3D Scanner using Kinect

Xi Chen, Yuheng Huang, Sida Wang

Department of Engineering Science

University of Toronto

Final design report for ESC471

April 25, 2010

Summary

In this report, we describe, justify and assess the quality of our design for a 3D scanner using Microsoft Kinect.

A 3D model of an object can be obtained by taking both color and depth images from different viewpoints. We developed a streamline and a novel combination algorithm for this task. In the process, we had to solve problems involving noise, miss-alignment between color and depth, combination of different viewpoints and meshing. The final mesh can be converted to a format that can be fed into the RepRap 3D printer.

We provide justifications for some of our major design choices such as meshing methods, combination, alignment. And we also assess the quality of our final design with respect to the revised requirements and find that the final design satisfies all our original requirements.

Appendix A provides details of our combination algorithm, appendix chap:meshsoft provides an overview of software that take pointclouds to meshes, and appendix C provides an overview on the alignment procedure dealing with color/depth mismatch.

We thank Orbis Mutual Funds for sponsoring this course and Prof. Jason Foster for his enthusiasm and helpful suggestions throughout this course.

Our code is hosted at `http://code.google.com/p/kinnectproj/`.

Contents

Chapter 1

The Design

1.1 Overview of the design

The design can be divided into a few main parts, corresponding to each section in this chapter. A basic description is given here, with more details to follow in the respective chapters. See figure 1.2 for a more detailed flowchart.

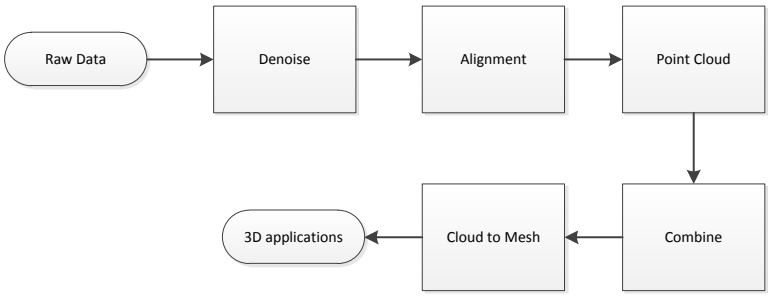

Figure 1.1: simple overview of this design, a more detailed flowchart is on figure 1.2.

First, images are taken from different viewpoints of some object. The sensor image first goes through a few preprocessing steps so it has less noise than raw sensor inputs. The denoised images then go through an alignment procedure which aligns depth images with color images. The aligned image gets converted to a point cloud in world-coordinates. Point clouds from adjacent viewpoints are

combined pairwise into a single point cloud. And all point clouds eventually get combined together to form a single point cloud model for the whole object. Point cloud can be converted to a mesh by sampling and adding edges as well as faces. Lastly, the RepRap host software can take the mesh and print the object scanned.

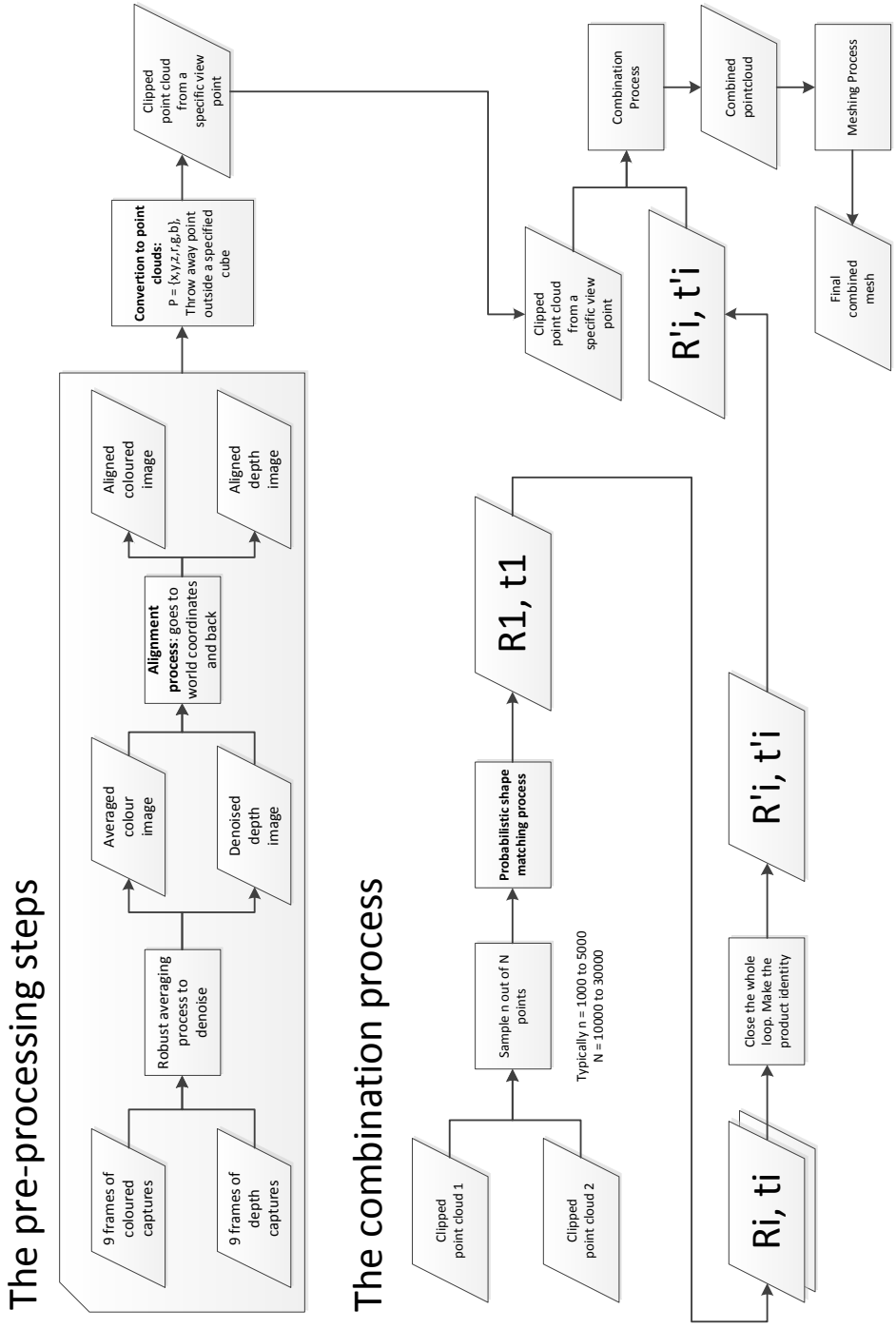

Figure 1.2: Flowchart overview of this design

1.2 Preprocessing steps

Raw depth input from the sensor is very noisy (See figure 1.3). However, because the structured light used to get depth image is randomly projected, noise in each depth frame is different from another. It is therefore possible to take advantage of multiple captures and using all those information to denoise the image.

We take the following approach called weighted robust averaging. If the sensor returns black, meaning the depth is unknown, then the value is not averaged into the final image; otherwise, the final depth value is the average of several frames. To be more precise, let $d^1, d^2, ..., d^9 \in \mathbb{R}^{640\times480}$ be 9 consecutive frames. Let $W^1, W^2, ..., W^9 \in \mathbb{R}^{640\times480}$ be the corresponding weight images. So $W^k_{ij} = 0$ if $d^k_{ij} =$ `NoReading` and $W^k_{ij} = 1$ otherwise. Define $\odot$ as the pixel-wise multiplication operations. Then the final output D is:

$$D = \frac{\sum_{i=0}^{9} W^i \odot d^i}{\sum_{i=0}^{9} W^i}$$

Where the division is also pixel-wise.

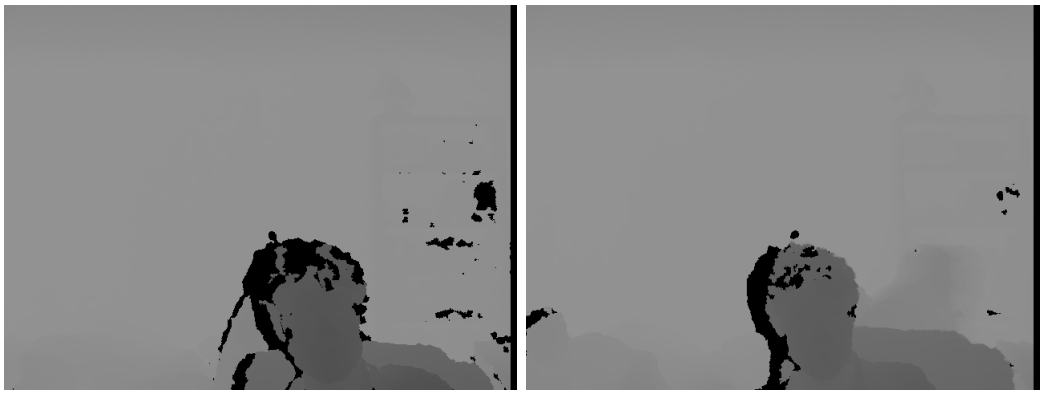

Figure 1.3: The sensor input. **left image**: sensor input before robust averaging **right image**: after robust averaging

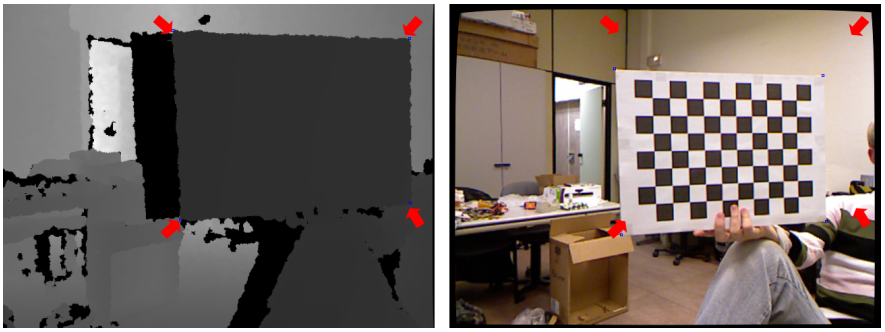

Figure 1.4: Miss-aligned image. The red arrows are miss-aligned by a large amount on the color and depth images [1].

1.3 Alignment

1.3.1 The problem

This problem is due to the way the Kinect sensors works. Kinect uses different camera at different locations for color and depth. Furthermore, the infrared laser projector is at yet a different location from depth, and color cameras. So miss-alignment is to be expected. See figure 1.4 for a demonstration of this problem [1].

1.3.2 The solution

Since we would like to build a 3D colored model, we have to solve this problem. The major objectives of this component are two folds:

- Take a depth image and construct the corresponding 3D scenery in a Cartesian coordinate system. The coordinates are in meters.
- Find the mapping between a pixel in the depth image with another pixel in the color image.

The solution to those two problems are well-studied within the Kinect community, and a solution used in this project is outlined in C with greater detail.

1.4 The point cloud

The input from the Kinect is a 2D array of depth values and color measures, but the output PLY file format as well as the combine algorithms in the next section works on point clouds. So it is necessary to convert the 2D arrays to a point cloud form. The conversion is done in the constructor of the pointcloud class. In essence, it simply goes over the 2-D array point by point and applies the alignment algorithm mentioned in previous section. The code is optimized for performance by using matrix operations instead of loops, so it imposes no overhead for the program. After conversion, the data are stored into two lists: one for vertex coordinates and another for its color. The two lists share the same index. The reason for separating the coordinates and color is because the combination algorithms only need to use the coordinate values. This implementation makes the list more manipulable.

Other than the constructor, three more methods are defined for this class: addition, clipping, and PLY output. The addition operation is overloaded to concatenate two objects together, which is done by concatenating all internal matrices. A clipping method is defined to cut off extraneous points around the object. The input variables defines the 6 planes that forms a cube, and the method will search in the point cloud and delete all points that are outside the cube. The code is written so that clipping on one side a time is also possible. The PLY output method follows the PLY file format standard by first printing out the headers and then a list of all points in the point cloud. The coordinates are scaled 1000 times when output, because the internal unit used in this project is meter, while the RepRap software used in 3D printing assumes all coordinates in millimeters ([2]). The PLY format specification is unit-less (any unit can be used for the coordinates) so this scaling will not affect the portability of the outputted PLY file.

1.5 Combination step

Once we have a bunch of point clouds, we can compare them pairwise and extract the transformation between them. We extract the rotation matrix $R \in \mathbb{R}^{3\times3}$ and

the translation vector $t \in \mathbb{R}^3$ from each pair of adjacent point clouds. The detailed algorithm is described in A, which is based on [3] and [4].

After all these transformations are extracted, each point cloud is modified by its respective transformation, and all point clouds are combined into a single point cloud in the world coordinate.

The combination algorithm works by simulated annealing on the variance σ^2 as EM steps are taken. With bigger variance, combination algorithm looks for rough matches, and with smaller variance, finer matches. An uniform component is also added to the Gaussian mixture in [3] to deal with outliers. The process stops when 20% of the data are explained by the uniform component. For more details on this algorithm we developed, see A. We also tried a Bayesian model for this task, also described in A, which unfortunately does not meet performance standards using Markov Chain Monte Carlo for inference.

There is one last hack used for combination. Because we do pairwise combination, errors accumulates so the last frame does not necessarily wrap back to the first frame. So we introduce an ad-hoc algorithm forcing all transformation matrices to multiply to identity. We use an iterative procedure.

$$P = R_1 R_2 ... R_n$$

We would like to modify each R_i by a small amount, to make $P = I$. This can be done using the following iterative procedure, for each i from 1 to n:

1. Set $P_i = R'^{-1}_{i-1} R'^{-1}_{i-2} ... R'^{-1}_1$ and $Q_i = R_i R_{i+1} ... R_n$
2. Set the correction term $D_i = Q_i^{-1} P_i = I + \Delta$. D should equal to I if no error accumulation, so Δ represent the error of how far from I we are.
3. Take the n^{th} root: $C = D^{1/n} \approx I + \Delta/n$
4. Set $R'_i = C R_i$
5. Multiply by R'^{-1}_i on both sides of the equation, and repeat.

Upon each iteration, the product P becomes more like identity.

1.6 Mesh

Existing 3D applications often support 3D models in mesh format. We use a third-party software MeshLab to convert a point cloud model to a triangular mesh model.

We first subsample the point cloud using the Poisson Sampling algorithm. This process forces all vertices to be uniformly distributed, while also eliminating the noisy data points. Next, we apply the Poisson Surface Construction operation to construct the 3D mesh. It is worth noting that the resulting mesh does not include color information at this point. In the final step, we run the Vertex Attribute Transfer operation to transfer the color information from the original point cloud onto the new mesh model. The transfer algorithm uses a simple closest point heuristic to match the points between the two models. Moreover, MeshLab also allows users to export all the operations described above into a single script file (.mlx). The script can be invoked using a shell script adhere to the specifications of MeshLabServer [5].

See B for a detailed comparison between alternative meshing softwares.

1.7 RepRap

Although RepRap indicated PLY as a recommended file format on its official site ([6]), the actual software can only read STL file and RepRap-specific (none-portable) RFO file. The STL must be in binary form (ASCII format will not load properly) and has unit of millimeter ([2]). The PLY file is already in unit of millimeter, so the only thing left to do is converting PLY format to STL (MeshLab is chosen to do this job simply because previous steps used it). It is important to note that STL does not support color, the color information is lost (the RepRap printer cannot print color anyways).

After the file has been converted to STL format, it can be loaded into the RepRap host program and used to generate the 3D object in the machine! See [7] for how to use RepRap host program to print parts. The printing is time consuming, for example, printing a toy bear requires about 11 hours.

1.8 Accessing our code, replicating the results

First, you would need the Mercurial source control system to access the up-to-date code. The code is hosted at http://code.google.com/p/kinnectproj/. A few original dataset and results can be found in the Downloads section.

If you would like to become a contributor, email one of us. Our email can be found in this project page.

Chapter 2

Justifications

2.1 Using Python as the programming language

2.1.1 Expressiveness

The process shown in the flowchart is quite complicated, and most steps in the process are not performance intensive. Therefore, we would like to be able to work in an environment where we can rapidly prototype both known algorithms and our ideas. Python is known to be much more expressive than other alternatives.

2.1.2 Interactivity

Since Python is an interpreted language, we are able to run our programs interactively, with the freedom of making modifications while running. This was a very helpful and commonly used technique during the design process. A common scenario goes like this:

- Part A of our program finished running for the first time
- We get to Part B of this program that was never executed before
- Part B has a bug in it
- We fix the bug and continue

This would not be possible with compiled language unless we save all our intermediates results in files, which introduces considerable overhead.

2.1.3 Numerical Package

Good OpenSource numerical and plotting packages like NumPy, SciPy and matplotlib are avalable for the Python language. This project involves tons of numerical computations.

2.1.4 Extensibility to GPU

If our core code are written in Python, we can easily substitute the NumPy package used by GNumPy. The code can then be ran on a GPU, which offers a speed improvement up to 30 times. GNumPy [8] is based on CUDAmat [9], which is in turn based on CUDA.

2.1.5 Familiarity, OpenSource and availability of support

Finally, there are possibly other languages that are expressive and have nice numerical packages such as MATLAB, C++, Java etc. Python is separated by the following important attributes.

- people in this team are familiar with Python
- Python is OpenSource and has a global standard
- many people use the language so it is widely supported

2.2 Using OpenKinect as driver

In the current market, there are two primary Kinect open source drivers: libfreenect and OpenNI. In order to choose the driver that best suits our needs, here is a list of requirements that the chosen Kinect driver must support:

- Python wrapper

- Control the tilting motor
- RGB data callback
- Depth data callback
- Run on Linux

Both drivers meet the requirements listed above, however at the time of our initial research, python wrapper was not available for openni. In terms of functionalities, OpenNI provides a rich set of middleware features such as full body analysis, gesture detection and hand point analysis that are outside of the scope of this project. Although, it can be helpful to have a rich set of external libraries, but it also affects the portability of the final software. Overall, libfreenect was chosen over OpenNI, for its simple design and its support for Python [10; 11].

2.3 Meshing

The general problem of converting a point cloud to a mesh model is thoroughly studied in the field of computer graphics [12] [13], as there is a number of third-party software available that we could use to perform this operation. In order to choose which application is the most suitable for this project, here are the requirements and criteria that we used for our selection process:

- Performance: It is critical that the final mesh closely resemble to the input point cloud in terms of physical dimensions and also color coating. Moreover, there are some noisy data from the point cloud such as layer overlaps that the software should adjust and ignore.
- Time: The whole process time should be fast and comparable with the timing of the other components. We dont want this step to be the bottleneck of the overall flow.
- Cost: The whole aim of this project is to make a cheap 3D scanner. Hence, if free open source software can handle the job reasonably well, we dont want to waste money on commercial software.

- Automation: The whole process should be automated. Ideally, we want to write a shell script to execute the whole operation. The script specifies the input file in a point cloud form and the output should a 3D model file in the corresponding mesh form.

- Platform: So far, all the implementations are done on Ubuntu Linux machines. Hence, Linux based programs are preferred.

  See B for a detailed comparison between different meshing softwares. MeshLab was chosen as the best fit for this project.

2.4 Using the PLY file format

The RepRap official website provided three recommended file formats for representing a 3D object: STL, PLY, and COLLADA [6].

The STL (StereoLithography) format is a commonly used file format for 3D prototyping. A STL file is simply a list of triangles that the object is composed of. No colour or texture information is stored. [14].

The PLY polygon file format is a simple portable file format between different design tools. It breaks the 3D object into multiple polygons, and stored the information of each polygon in a list. On top of the vertices and surface normal vectors that are supported by STL format, PLY format can also store colour, transparency, texture information, and even user customized variables.

The COLLADA file format is intended for lossless information conversion between different 3D design software. It defines a XML schema for all object attributes across multiple 3D design programs and a COLLADA file is simply a XML file that follows the schema. It supports much more properties that a PLY file does, such as curves and animation [15].

All three file formats are supported by MeshLab and other major graphic processing programs, so they are all portable formats. Because a triangle is basically a 3-edge polygon, the PLY format is a superset of STL format. PLY can store all required information specified in the requirement, while STL is missing the color information. As a result, PLY format is preferred instead of STL. The COLLADA format is much more complicated than PLY but the extra

information it supports is beyond the scope of this project. Since PLY already provided all required information, there is no need to use the COLLADA format which will make the programming work much harder to implement. As a result, the PLY file format was chosen as the official output format [16].

2.5 Combination Method

A large part of the combination algorithm is based on published works [3; 4]. After making several important modifications inspired by [4] and based on [3], the algorithm generally works well. Runtime can still be quite long, and scales quadratically with the number of points sampled. The final algorithm takes between 10 minutes (for 1000 points sampled) and 5 hours (for 5000 points sampled) on a i7 2.5GHz computer. This amount of time is negligible compared to the time needed to print the object (11 hours on RepRap). On the other hand, the algorithm is highly parallelizable, and can almost be made n times faster if ran on n processors. The exact algorithm published also had similar performance characteristics.

We also tried a Bayesian model using Markov Chain Monte Carlo for inference, the performance is even worse. See Appendix A for more information.

Chapter 3

Assessments

In this section, we will use the set of requirements and criteria outlined in the Revised Requirements document as the metrics to evaluate the quality of our project from different aspects.

3.1 Assessment against requirements

3.1.1 The software must be compatible with any Kinect device

The constants described in section Alignment works pretty well with all three Kinect devices that we have purchased for this project. Moreover, there is also a way to derive those constants for an individual Kinect to optimize the performance [17].

3.1.2 Produce a 360 degrees 3D model

The final output 3D model delivers a 360 degrees view. Moreover, the merging heuristic makes no explicit assumptions about sequential frames. Hence, we can also take two separate frames for both the top and the bottom of the scanning object and have them merged to the final model as well.

3.1.3 Display only the scanning object

In order to clip unwanted scenes from the object, we construct a magic cube based on a set of parameters. All surroundings outside of the cube are ignored. This means that the user has to estimate both the size of the object and its distance away from Kinect before conducting the clipping operation. The parameters can be reused if all scanning objects are of similar size and the distance between Kinect and the object is relatively constant.

3.1.4 Output to standard 3D format

The final output file is in the PLY format, which is a standard 3D format designed to store 3D data from scanners [15].

3.2 Assessment against Criteria

3.2.1 Technical Criteria

3.2.1.1 Filter input noise

To eliminate the amount of noise, a number of measures have been taking. First, we use the robust weighted averaging method to achieve a more stable depth image. In addition, the clipping process removes the surrounding from the real object. The merging heuristics also assume a certain amount of outliers (20%). Finally, the Poisson Sampling algorithm enforces an even vertex distribution throughout the whole model. Overall, as illustrated in the figures shown previously, the final output model does an excellent job in regards to noise filter. Color mapping: In general, the color mapping error is relatively small, within a few pixels. The main bottleneck for this criterion is Kinect itself, because the color images only have a resolution of 640x480 pixels. At the moment, the color coating can be blurry on 3D models. However, we expect the color quality to be improved if more data points are provided from Kinect. Computation time: The major bottleneck in terms of computation time is the merge component. The algorithm has a complexity of N^2. If we use a sample size of 1000 points, this translates to a running time of roughly 10 minutes. Depending on the complexity

of the object, 1000 sample points may be enough for simple objects such a box, but not sufficient for more sophisticated objects such as a real person. Hence, the computation time is acceptable for the purpose of 3D printing. However, the speed is insufficient for fast response applications such as 3D video streaming.

3.2.1.2 Adhere to software design pattern

The entire program is comprised of python code, shell scripts, and MeshLab scripts. During the implementation stage, we tried our best to comment all changes and also avoid hardcoded programs to deliver a generic implementation. Moreover, there is an online code repository that all team members share [18]. This allows all members to synchronize to the latest work and also provides a safety mechanism to backup all code onto the cloud server.

3.2.1.3 The output format to be supported by other existing applications

The output model is of type PLY, which is supported by a variety of 3D applications such as Maya and RepRap.

3.2.1.4 Relative accuracy

The relative accuracy between the real object and its 3D model is within a few centimeters. This variation is mainly due to rounding errors when performing the point cloud to mesh conversion.

3.2.1.5 Handle complex objects

The objects tested so far are a drawer, a soy sauce bottle, a box, a bear, and a real person. The outcome is relatively accurate for the first 4 objects. The algorithm performs well against both convex and concave objects. The soy sauce bottle had a hole around the handle; this void space was also reflected in the output model. In the case of a real person, although the general 3D shape resembled to the actual person, the color mapping was poorly constructed. However, since the person was rotating himself in order to capture image frames from all 360

degrees, the input data are less accurate than the other static objects. Overall, we believe the scanner is fairly accurate against household items.

3.2.1.6 Scan range

The software doesnt impose further physical restrictions in terms of the scanning range; the range fully depends on the camera limitations on Kinect.

3.2.2 Application Criteria

The software should not be restricted to a specific application. If a particular application is very appealing, but requires a hardware upgrade from the current Kinect, we would still like to demonstrate its feasibility although the application may not necessarily be practical at the present moment. Below are three possible applications that we purposed in the initial requirement document:

3.2.2.1 3D printing

3D printing is a time consuming application, for the computation time is non-critical in this case. However, the printing job may require a high relative accuracy with respect to the real object. Depending on the size of the object, a precision in the order of centimeters may or may not fulfill the accuracy requirement.

3.2.2.2 3D Animation

This application requires low relative accuracy and no specific constraint for the computation time. It is a good match to the current state of the project.

3.2.2.3 3D live stream

While accuracy is not very important in live streaming, the output has to be delivered instantaneously. The current running time takes over 10 minutes, however it is possible to improve the performance dramatically with the help of 100+ core GPUs.

3.2.3 Economic Criteria

The major objective of this project is to build a 3D scanner using Kinect as cheap as possible. A brand new Kinect costs $149.99 at any BestBuy local store [19]. Initially, we planned to build a rotation apparatus, such that the scanning object can be placed on it to rotate around. However, the merging algorithm doesnt assume all captured frames to have the same rotational axis; hence there is no need for such apparatus. Moreover, both OpenKinect and MeshLab are open sources project that are freely available to the general public. As result, the total cost of building our Kinect 3D scanner is equivalent to the cost of Kinect itself.

Appendix A

Combination Algorithm

A.1 Introduction

Readers are assumed to understand basic mixture models and EM algorithm in this appendix. See relevant chapters in Bishop's text [20].

A.1.1 The problem

We would like to reconstruct 3D models of objects from images and depth maps of the object from different viewpoints. The input data is shown in figure A.1.

These inputs are then converted to a point cloud. A point cloud P is a set of points $P = \{p_i\}_{i=0}^{M}$. Where M is the number of points in this cloud, and $p_i = \{x, y, z, \mathbf{a}\}$, contains the spatial coordinates of a point and some attributes $\mathbf{a}$ of this point such as its color and other properties.

A point cloud corresponding to figure A.1, with colors is shown in figure A.2.

We would like to build a full 3d reconstruction of an object from such point clouds taken in different viewpoints. Now the problem is combining these point clouds - with each point cloud having considerate overlap with its neighboring point clouds. However, each pair of adjacent point clouds is related by some unknown transformation. We require this transformation to be a rotation, plus translation. The goal is to find the rotation matrix $R \in^3 \mathbb{R}^3$ and translation vector $t \in \mathbb{R}^3$.

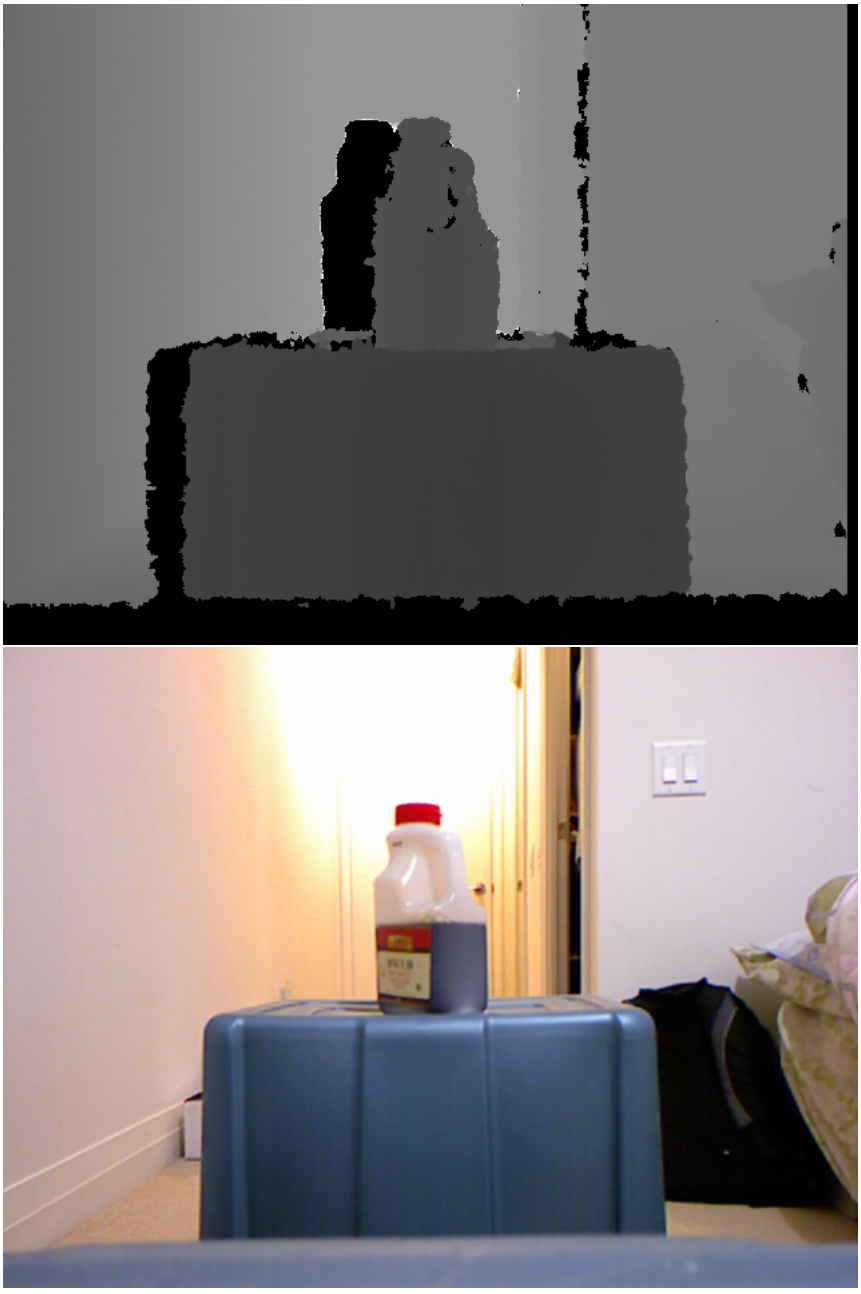

Figure A.1: The sensor input. **top image**: depth map, whiter color means farther away, pure black means the value is unknown due to sensor limitations (shadow of projected light used for depth detection. The Kinect sensor uses the structured light method for depth capturing.). **bottom image**: the corresponding RGB map

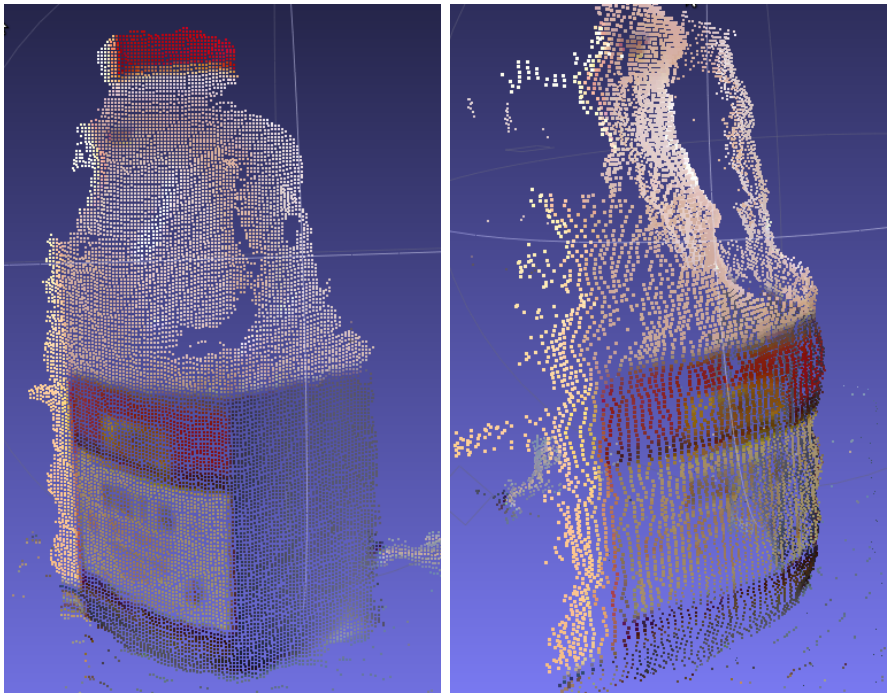

Figure A.2: Point clouds with color, seen from two different viewpoints of the above capture. This is obtained with some alignment and other processing steps, from the data shown in figure A.1. But the process is straight-forward, just takes some work.

A.1.2 The Methods

We quickly outline our approaches here, with a lot more details in the sections to follow. The problem can be formulated somewhat probabilistically, We do realize that the formulation is not the most natural. Also note that colors and other attributes are completely ignored, so we only deal with shapes here. One approach is EM, and the other is MCMC on a Bayesian model.

Basically, we could treat each point in some point cloud as a mixture component in a Gaussian mixture model (GMM) with mean equal to location of that point, and some variance σ^2. Then we can try to maximize the probability of this GMM generating some adjacent point cloud as data. Notice each point cloud is rigid, so the mean and variance of a GMM cannot be changed freely as in the regular EM method for GMMs. Instead, we rotate, translate all means of each mixture component by a specific rotation matrix $R \in^3 \mathbb{R}^3$ and translation vector $t \in \mathbb{R}^3$. To a first approximation, if we can maximize the likelihood, with respect to R and t, of generating an adjacent point cloud, then the R and t which maximizes the likelihood can be treated as the solution. Similarly, a vague prior can also be added to R and t since the object might be rotated in a somewhat regular way, and we may incorporate our prior belief into this framework. Slightly more complexity arises from having to deal with systematic outliers, since adjacent point clouds are not completely the same, but just have considerate overlap (say 80% overlap). Note that outliers here include both measurement errors, as well as the systematic outliers due to non-overlapping parts.

A.2 A probabilistic formulation

A.2.1 Basic formulation

Given two point clouds, P and Q, where $P = \{p_1, p_2, ..., p_M\}$ and $Q = \{q_1, q_2, ..., q_N\}$. Each point in the point clouds, $p_i = \{x_i, y_i, z_i, \mathbf{a}\}$, $q_i = \{x'_i, y'_i, z'_i, \mathbf{a}\}$ where the attributes $\mathbf{a}$ is completely ignored in what to follow except when rendering the final images (so color information. inside the attributes vector, is NOT used at all, although color does contain a lot of potential information). From now on, we drop the attributes term and treat elements in the point cloud as 3d vectors.

we treat points in point cloud P as data, and points in point cloud Q as mixture components.

Let $Q' = \{q'_1, q'_2, ..., q'_N\}$ where $q'_i = Rq_i + t$. Now the task is to find rotation matrix $R \in^3 \mathbb{R}^3$ and translation vector $t \in \mathbb{R}^3$, so that the likelihood of generating the data points P under Gaussian mixture components with means Q' and variance σ^2. It is sensible for this task to use the same variance in all directions and for all components, or at least use the same specific prior of variance for all components. we will also use the same mixing proportions. Think of it as an "cloud" of uniform thickness so that its shape should match another cloud. The variance should not vary significantly as in usual GMMs.

The basic framework is presented in the coherent point drift paper [4], and an EM algorithm specific for this application is presented in [3]. The EM algorithm we used also does annealing and accounts for outliers, which are not in the second paper. But we refer to these two papers without providing excessive details on the EM approach, except a few additions we made that are not mentioned in the paper.

A.2.2 Our additions to EM

GMM is very sensitive to outliers, especially with small variance. So it is sensible to add a single component with large variance, or just a uniform component to deal with all these outliers.

Simulated annealing is also a sensible addition to the EM algorithm. σ can be annealed, so it matches big features first and focus on precision later on. For the EM approach, we anneal σ until 20% of the data is explained by the outlier component and 80% are explained by the Gaussian components.

A.3 The EM approach

The EM approach can be summarized as follows. We refer readers to [3] and [4] for more details.

Repeat until $\sigma < \sigma_{stop}$ or when 20% of data are explained by the uniform component

- **E-step**: Evaluate the $M \times N$ responsibility matrix
- **M-step**:Use the responsibility matrix to evaluate R and t. Force R to have unit determinant by singular value decomposition
- $\sigma = \sigma \times \alpha$

Where $\alpha = 0.9 \sim 0.98$ is the annealing rate.

A.4 Some experiments

A.4.1 A 2D experiment first

we first generate artificial data with unknown R and t. The algorithm was able to recover R and t to within 1% error in component magnitudes. The true translation is $t = [1, 0.5, 0.2]$, and the recovered translation is $t = [1.00466532, 0.50431795, 0.20684235]$. Performance on other fake datasets are similar. Some samples are shown in figures A.3 and A.4 with an entire "arm" of outliers. Other experiments yielded similar results.

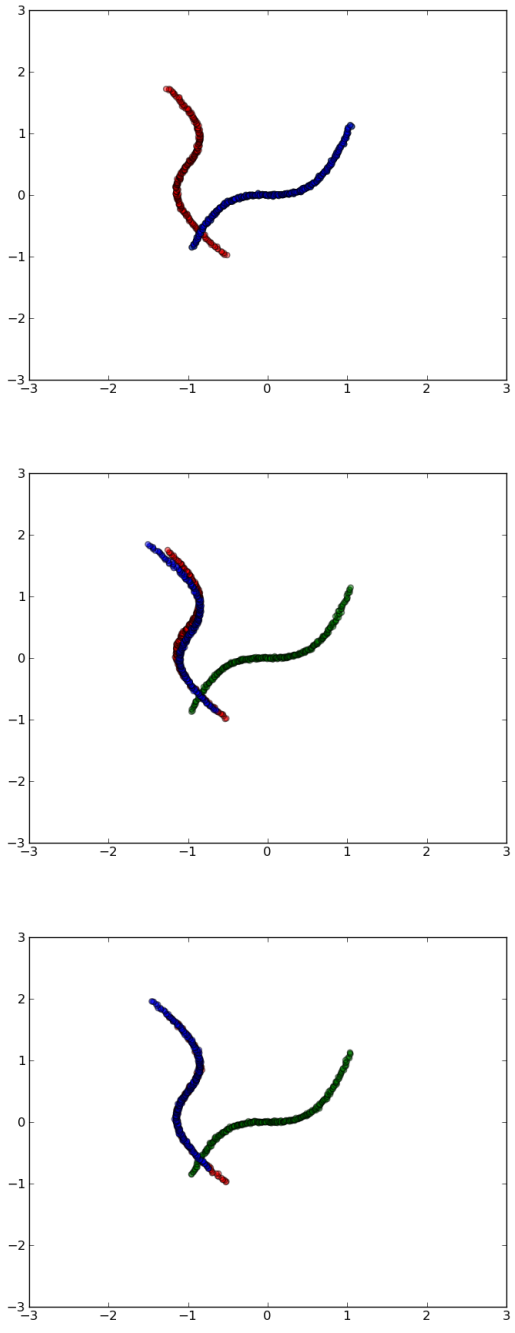

Figure A.3: Matching cubic curves. Green scatter shows the original mixing components location. Red scatter shows the data location. Blue scatter shows the new mixing component location

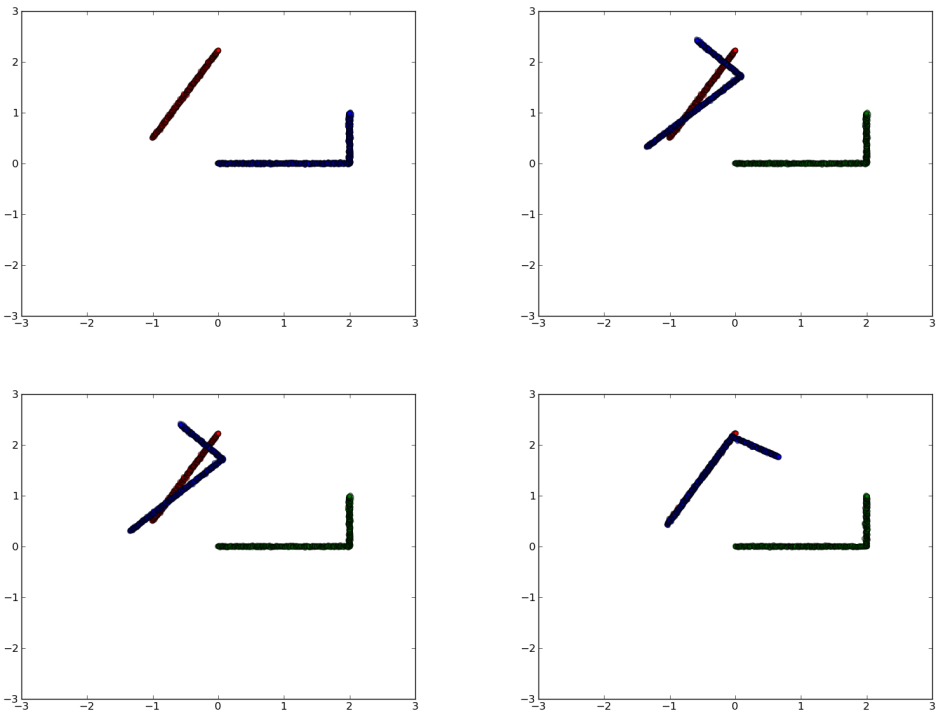

Figure A.4: Matching cubic curves. Green scatter shows the original mixing components location. Red scatter shows the data location. Blue scatter shows the new mixing component location

A.4.2 3D reconstruction

Then we sample points and apply this method to the 3D soysauce dataset. The combination result is shown in figure A.5.

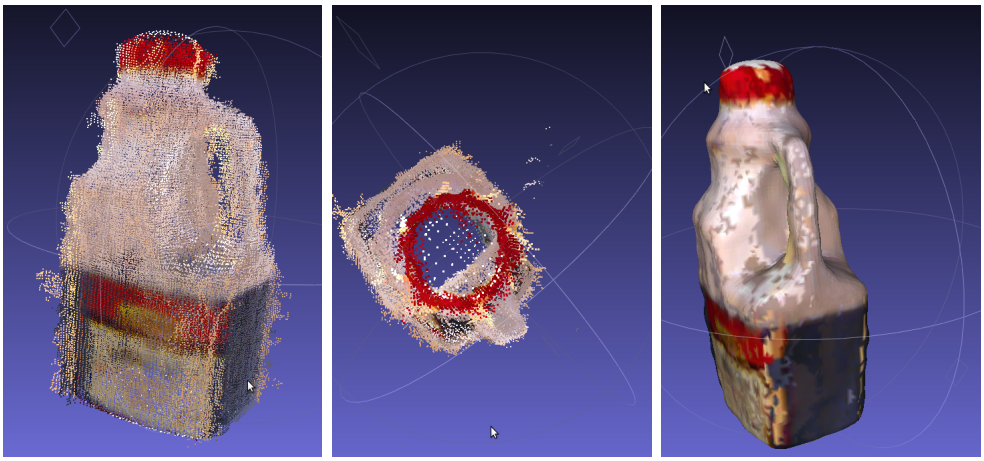

Figure A.5: Matching real pointclouds. Left is the combined pointcloud, with many visible outliers. Middle is the topview. Right is after meshing the pointclouds

A.5 The Bayesian approach

Readers are assumed to have some background in Bayesian inference and MCMC methods to understand this section.

The application here may not be the most natural application for MCMC methods, since a point estimate is required in the end and it is sensible for this estimate to be the mode instead of the posterior average. So simple MAP actually does seem somewhat more natural, but we would like to deal with outliers in a systematic way, and use prior information effectively. These complications make the model intractable so I use MCMC. I am also interested in comparing the performance of MCMC vs. EM in this task. The EM algorithm presented in [3] did not work well at all, while my customized EM method with annealing seems rather inefficient.

A.5.1 The model

Recall we have two pointclouds, P and Q, where $P = \{p_1, p_2, ..., p_M\}$ and $Q = \{q_1, q_2, ..., q_N\}$. Set $Q' = \{q'_1, q'_2, ..., q'_N\}$ where $q'_i = Rq_i + t$. We treat P as the data pointcloud, and Q as the mixture component pointcloud. Mixing proportions under the outliers GMM are constants, and sums to a total of $\Pi = (m+a)/(M+a+b)$ and Π/M each, where $m = \sum_i o_i$ a, b are constants for the beta prior for outlier indicators. Once a point is labeled outlier then it comes from an uniform component, with mixing proportion $1-\Pi$. Alternatively, we may also use a GMM with larger variance to allow for softer assignments.

So, the simpler model (model A) is as follows:

$$
\begin{aligned}
p_i &\sim \begin{cases} \texttt{GMM}(Q', \sigma_0^2) & : o_i = 0 \\ \texttt{Uniform}(-l, l) & : o_i = 1 \end{cases} \\
o_i|\theta &\sim \texttt{Bernoulli}(\theta) \\
\theta &\sim \texttt{Beta}(a = 4, b = 16) \\
\log(\sigma_0) &\sim \mathcal{N}(-4, 0.1) \\
R &\sim \texttt{Uniform}(\texttt{rotationmatrices}) \\
t &\sim \texttt{Uniform}(-l, l).
\end{aligned}
$$

The slightly more complicated version is:

$$
\begin{aligned}
p_i &\sim \begin{cases} \texttt{GMM}(Q', \sigma_0^2) & : o_i = 1 \\ \texttt{GMM}(Q', \sigma_1^2) & : o_i = 0 \end{cases} \\
o_i|\theta &\sim \texttt{Bernoulli}(\theta) \\
\theta &\sim \texttt{Beta}(a = 4, b = 16) \\
\log(\sigma_0) &\sim \mathcal{N}(-4, 0.1) \\
\log(\sigma_1) &\sim \mathcal{N}(-2, 1) \\
R &\sim \mathcal{N}(\texttt{rotationmatrices}, R_0, \sigma_R) \\
t &\sim \texttt{Uniform}(t_0, \sigma_t).
\end{aligned}
$$

Problem domain knowledge is built into the prior. In this case, the object we are interested in have roughly a radius of decimeters. As a result, $\mathbb{E}[\log(\sigma_1)] = -2$. The feature size that should be matched are roughly centimeters, or even sub-centimeter, so $\mathbb{E}[\log(\sigma_0)] = -4$ with small variance. These values come from my beliefs, but arbitrarily specifying the prior means of σ_0, σ_1 in log domain is much better than arbitrarily specifying σ_0, σ_1 themselves. We use $\texttt{Uniform}(-l, l)$ to mean an uniform distribution that has range $-l$ to l in all dimensions.

In the simple case, R and t, can be assumed to be have uniform distributions over its domain. Since R is a rotation matrix, it really only has 3 degrees of freedom, instead of 9, which is its dimension. we do random walk first, to get $R' = R + e$ and then do SVD to get $R = UCV$, and finally get the new R as UV. For the actual application, we might have a rather good idea of what R and t might be, although we do not know them exactly, and this can be incorporated into the prior for R and t. This is another attraction of the Bayesian approach, as such information is rather important and cannot be naturally incorporated into the EM algorithm.

A.5.2 The updates

So the state space consists of $R, t, \sigma_0, \sigma_1, o_1, ..., o_M$ with θ integrated out.

We use Metropolis-Hasting updates for this task. Specifically, we just use Metropolis updates for R, t, σ_0 and σ_1 with a normally distributed step. On the other hand, the outlier indicator variables could benefit from some heuristics. Since the so-called outliers here are actually systematic, it is conceivable to propose according to neighboring components as well as fit. We will start with just proposing the opposite value and then look into the more complicated proposal method later. The probability of proposing a datapoint to be outlier can be the fraction of its nearest r neighbors that are also outliers.

Because this is a mixture model. There is also significant potential for computation savings in updating the outlier indicators. Only the change in log likelihood of element i needs to be evaluated without looking at all the others, which remain unchanged. The sum of the mixing proportions from inlier and outlier components then need to be saved to get this computation saving.

We only use the simple model above, and proposing opposite values for outlier indicators in this project.

Appendix B

Comparison between different meshing softwares

We compared 3 different meshing softwares, and conclude Meshlab is the best.

B.1 CloudMesh-0.1x

CloudMesh is an open source software that is readily available on source forge [21]. We were able to download the full source code. Unfortunately, the program was developed using Visual Studio, this means that the application can only run natively on Windows machines. Moreover, the program only supports the .off file format, which only includes 3D vertices with no colors. Lastly, the performance is very weak against noisy point clouds. As shown in figure B.1, the exterior surface of the soy sauce bottle is very rough. In addition, there should be an empty gap around the handle; however the gap is filled up with undesired meshes.

B.2 VRMesh v6.0 Studio

VRMesh is a commercial software that specializes in point cloud to mesh conversion. We were able to test the demo version of the software, which includes a set of useful features. The overall performance is impressive; the operation time ranges between few seconds to a couple minutes. However, the single-user license

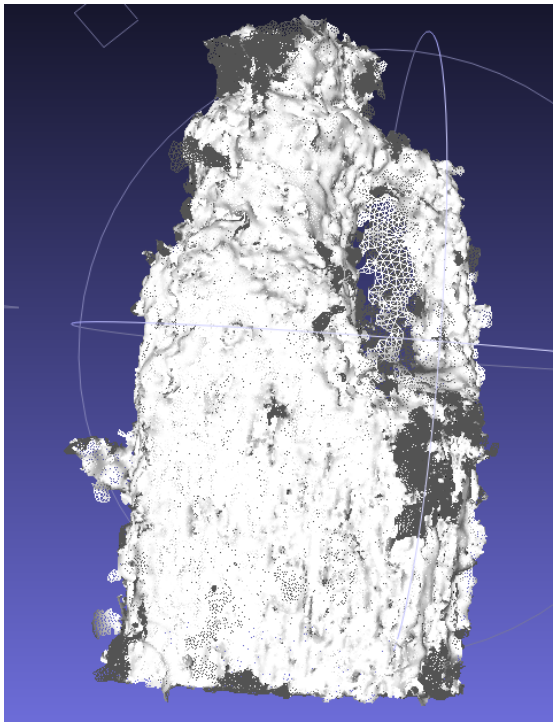

Figure B.1: The mesh model of the soy sauce bottle using CloudMesh.

costs $2995 [22], which is outside of our budget range.

B.3 Meshlab v1.3.0

MeshLab is an open source, portable, and extensible system for the processing and editing of unstructured 3D triangular meshes [23]. The software supports all major OS platforms: Linux, Windows and MacOSX. The source code can be readily downloaded on its source forge website. The interface is quite simple to use, as there is a step-by-step instruction guide that provide all the details to convert a point cloud to a mesh model [24]. Depending on the size of the point cloud, the operation time varies between a few seconds to a minute. The conversion is fairly accurate, as illustrated in figure B.2. We first subsample the point cloud using the Poisson Sampling algorithm. This process forces all vertices to be uniformly distributed, while also eliminating the noisy data points. Next, we apply the Poisson Surface Construction operation to construct the 3D mesh.

It is worth noting that the resulting mesh does not include color information at this point. In the final step, we run the Vertex Attribute Transfer operation to transfer the color information from the original point cloud onto the new mesh model. The transfer algorithm uses a simple closest point heuristic to match the points between the two models.

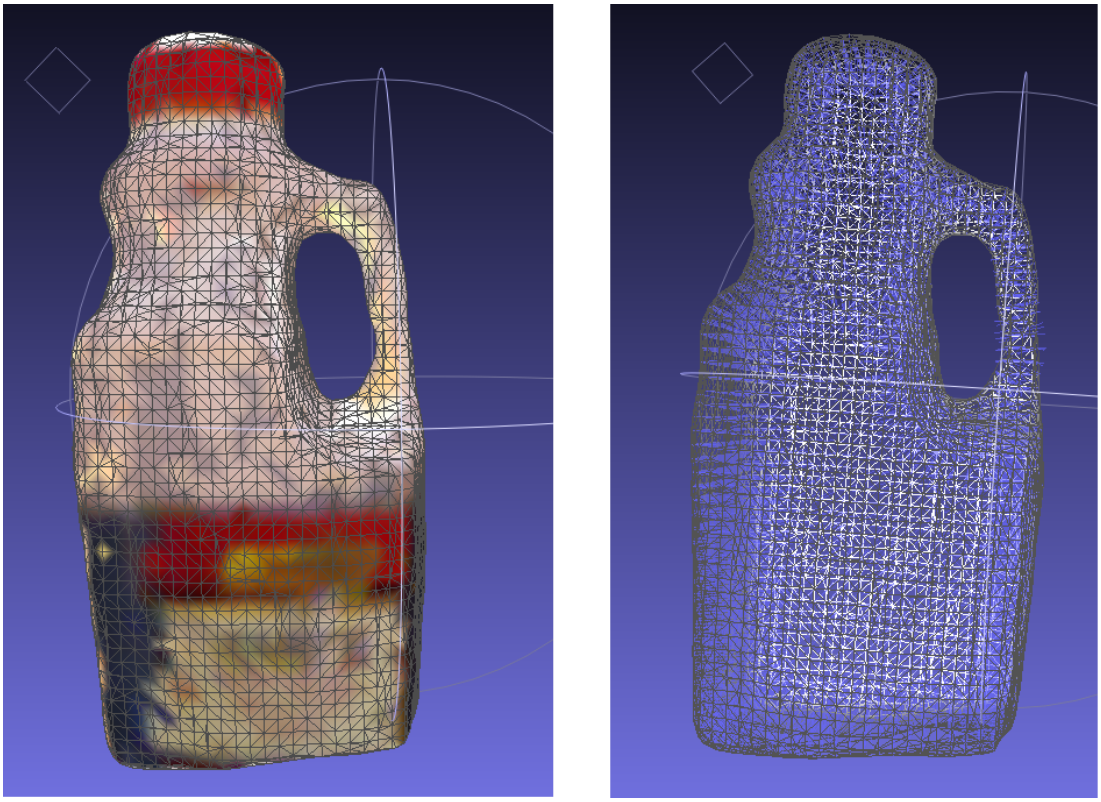

Figure B.2: Output results using MeshLab

Moreover, MeshLab also allows users to export all the operations described above into a single script file (.mlx). The script can be invoked using a shell script adhere to the specifications of MeshLabServer [10].

Overall, we believe MeshLab is the perfect software to use for this project, as it fulfills all the requirements that we discussed.

Appendix C

Alignment Methods

The major objectives of this component are two folds:

- Take a depth image and construct the corresponding 3D scenery in a Cartesian coordinate system. The coordinates are in meters.
- Find the mapping between a pixel in the depth image with another pixel in the color image.

C.1 Depth coordinates to world coordinates

By default, the depth images are 640x480 pixel arrays with each pixel having a depth value between 0 and 2047. It is easy to construct a greyscale image of such array as shown in figure C.1. In the figure, a darker coloured pixel represents a spot location nearer to the depth camera, while a brighter pixel locates farther to the camera. Moreover, the black regions are areas that the camera cannot see given the shooting angle.

Given that in greyscale image, black is defined with a value of 0 and white is defined with a value of 256 [25], we can see that there is an increasing relationship between the depth value and the real distance. Indeed, there exists a linear relationship between the depth measurement and its inverse distance to the camera.

The data points were collected experimentally [26] and are showcased in C.2. It is worth noting all experimental data discussed in this section were not collected

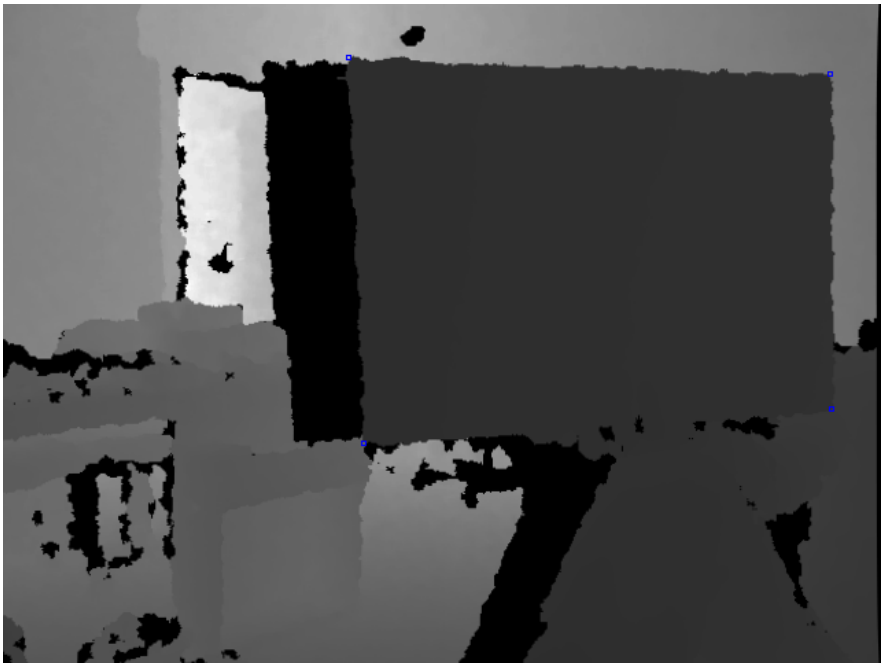

Figure C.1: Greyscale image of a depth array

by us; however, we did run a series of sample tests. Our findings did match closely with the claims. Now that we know the z-axis value of our world coordinates. To find both x-axis and y-axis values, it is just a matter of image projection using the formula listed below:

```
P3D.x = (x_d - cx_d) * P3D.z / fx_d
P3D.y = (y_d - cy_d) * P3D.z / fy_d
P3D.z = depth(x_d, y_d)
```

A point cloud is a set of vertices in a three-dimensional coordinate system. If we take each pixel on the depth image and convert each of them to its perspective world coordinate, the point cloud is thus constructed.

C.2 Color and Depth Mapping

At this stage, the point cloud only contains vertices with no color. The next step is to add RGB values to each of those vertices. In order to do, we must map each vertex with its corresponding pixel on the color image.

Distance [m]	Reading	Inv. Distance [1/m]	Power
0.60	544.0	1.6667	-1
0.80	681.0	1.2500	-1
1.00	756.0	1.0000	-1
1.20	810.0	0.8333	-1
1.40	850.0	0.7143	-1
1.60	878.0	0.6250	-1
1.80	902.5	0.5556	-1
2.00	921.5	0.5000	-1
2.20	937.0	0.4545	-1
2.40	949.5	0.4167	-1
2.60	959.5	0.3846	-1
2.80	969.5	0.3571	-1
3.00	978.0	0.3333	-1
3.50	993.0	0.2857	-1
4.00	1005.0	0.2500	-1

Figure C.2: Relationship between depth measurements and inverse distances

As illustrated on figure C.4, both the color image and the depth image are taken simultaneously. We can choose the four corners of the check board as feature points (marked with red arrows) to analyze the mapping relationship. Contrary to common sense, the mapping relationship is non-linear. The displacement between the color camera and the depth camera implies an affine transformation between the two images in both rotation and translation. Here are the formulas that we used in our implementation [1]:

```
P3D'=R.P3D + T
P2D_rgb.x=(P3D'.x*fx_rgb / P3D'.z) + cx_rgb
P2D_rgb.y=(P3D'.y*fy_rgb / P3D'.z) + cy_rgb
```

R and T represent the rotational and the translational matrices respectively, while fx_rgb, fy_rgb, cx_rgb and cy_rgb are intrinsic values associated with the Kinect device. Nicolas Burrus, a PhD student in computer vision did significant

constant	value	details
fx_d	1.0 / 5.9421434211923247E+02	scale factor in x-axis [1/pixel]
fy_d	1.0 / 5.9104053696870778E+02	scale factor in y-axis [1/pixel]
cx_d	339	camera x-position is a little bit to the right to the center of the depth image, given that 640/2=320
cy_d	243	camera y-position is a roughly in the center of the depth image, given that 480/2=240

Figure C.3: Constants used for the conversion to world coordinates

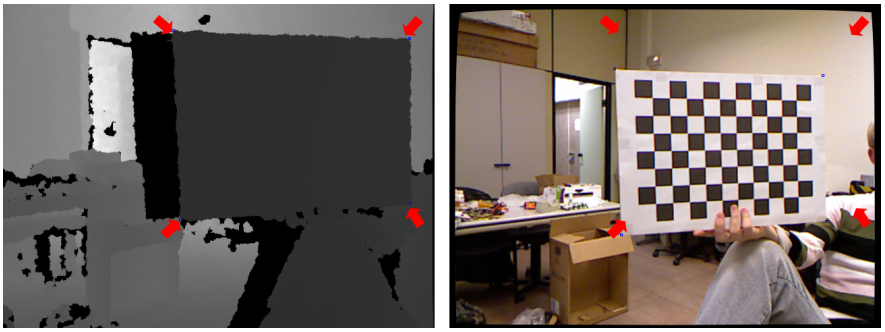

Figure C.4: Same checker board on both the depth image and the color image

works to derive those constants. We took the values that he purposed and ran a number of sample tests with different objects. The formula works genuinely well with small deviations. Accordingly, we modified some of the values slightly to introduce a better fitting for our own Kinect.

References

[1] N. Burrus, "Kinect calibration, http://nicolas.burrus.name/index.php/Research/KinectCalibration," April 2011. 5, 37

[2] "Section 1, http://reprap.org/wiki/AoI," Mar. 2011. 6, 8

[3] Y. Cui, S. Schuon, D. Chan, S. Thrun, and C. Theobalt, "3D shape scanning with a time-of-flight camera," in *CVPR*, pp. 1173–1180, IEEE, 2010. 7, 14, 24, 28

[4] A. Myronenko, X. Song, and M. Carreira-Perpinan, "Non-rigid point set registration: Coherent point drift," in *Advances in Neural Information Processing Systems 19* (B. Schölkopf, J. Platt, and T. Hoffman, eds.), Cambridge, MA: MIT Press, 2007. 7, 14, 24

[5] "Meshlab v1.3a, http://btbtracks-rbr.foroactivo.com/t538-meshlab_v123a," April 2011. 8

[6] "Reprap, http://reprap.org/wiki/Recommended_File_Formats," Mar. 2011. 8, 13

[7] "Reprap procedure, http://reprap.org/wiki/Mendel_User_Manual:_Host_Software," Mar. 2011. 8

[8] T. Tieleman, "Gnumpy: an easy way to use GPU boards in Python," Tech. Rep. UTML TR 2010-002, University of Toronto, Department of Computer Science, 2010. 11

[9] V. Mnih, "Cudamat: a CUDA-based matrix class for python," Tech. Rep. UTML TR 2009-004, Department of Computer Science, University of Toronto, November 2009. 11

[10] "python wrapper for libfreenect, http://openkinect.org/wiki/Python_Wrapper," Mar. 2011. 12

[11] "Openni manual, http://www.openni.org/documentation," Mar. 2011. 12

[12] S. Rusinkiewicz and Levoy, "a multiresolution point rendering system for large meshes," *Siggraph*, 2000. 12

[13] B. S. N. P. K. Hammoudi, F. Dornaika, "Extracting wire-frame models of street facades from 3d point clouds and the corresponding cadastral map," *Remote Sensing and Spatial Information Sciences*, vol. 38, 2010. 12

[14] "Stl format, http://www.ennex.com/~fabbers/StL.asp," Mar. 2011. 13

[15] "Ply format, http://paulbourke.net/dataformats/ply/," Mar. 2011. 13, 16

[16] "Meshlab, http://www.khronos.org/collada/," Mar. 2011. 14

[17] M. Bader, "Monocular calibration," April 2011. 15

[18] Y. H. Sida Wang, Xi Chen, "kinnectproj, http://code.google.com/p/kinnectproj/," April 2011. 17

[19] "Xbox 360 kinect," April 2011. 19

[20] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 1 ed., 2007. 20

[21] "cloudmesh, http://sourceforge.net/projects/cloudmesh," April 2011. 32

[22] "Order online. vrmesh, http://www.vrmesh.com/store/order.asp," April 2011. 33

[23] "Meshlab, http://www.vrmesh.com/store/order.asp," April 2011. 33

[24] "Meshing point clouds, http://meshlabstuff.blogspot.com/2009/09/meshing-point-clouds.html," April 2011. 33

[25] S. Johnson, *Digital Photography*. O'Reilly, 2006. 35

[26] "kinect node, http://www.ros.org/wiki/kinect_node," 2011. 35