NAIT

Edmonton, Alberta

Automatic 3D Scanner

As a submission to
Mr. Kelly Shepherd, Instructor
English and Communications Department

Submitted by Jonas Buro, Student
CMPE 2960
Computer Engineering Technology

April 27, 2018

11762 106 St NW
Edmonton, AB T6J 2R1

April 20, 2018

Mr. Kelly Shepherd
Instructor, CNT Department
NAIT
11762 106 St NW
Edmonton, AB T6J 2R1

Dear Mr. Shepherd,

I am submitting the report *Automatic 3D Scanner* for your evaluation. This report fulfills the major focus of CMPE2960: Computer Engineering Capstone.

This report details the hardware and software design of an automatic 3D scanner. It explains fully the automatic process which transforms a physical real world object into a standardized digital representation.

I'd like to thank all of my instructors within the CNT department at NAIT for their willingness to share their detailed knowledge of everything computer related. Their helpfulness was a valuable resource in the creation of this project.

Sincerely,

Jonas Buro
CNT Student

Contents

List of Figures

Abstract

The digitization of real world objects is incredibly powerful and is useful for a wide variety of applications. From understanding and replicating the functionality of complex parts and shapes, to accurately mapping the environment, to creating more realistic digital entertainment, 3D scanning is growing quickly throughout many disciplines, especially mechanical engineering. The automatic 3D scanning process involves acquiring depth information of the target object, transforming this data into a clean, usable format, and then processing it to create a digital representation. We leveraged the Windows Kinect's infrared sensing technology and its C# development kit in tandem with a rotating platform to construct a point cloud from a real world object. We then transformed the points into a standardized 3D digital object that represents the original by incorporating robust reconstruction algorithms developed by pioneers in the 3D scanning field in addition to our own. The final deliverable is accessible in most 3D printing and viewing applications.

1 Introduction

3D scanning technology is a very useful tool in a multitude of applications. The digitization of real world objects allows for the non intrusive analysis of the object's structure, the modification of its structure, and the accurate replication of the object through 3D printing. These processes extend benefits into most design processes and will continue to gain traction as the technology is refined.

The purpose of this report is to fully explain the processes involved in constructing a 3D scanner, as well as developing the software which transforms the acquired data into a clean and useful format. We decided to take on this project as we were interested in developing a fully fledged software project which integrates a hardware component. The Windows Kinect v2 sensor which we leveraged in this project has a standard development kit accessible through C#, the primary object oriented programming language that was taught in the CNT program at NAIT. The mathematical processes which constitute the post processing reconstruction of our design are also of interest to us. We set out to deliver a fully transferable digitized 3D representation of a real world object and succeeded in doing so.

This report will first analyze the hardware which we chose for our design, then inspect the communication procedures and protocols between devices, and finally detail the software development process of building a client application and reconstructing the gathered data. The sections of this report are visible in the table of contents for navigation purposes.

2 Hardware

Within this section are descriptions of the mechanical and electrical technologies we used to create an automatic 3D scanner.

2.1 Kinect

In order to represent a real world object digitally, all exterior surfaces of the object must be sampled and reconstructed. In order to acquire this surface information, some kind of sensor/receiver pairing must be utilized. The choice we made for our project was the Windows Kinect v2 for Xbox 1. This powerful device can be controlled through its standard development kit from Microsoft, which allows the user to program in C#. The Kinect v2 features a 640-480 pixel IR-depth finding camera which runs at up to 30Hz. The depth sensor consists of an infrared laser projector combined with a monochrome CMOS sensor, which captures video data in 3D under any lighting conditions, at 11-bit depth resolution. The infrared transmitter array transmits invisible light and measures its time of flight after reflecting off of the target object, in order to construct a field of points with depth information. The sensor's acquisition field is user definable and allows us to restrict the captures to the minimum volume required for the scanning of an object. Although no official documentation exists on the accuracy or precision of the Kinect v2, the test runs within the community conclude that at 50cm, the depth resolution is accurate to within 1.5mm, and although this is not great, we can account for this by taking multiple samples and then averaging their values (more on this in the software section). To interface with the Kinect, users must power it and connect to it to a computer via USB 2.0 or greater, through its combined power and data adapter. We circumvented this peculiarity with our second Kinect device by opening it up and manually soldering in the power connection and connecting to it with a standard USB2.0 cable.

Figure 1: Microsoft Kinect v2

2.2 Platform

In order to obtain the entire surface information of a particular object, one must view it from all angles. One can achieve this in a few ways, including a multiple sensor setup or a rotating sensor. We opted instead to implement a static sensing/receiving unit and a rotating platform, as we believed that building a moving sensor would have required a higher degree of mechanical precision. The platform was built in AutoCAD and 3D printed. It consists of a circular platform who's center point connector is form fit for a type D stepper motor shaft and is secured with a set screw. The motor is connected to the platform and then attached to a 3D printed plastic support mount which features bearings to support the rotating platform from below. The inclusion of bearings is a necessity in this case, as the added support allows the platform to rotate orthogonally with respect to the sensor in a smooth manner.

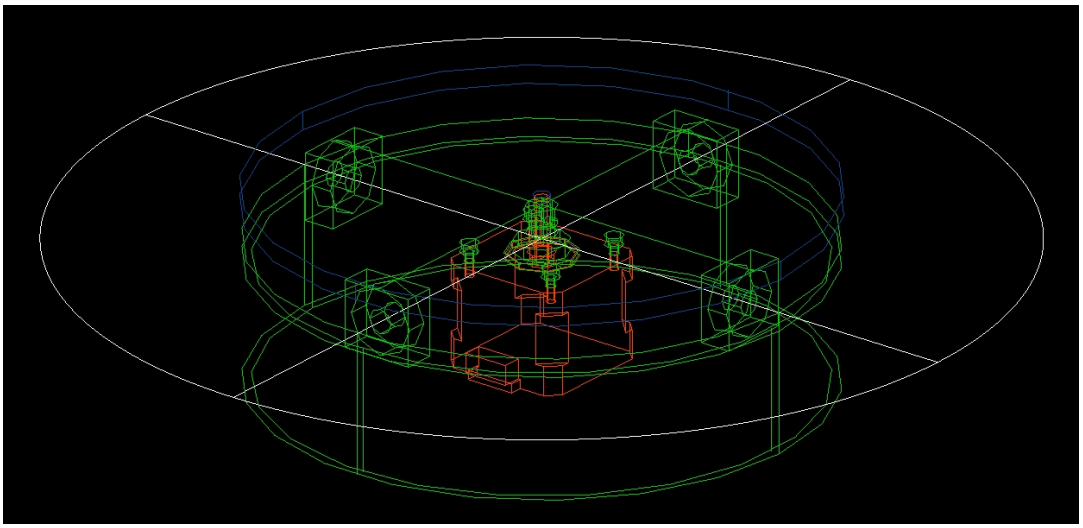


Figure 2: Rotating platform schematic

2.3 Motor

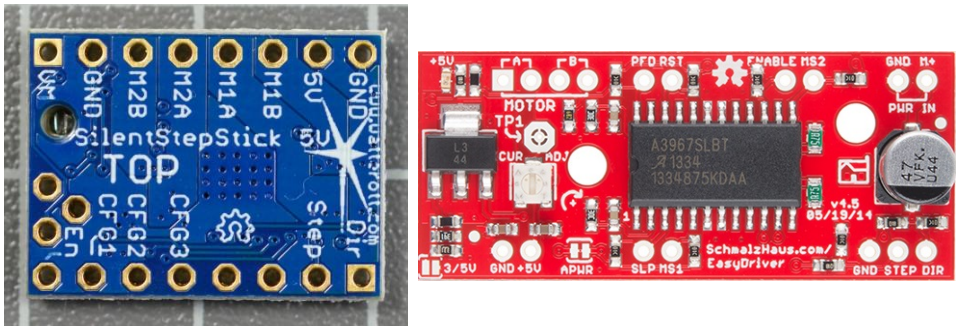
The driving force behind our project is a Nema 17 200 step bipolar stepper motor, selected specifically for efficiency and precision. Our motor is rated for 5V/2A, and allows for a holding torque of 3.2 kg-cm, more than enough to handle our maximum target object weight specification of 3kg. Stepper motors are able to control the angular position of the rotating shaft, called the rotor, without a feedback loop and are very precise, both great features for our design. Stepper motors have multiple electromagnetic coils around a central gear shaped piece of magnetic metal, such as iron. These electromagnets are energized through an external source, creating a temporary electric field which in turn creates a temporary magnetic field which, depending on the particular orientation, attract or repel the metal gear on the rotor, shifting it to the next position. Bipolar motors have a single electromagnet per phase, so in order to achieve full rotation, current must pass through these windings in both directions, resulting in a more complex driving sequence compared to a unipolar motor. This problem was easily routed by obtaining a stepper motor driver chip which contains a driver block which can supply the necessary reversal of current.

Figure 3: Nema 17 stepper motor

2.4 Motor Driver Chips: TMC2130 and EasyDriver v4.5

In order for the stepper motor to turn, it requires current to flow through the windings to align the magnetic rotor. The sequence in which the windings are energized is essential for coherent operation of the motor, and can be relatively complicated based on the motor type. The TMC2130 chip provides an integrated motor driver solution to this problem. Additionally, the chip contains a multitude of additional features, such as a microstepping indexer, stall detection technology, load dependant current control, as well as other features. The driver block of this chip consists of N-Channel MOSFETS in an H-Bridge configuration to drive the motor windings. Additionally, the device has an SPI interface for configuration and diagnostics, as well as a standalone step/direction interface for easy operation. We combined both interfaces in order to analyze the status of the rotating platform while issuing it step commands. The primary motivation behind sourcing this chip was to leverage its PWM microstepping capabilities to gain a higher resolution point cloud, however, we were unable to implement this before the deadline. From a software point of view, the chip is a peripheral with a number of control and status registers which can be written to or read from. We utilized a third party library to interface with this chip, and modified it to match our demands.

The EasyDriver v4.5 motor driver chip is our fallback hot switch chip. It's functionality and use is mostly identical to the TMC2130, however it lacks SPI support. Both chips can drive around 750 mA per phase of a bi-polar motor, and are a necessary intermittent step in the communication chain as the microcontroller can't issue this amperage from an output pin.



(a) TMC2130 (b) EasyDriver

Figure 4: Stepper motor driver chips

2.5 Microcontroller

In order to interface between the client application and the stepper motor driver chip, an intermediary device must be used in order to supply the necessary digital pulses to the required pins. We chose a Genuino Uno, because it is well documented, easy to use, and has a small form factor. The Uno is a widely used open source microcontroller board based on the ATmega328P. It contains 14 digital IO pins, a 16 MHz crystal, and a type B USB interface. It is programmable with the Arduino integrated development environment via USB. The ATMega328 has a bootloader already built in so that you can use it without the use of an external hardware programmer out of the box. Conveniently, the Uno also has SPI support through the standard Arduino SPI library, necessary for sending configuration commands to the motor driver chip. In total, we used seven digital output pins of the Uno to communicate with the motor driver chip; the configuration is detailed in Figure 5.

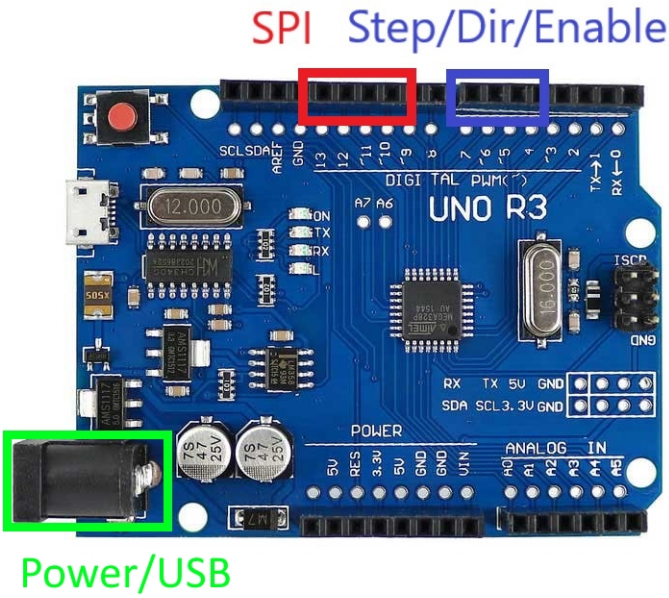


Figure 5: Genuino Uno microcontroller

3 Communications

This section contains information detailing the communication protocols between our hardware components.

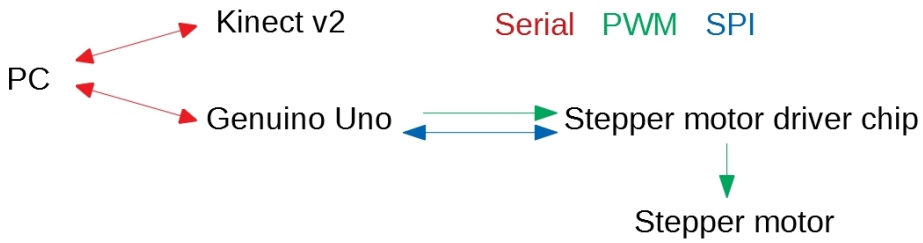


Figure 6: Data flow

3.1 Stepper Motor Driver ↔ Micro

Forming an information bridge between the Genuino Uno and the TMC2130 stepper motor driver chip was relatively straightforward, thanks to both the built in Arduino SPI library and a third party library we sourced for the chip. By connecting pins 10-13 to their counterparts on the chip (see figure below), we formed a SPI connection, which is an interface bus commonly used to send data between microcontrollers and peripherals. It utilizes a master-slave configuration, in which the master device (in this case, the microcontroller) slave selects a single peripheral at a time for communication. This configuration requires more signal wires than serial, and the communications must be well-defined prior to operation, but it yields higher transmission speeds, allows for data exchange in both directions simultaneously, and can support more than one communication partner. SPI communications require a clock line, two data lines (MISO/MOSI), and a select line, as opposed to serial's two data differential wires. SPI is synchronous, which syncs both the sender and receiver through the clock. The oscillating clock tells the designated receiver when to sample the bits on the data line. Depending on the configuration, this can either occur at the rising or falling edge of the clock pulse, based on the polarity configuration of the clock. Another advantage this interface

provides is that the receiver can be a simple shift register instead of the fully developed UART. For our specific peripheral, the TMC2130, whenever data is read from or written to it, the MSBs delivered back to the micro contain the status of the driver. Based on the parity of the returned bits, we can compare the result to a register lookup table and determine the status of the rotating platform. Within our communications protocol coded using the gcc compiled Uno IDE, we read from the SPI directly after issuing a step command to the motor. If we receive an error status, we acknowledge it and let the client application know. More on this in the next section.

3.2 Microcontroller

The Uno communicates with the PC running the client application through a USB ≥ 2.0 connection. A serial communication interface transfers one bit in or out at a time, a major difference to the above mentioned SPI. This method of communication has been the most common method to transfer data between two digital devices throughout the history of computing. Since asynchronous serial interfaces do not pair data lines with a clock line, it means that we need to put more effort into reliably receiving and transferring data. Both parties must be correctly configured in order to have an exchange which makes sense. The Arduino program is set up into a setup block, and a loop block. On power up, the setup code executes once, and then the code inside of the loop block gets repeatedly executed as quickly as possible, if not throttled. The protocol we built contains a blocking delay while the data line is clear, and as soon as data is available, it is checked for an expected start of transmission character. If the character is found, the remaining bytes in the line will be read into a char array until an end of transmission character is found, or the read times out. If the end of transmission character is found, the data received is flagged as good, and an if-else ladder determines the next computational step to take. On the application side, things get more interesting.

3.3 PC

Within our C# application, a layered class hierarchy constructs and handles a serial port object's interactions with the outside world. The first layer, named the abstraction layer, communicates directly with the hardware and builds a thin wrapper around .NET's SerialPort class. It is built for re-usability and allows the user to easily instantiate a serial port and communicate with it directly or through a business logic layer class. This application specific class packages the message to be sent out and attempts to unwrap received messages, as well as handling any error conditions found after unwrapping. Finally, the main WPF within our application spins up an instance of the business layer and communicates with it in basic terms through its public interface. All data exchange between classes follows an event-driven programming model; any send or receive event cascades events into the other classes. Figure 7 gives a visual representation.

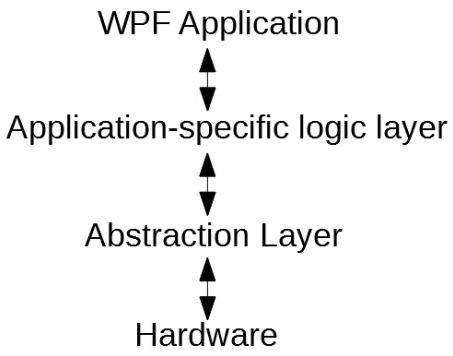


Figure 7: Serial communication layers

4 Software

This section contains details on the pre and post-processing required to transform the depth frames acquired from the sensor/receiver pairing into a standardized 3D representation of the target object, as well as the navigation of the user interface of our application. The application back end was developed using C# in the Microsoft Visual Studio IDE, Microsoft's WPF graphical subsystem was used to construct the user interface, the Genuino Uno was programmed in C++ and compiled within the Arduino IDE, the majority of the C++ postprocessing code was sourced and compiled in MSVC, the C++ motor stepper driver chip libraries were modified and compiled within Visual Studio, and finally some of the file conversion methods we sourced are written in C++, modified within notepad++, and compiled using mingw32's gcc compiler. All project files were managed using multiple GitHub repositories. This report was created with TeXstudio.

4.1 Initialization

In order to initiate the automatic scanning sequence, the user must first configure the environmental and processing variables located within the application. As soon as the user executes the program, a window appears with several tabs which allow for movement to the different parts of the program. The first step is to set the alignment of the axis of rotation. This is completed by placing a solid, regular polyhedral shape perpendicular to the scanner with one edge passing over the absolute center of the platform. Using the calibration tab, the user is able to set the center point of rotation, a critical value which will be paramount in conglomerating successive scans into a point cloud which is processable. After the calibration axis is determined either programmatically or manually, within the alignment tab, the user is required to align the Kinect's field of view by indicating a spanning 3D volume which encapsulates the target object completely and does not contain any unwanted objects, such as a background object or the rotating platform itself. Then, the user has a plethora of sampling options to experiment with, in order to obtain the most accurate point cloud from scanning. These options include setting the rotation angle, which is realized as the amount of different scans per complete rotation, the amount of depth frames to to average per scan, and octal tree point cloud downsampling options, which are discussed in the software section. These settings

will propogate between program sessions by writing them to a configuration file on application close, and loading them in when the application is reopened.

4.2 Data Acquisition

As soon as the user is satisfied, he presses the scan button, which starts the scanning sequence. The below flow chart illustrates the logic flow visually. The thread which handles the scanning begins a handshaking sequence in which the Kinect Sensor acquires one or several depth information frames and stores it or the averaged result in a collection, then tells the application that the scan has been completed. Then, the application issues a step command to the microboard which routes the command to the stepper motor driver chip. After stepping and then verifying that the step occured without error, the scan command is relayed to the Kinect, and so on and so forth, until the provided number of scans of the target object have been taken and stored in memory. The collection of scans is then conglomerated into a single Vector3 array, applying a rotational transform to the points about the axis of rotation based on the angle at which the currently in focus scan was taken relative to the original scan.

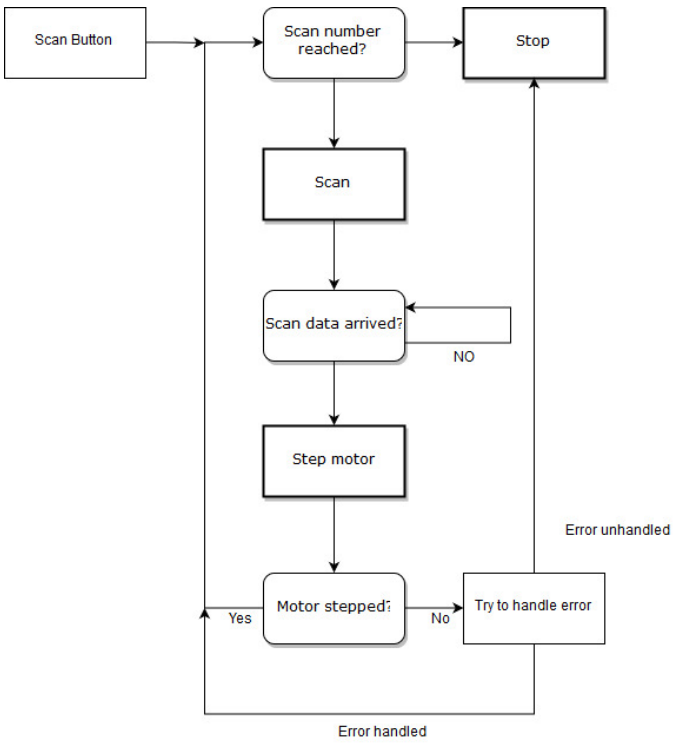


Figure 8: Data flow

4.3 Data Preprocessing

The resultant array of points is not a good approximation of the surface of the object in question, no matter what its dimensions, as the composite error of the sampling rate, the alignment of the center point, and any mechanically introduced offset of the object is non-trivial. The noisy point cloud is not particularly useful to us. At this stage we are interested in programmatically refining the data into something usable. We do this by applying an Octree inspired algorithm to intelligently downsample the data. The algorithm constructs a bounding cube around the target object, subdivides it evenly into smaller cubes of minimum user indicated length, based upon the rough estimation of the sampling error. Each cube contains a given amount of points of the original point cloud. We conduct statistical analysis on the number of points in each cube, removing all cubes containing points less than one standard deviation away from the mean. This dramatically improves the quality of the cloud. A bottom layer is constructed using the bottom most layer of the Octree subdivisions and determining whether or not a constructed grid point falls within the shape bounds. Optionally, the user may choose to apply the same algorithm, but to the top of the shape. After the data cleaning has finished, the file is exported to a temporary processing directory within the file structure of the application as a .pts points file, which is a text file which describes each point in the point cloud as Vector with an X,Y and Z component.

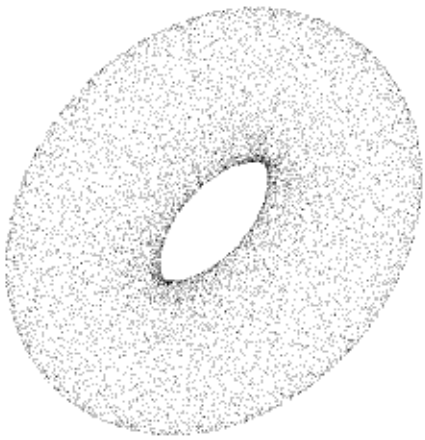


Figure 9: Example of a clean point cloud

4.4 Data Postprocessing

After the point cloud is acquired, the application programmatically builds and then executes a batch file based on the user's configuration selections. This execution begins the C++ post processing layer. The goal of this layer is to transform the point cloud into an isosurface representation of the target object. After researching methods of point cloud reconstruction, we decided to use the works of a pioneer in the field, Hugues Hoppe, to achieve this. Hugues Hoppe holds a PhD in Computing Science from the University of Washington for his thesis [1]. This thesis is an analysis and solution of how to mathematically reconstruct a 3D shape from a point cloud, using a step by step procedure. The major advantage his method provides relative to previous reconstruction methods is that no additional information is required along with the point cloud, such as structure in the data, surface genus, or orientation information. Through mathematical analysis, the method is able to infer the topological type of the object, its geometry, and location of problem features, such as sharp edges. His reconstruction method contains three major phases: initial surface estimation, mesh optimization, and piecewise smooth surface optimization. A major issue faced by our project was the application of the latter functionality due to the lack of precision in our point cloud. Hoppe's advanced reconstruction techniques' effectiveness depends primarily on the quality of points collected. A description of the algorithm's assumptions and procedures to deliver a smooth isosurface representation of a point cloud follow:

4.4.1 Definitions

In order to explain the math behind the processes involved, some definitions are necessary:

Definition 1 (mesh) *A* mesh *can be defined as a piecewise linear surface, consisting of triangular faces pasted together along their edges.*

Definition 2 (sampling process assumption(sampling noise/density)) *The* sampling process assumption *is an assumption based upon the sampling accuracy and point cloud density which logically eliminates the possibility of a given point* $r \in \mathbb{R}^3$ *being part of the surface* U *of the scanned object. More precisely, if* X *is the point cloud, then if* $\exists\delta$ *such that* $\forall x_i, y_j \in X, \delta \geq |x_i - y_j|$ *we define* X *as* δ*-noisy. If we find the minimum radius* ρ *of a sphere with center in* U *such that*

any of these spheres contains at least one point in Y, Y *being a noiseless* ρ*-dense sample of the surface* U *then* Y *is defined as* ρ*-dense. From these definitions it follows that no point* $p \in U$ *has a distance to the point cloud* X *greater than* $\delta + \rho$. *This is an important limiting factor in the size of mesh triangles, as if we go smaller than this composite error, holes begin to appear in the reconstructed mesh.*

Definition 3 (principal analysis) Principal analysis *is a technique used for identification of a smaller number of uncorrelated variables known as principal components from a larger set of data.*

4.4.2 Initial Mesh Reconstruction

The problem statement for the first part of the reconstruction process can be stated as follows: for a given unorganized, noisy sample X of an unknown surface U, create a mesh M_o that approximates U.

The key idea in this phase of processing is to develop a function $F(p), p \in X$ that determines the signed distance D to the mesh M_o in order to provide a more powerful algorithm which can compute the triangle mesh of the point cloud the information it needs to execute. To do this, we need several ingredients. The first step is to associate an oriented tangent plane $T_p \ \forall p \in X$. These shall be local approximations of the surface U. To determine these local approximations and their normals we need to inspect the group of points within distance $\delta + \rho$, denoted as $Nbhd(p_i)$. The center of the tangent plane $T(p_i)$ is the centroid of the $Nbhd(p_i)$, and the normal n_i is computed using principal analysis. A covariance matrix Cvi is formed and the normal n_i is computed by finding the unit eigenvector with smallest eigenvalue of this matrix, and then selecting the orientation such that nearby tangent planes are consistently oriented. The procedure to determine whether or not the normals are consistent within a neighbourhood can be modeled as a graph optimization problem, but we won't go into the solution here.

Now that we can compute n_i for each point p, the next ingredient comes from the realization that the distance function $D(r)$ for an arbitrary point $r \in \mathbb{R}^3$ to the surface U is simply the distance between the closest point in U to r multiplied by ± 1, depending on whether the point is inside or outside. Since U is unknown, we approximate the distance using the tangent plane $T(p)$

who's centroid is closest to r. The tangent plane is a local approximation to U so we take the signed distance between r and its projection z onto $T(p)$.

Now that we have come up with an approximating function which expresses the signed distance from an arbitrary point $r \in \mathbb{R}^3$ to the surface U, we are able to implement a variation of the marching cubes algorithm to reconstruct a mesh from the point cloud. This algorithm works by dividing the smallest bounding cube of the object into smaller cubes C, with edge length greater than the composite error in sampling rate explained above. The vertices of each smallest cube are inspected as either inside or outside of the object U. The cubes' vertex configurations (of which there are 2^8 variations as in/out is binary and there are eight edges in a cube) are then passed into a lookup table which returns the polygonal representation of the cube. These representations, taken fully, are then the polygonal mesh M_o that we are interested in.

4.4.3 Mesh Optimization

This process seeks to improve the accuracy and conciseness of the mesh M_o, in preparation for smoothing the surface. The problem can be stated as follows: Given a collection of data points $X \in \mathbb{R}^3$ and an initial mesh M_o near the data, find a mesh M of the same topological type as M_o that both fits the data well and has a small number of vertices. To solve the optimization problem, an energy function which captures the competing desires of tight geometric fit and compact representation is minimized. As input for this function we use the resultant mesh M_o from the first phase. Then, an optimization algorithm minimizes this function by varying the number of vertices, their positions, and their connectivity under the constraint that the shape of the original mesh is maintained. Simply stated, the optimization code works on the space of all meshes M which are of the same topological type as the input, seeking the simplest representation. A user defined parameter indicates how the accuracy and conciseness of the mesh should be weighed when determining the optimal mesh. This control is visible in the processing tab of our application.

4.4.4 Surface Smoothing

In essence, this final phase of the process is a natural extension of the second phase. The problem can be stated as follows: given an optimized piecewise linear surface find a concise control mesh

M of the same topological type and define a piecewise smooth subdivision surface that accurately fits the points. Similarly to phase 2, an energy minimization problem is set up that trades off conciseness and fit to the data.

4.4.5 Output

The resultant smoothed .m mesh file produced via the executed batch file is saved into a processing subdirectory in the project. A FileWatcher watches for this file creation and upon creation immediately consumes it and transforms it into an .stl file. This file is then available for viewing within the application. If the user is satisfied with the result, he can save the file on his computer using a dialog. This completes the reconstruction process.

4.5 User Interface

Our application features a very concise and easy to operate user interface. It was designed using XAML, which stands for eXtensible Application Markup Language. XAML is Microsoft's variation of XML for describing graphical user interfaces. Much like HTML, the UI is easy to build and the C# back end handles events and hooks up easily to the controls in the front end. XAML controls can be built and reused, similar to creating skin files in ASP.NET.

The UI features several tabs which the user can click to navigate to different parts of the application. Generally speaking, the options within the different tabs are used to tweak configuration options to generate the best result possible. Multiple values are taken into consideration when generating the batch file to execute the postprocessing C++ code, and .NET Dialog objects handle the file RW for a clean user experience.

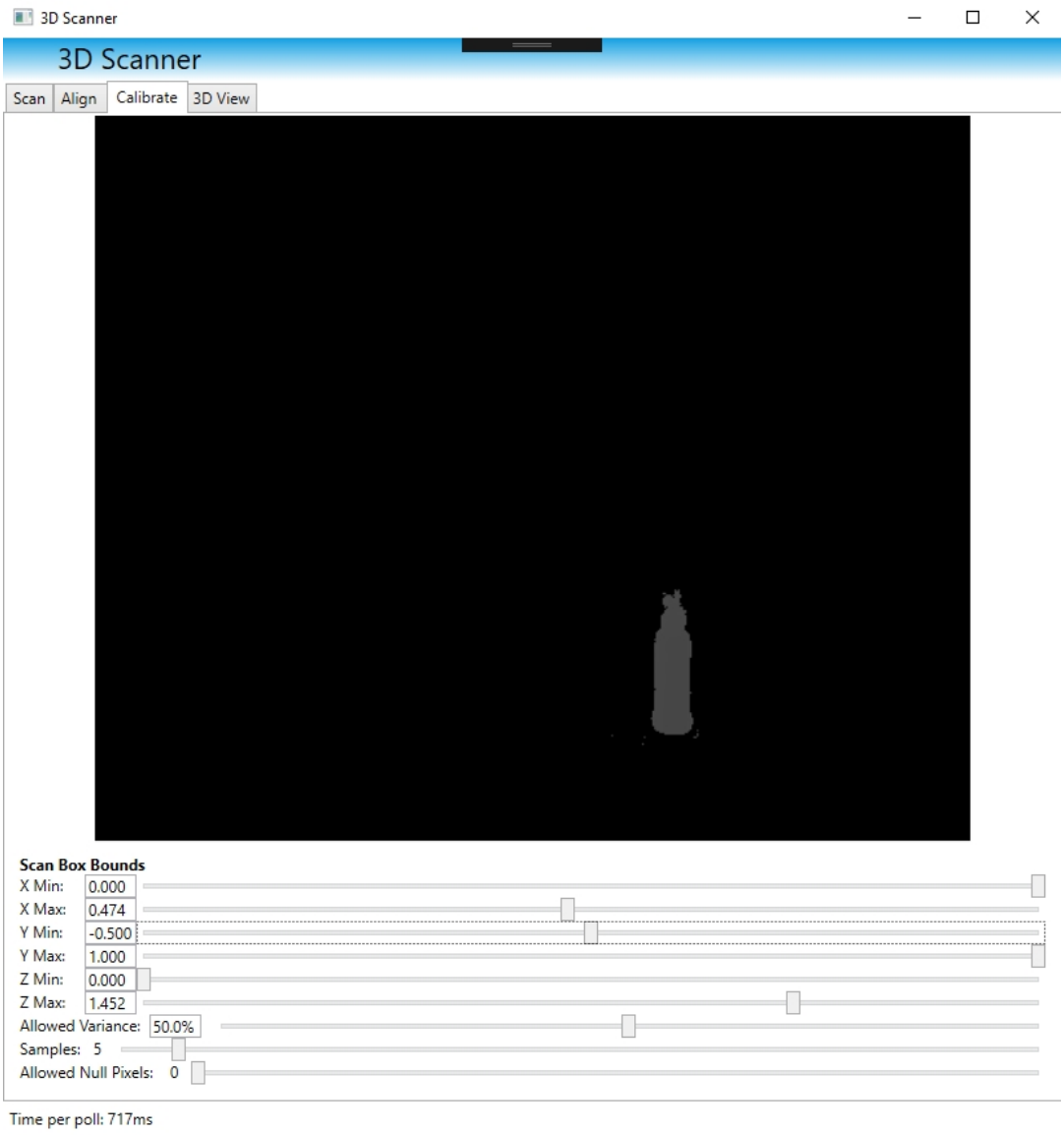


Figure 10: User interface

5 Results/Conclusion

We managed to transform non-complex real world objects into isosurface 3D representations through a completely automatic process. The deliverable at the time of writing this report is a standard .stl file which represents the object which is scanned, to varying degrees of accuracy. Reconstructing simple polyhedral shapes works great, but the quality of gathered point clouds quickly diminishes proportionally to the complexity of the target object. We were pleased with the results that we obtained, however the system has much room for improvement. If we had additional time to work on this project we would have liked to reduce the composite error of the sampling process and the point density, as it was the limiting factor in applying higher resolutions of the reconstruction algorithms we used. Much depends on the quality of point cloud data gathered at the start of process, and we would have liked to have improved on our methodology in acquiring data. Overall, the project was a success, and the team dynamic was great throughout the work period. The skills we developed will without a doubt help us in the near future when we step into our careers.