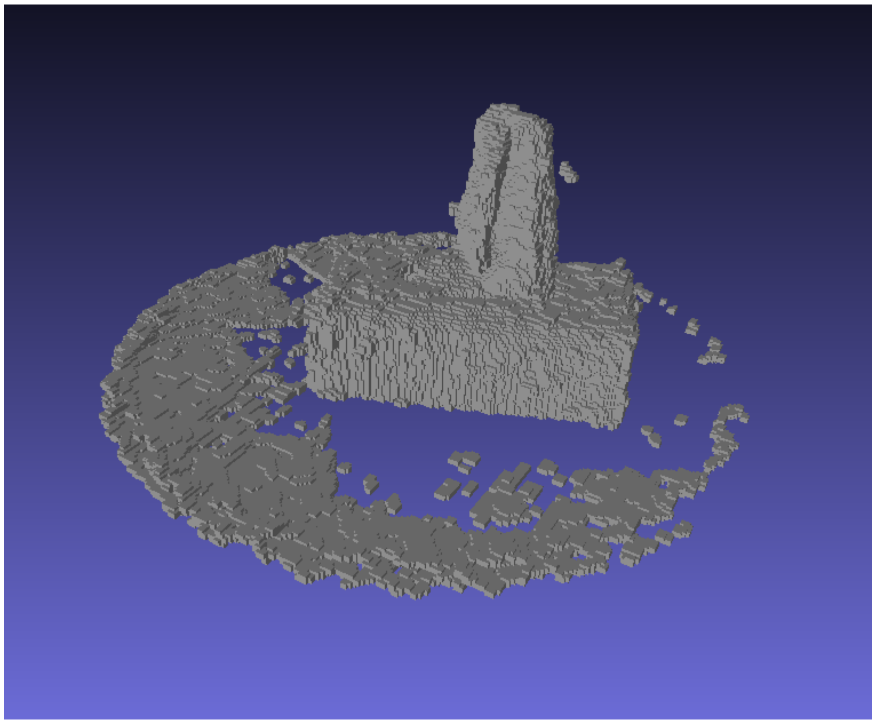


Figure A.3: A scan showing non-planar platforms.

with a base model or explicitly tracking scanner movements, such as that done by Kinect Fusion [39].

In general, the design has proven to be robust enough to allow for the results shown in Chapter 4. The only issue encountered early on was that of some torsion of the frame between the tower and scanner base. As a result, the scanning platform and Kinect platform were not perfectly aligned within the same plane. This effect is visible in Figure A.3. If the scanning platform and Kinect platform were in perfect alignment, the portions of the scanning platform included in the scan would have been uniform on all sides of the object. To hide this, the lower clipping value on the Y axis was increased so that no portions of the scanner platform were included in any of the scans. Although this hid the problem, it also resulted in the bottom few mm of each scan being removed in the clipping step of the technique. To prevent this from happening, future designs should consider increasing the rigidity of the scanner structure, adding supporting legs to the tower, or add a method of adjustment to the Kinect platform to allow its roll to be adjusted.

A.2 Electronics

The electronic components of the hardware design include a Raspberry Pi computer, a stepper motor controller, and a stepper motor. The Raspberry Pi receives signals across the network from the computer running the Kinect sensor. These signals instruct the Pi to turn the platform by a given amount. The Raspberry Pi then uses its GPIO (General Purpose Input Output) pins to send a signal to a stepper motor controller which turns the platform by a given number of steps. The stepper motor controller used in this setup was a

A4988 based unit. This unit is powered from the same small brick-like power supply which provides both the 5v for the logic and the 12v for the stepper motor.

To enable finer grain control when performing a scan, microstepping is enabled on the stepper motor controller. The A4988 supports several degrees of micro-stepping, including quarter-stepping, which was chosen for this application. Although the stepper motor controller supports up to sixteenth step microstepping, quarter stepping was found to provide more than adequate resolution without losing too much torque. With quarter-stepping enabled, the stepper motor turns 0.45 degrees per step, as opposed to the default 1.8 degrees per step.

The particular stepper motor chosen for this application is a uni-polar stepper motor capable of generating 125 oz./in (0.88 n/m) of torque. Enabling micro-stepping on the stepper motor controller does reduce this torque, but it has proven to still be adequate for the purposes for which it is used. For scanning heavier objects, it may be desireable to include a reduction gear system to allow for even greater torque to turn the scanning platform.

Appendix B

Cost Breakdown

As previously mentioned, one goal with this work is to keep the cost of a 3D scanner as low as possible. This is achieved by using low-cost, off the shelf, commercially available components. Table B.1 is a breakdown of costs associated with the project. All prices are stated in Canadian dollars, and do not include taxes or shipping costs on components, the computer running the Kinect sensor, networking equipment connecting the computer and Raspberry Pi, and other minor incidental costs such as the cost of screws and 3D printed brackets. Parts may not be available in all regions, but substitute parts may be used so long as they perform the same function as the parts listed below.

Table B.1: Approximate cost of scanning hardware.

Description	Approximate cost
Microsoft Kinect for Windows	\$140
1/2” (1.27 mm) Baltic birch plywood	\$50
Lazy suzan style bearing	\$13
Stepper motor	\$25
A4988 stepper motor controller	\$10
Raspberry Pi	\$40
Power supply for stepper and stepper motor controller	\$15
Total:	\$293

For our particular scanner, the baltic birch plywood was sourced from scrap material, the lazy suzan style bearing was purchased from a local hardware store, and the remaining materials were acquired from various online sources.