Laser Scan Quality 3D Mesh Using Kinect Capture

by

Ronan O'Mullane, B.Sc.

Dissertation

Presented to the

University of Dublin, Trinity College

in partial fulfilment

of the requirements

for the Degree of

Master of Science in Computer Science (Interactive Entertainment Technology)

University of Dublin, Trinity College

2013

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

____

Ronan O'Mullane

August 30, 2013

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Ronan O'Mullane

August 30, 2013

Acknowledgments

I wish to acknowledge the guidance and assistance I received from Dr. Rozenn Dahyot at all stages of this project at TCD.

Ronan O'Mullane

University of Dublin, Trinity College
2013

Laser Scan Quality 3D Mesh Using Kinect Capture

Ronan O'Mullane
Master of Science in Computer Science (Interactive Entertainment Technology)
University of Dublin, Trinity College, 2013

Supervisor: Dr. Rozenn Dahyot

Abstract:

The aim of this project was to extend a piece of open source software developed by David Ganter M.Sc.,Trinity College Dublin[1]. The software known as Kinect Stream Record tool records a depth video, converts it into a series of sets of points (known as point clouds) and stores it in a raw format so it can be accessed by researchers. The tool also features a Mesh Creator tool that loads the video and stitches the individual frames together. In order to do this, the point clouds need to be transformed according to the estimated position of the camera at the moment each frame was captured so that they fit correctly in the combined mesh in a process known as registration. The main aims of this project were to:

1. Extend the Mesh Creator tool so that it stored the point clouds generated from the individual video frames along with their edge data in a recognised mesh

format known as .ply rather than the entire video in the unrecognised raw file format that it was previously.

2. Explore the use of a more robust algorithm to determine the required transformation of each of the frames so that they fit correctly in the combined mesh; specifically the Gaussian Mixture Model registration method.

Both of these tasks were successfully carried out. The research tool was extended to not only store the frames as ply meshes but also to fill holes present in the depth frames. It was found that the robust Gaussian Mixture Model Registration method was indeed an accurate algorithm but would be too slow to construct meshes on the fly as Kinect Fusion can, using the Iterative Closest Point method.

Contents

Acknowledgments **iv**

Abstract **v**

List of Tables **x**

List of Figures **xi**

Chapter 1 Introduction **1**

1.1 Introduction 1
1.2 Overview 2

Chapter 2 State of The Art **4**

2.1 Current Technology 4
2.1.1 Time of Flight 4
2.1.2 Triangulation 5
2.1.3 Kinect 5
2.2 Common Concepts 6
2.3 Modeling with the Kinect 8
2.4 Registration 11
2.4.1 Iterative Closest Point 11
2.4.2 Gaussian Mixture Model Registration 11
2.5 Tools and Libraries 13
2.5.1 Kinect SDK for Windows 13
2.5.2 Kinect Stream Recording Tool 14
2.5.3 Point Cloud Library 16

2.5.4 MeshLab 16
2.5.5 OpenCV 16
2.5.6 MATLAB 17
2.5.7 VXL 17
2.6 Project Description 18

Chapter 3 Point Cloud File Formats & Kinect Fusion **20**
3.1 Point Cloud File Formats 20
3.1.1 PCD Format 20
3.1.2 PLY Mesh Format 22
3.2 Kinect Fusion & ICP 23

Chapter 4 Recovering Edges for better PLY **26**
4.1 Writing Meshes Using the Point Cloud Library 26
4.1.1 Greedy Projection Triangulation Mesh 26
4.1.2 Organised Fast Mesh 29
4.2 PLY Mesh Writer 31
4.2.1 Point Cloud Data vs Depth to 3D Space 31
4.2.2 Process Description 31

Chapter 5 Registration **34**
5.1 Using Gaussian Registration Code 34
5.1.1 Installing VXL library 34
5.1.2 Finding Correspondence 35
5.1.3 Registration on Point Clouds 35
5.1.4 Results 36
5.1.5 Registration Completion Time 43

Chapter 6 Conclusion **45**
6.1 Summary 45
6.2 Discussion of Results & Future Work 46
.1 USB Flash Drive Contents 48
.1.1 code 48
.1.2 data 48

Appendices **48**

List of Tables

5.1 A table displaying the length of time taken in seconds in order to run the registration algorithms with point clouds of different sizes. 44

List of Figures

2.1 Kinect Fusion taking the depth image from the Kinect camera with lots of missing data and within a few seconds producing a smooth 3D reconstruction of a static scene [2]. 10
2.2 The user interface of the Kinect Stream Record Tool. Depth representation of the current frame is displayed on the left. RGB representation of the same frame is displayed on the right [1]. 15
2.3 An Overview of the Project. Work carried out by David Ganter is highlighted in the blue textboxes. Work to be carried out by Ronan O'Mullane is highlighted in the green textboxes 19

3.1 The KinectFusionBasics-D2D example capturing a scene of a shoe on a table with chairs. Note that the program is returning the following error at the bottom of the window 'Kinect Fusion camera tracking failed. Align the camera to the last tracked position' 25

4.1 A PLY mesh of a frame's point cloud with a resolution of 320×240 generated using the greedy projection algorithm. This algorithm results in holes appearing in the mesh. 28
4.2 A PLY mesh generated from the point cloud of the same frame as 4.1 but recorded with a resolution of 80×60. This mesh was also generated using the greedy projection algorithm. The holes are also evident at this resolution. 28

4.3 A PLY mesh generated from a frame's point cloud using the Organised Fast Mesh algorithm. This mesh has a resolution of 320×240. The 'cone' effect is evident here where points bordering holes are being connected to the origin. 30
4.4 A PLY mesh generated from a frame's point cloud using the Organised Fast Mesh algorithm at the 80×60 resolution. Again, the 'cone' effect is evident at the lower resolution. 30
4.5 A PLY mesh of a point cloud generated from a depth frame with a resolution of 320×240 without the simple hole filling implementation. Areas with no depth value appear closest to the screen 33
4.6 A PLY mesh of a point cloud generated from the same frame as Fig. 4.5 with a resolution of 320×240 with the hole filling implementation. Problem areas are no longer visible 33

5.1 Gaussian registration and a rigid transformation performed on two 80 × 60 point clouds generated from frames 850(scene) and 854(model) as visualised in MeshLab 37
5.2 The MATLAB output of the rigid transformation seen in Fig. 5.1 from a slightly aerial perspective. Frame 850 is coloured in blue frame 854 is coloured in red. Frames before the rigid transformation are seen on the left; frames after the transformation are seen on the right. 37
5.3 Gaussian registration and a rigid transformation performed on two 320 × 240 point clouds generated from frames 70(scene) and 77(model) as visualised in MeshLab. The transformed frame 77 model appears darker in the merged point cloud. 38
5.4 The MATLAB output of the rigid transformation seen in Fig. 5.3 from a slightly aerial perspective. Frame 70 is coloured in blue frame 77 is coloured in red. Frames before the rigid transformation are seen on the left; frames after the transformation are seen on the right. 39

5.5 Gaussian registration and a non-rigid transformation using thin plate splines performed on two 80 $\times$ 60 point clouds generated from frames 850(scene) and 854(model) as visualised in MeshLab. The model point cloud is distorted slightly in order to align with the scene as expected with this type of transformation 40
5.6 The MATLAB output of the non-rigid transformation seen in Fig. 5.5 from a slightly aerial perspective. Frame 854 is coloured in blue frame 850 is coloured in red. Frames before the rigid transformation are seen on the left; frames after the transformation are seen on the right. . . . 41
5.7 Gaussian registration and a non-rigid transformation using thin plate splines performed on two 80 $\times$ 60 point clouds generated from frames 4(scene) and 80(model) as visualised in MeshLab. The registration algorithm failed to align the point clouds correctly 42
5.8 The MATLAB output of the non-rigid transformation seen in Fig. 5.7 from a head on perspective. Frame 4 is coloured in blue frame 80 is coloured in red. Frames before the rigid transformation are seen on the left; frames after the transformation are seen on the right. 43

Chapter 1

Introduction

1.1 Introduction

Three Dimensional geometric models of real world objects have become essential today in the entertainment industries of gaming and film through their constant demand for realistic special effects. Other application areas have emerged in more recent times such as design and virtual prototyping, quality assurance and medical software. This, coupled with the recent developments in the area of accessible 3D printing has made it necessary to broaden the availability of 3D scanners. Up until recently, the process of creating 3D models was time consuming and expensive; often requiring skilled artists and modelling software inaccessible to most unskilled users. Over the years progress has been made in automating the process of generating 3D models through the use of shape scanning technology. With the 2010 release of Microsoft′s Kinect device, such sensors have been made accessible to all.

Though the Kinect is quite accessible and Microsoft are very supportive of developers through their consistant updates for the Kinect SDK [3], the system remains quite closed. Much of the internal functionality that is known about the Kinect has been discovered through reverse engineering [4] and is not accessible to researchers. Herrera et al.[5] discuss in a recent paper that when developing a new calibration algorithm that relies on both depth and colour information they choose the Kinect for implementation. They cite the Kinects affordability as well as its popularity as deciding factors in that decision. They determined that a calibration method that functions on

the Kinect would be widely applicable to other devices. They do however reflect on how the device is not open and how they do not have access to some original intensity images used by Microsoft for the calibration. They also discuss how the Kinects depth camera undergoes no calibration out of the box and it falls into the hands of researchers to tackle such problems.

It is for these reasons that open source projects pertaining to the Kinect hold such merit. The Kinect could be a powerful device for research if more access to the internal functionality was made available to researchers. The project described throughout this dissertation makes steps towards both uncovering the functionality of the Kinect as well as exploring alternatives to some of the algorithms already in place.

1.2 Overview

Chapter 2 - Sate of the Art will discuss some of the current 3D scanning technology before giving a brief introduction to the major concepts of geometry as well as some of the terminology relating to this area. This chapter will go on to discuss some of the prominent research relating to 3D scanning with the Kinect before describing the relevant tools and libraries in the area of computer vision and modeling. Finally the chapter will give a brief overview of the work that was to be carried out and highlight the aims of the project.

Chapter 3 - Point Cloud File Formats & Kinect Fusion will present two file formats and discuss how well they would be suited to the goals of the project as described in chapter 2. The chapter will then go on to discuss the example of Kinect Fusion released in the Kinect for Windows SDK.

Chapter 4 - Recovering Edges for better PLY will describe how an application for Recording and managing Kinect video data was extended in order to store individual depth frames as point clouds. It will the go on to discuss some methods to create meshes from point cloud data before giving a detailed description of how a ply mesh writer was written to store that data along with the edges between the vertices.

Chapter 5 - Registration will describe how point clouds generated using the Kinect and the aforementioned application were merged using an alternative registration algorithm to the one used by Microsoft for Kinect Fusion. It will also detail how to install libraries that the implemented algorithm is dependant on before presenting the results

of the registration process.

Finally Chapter 6 will present the conclusion and describe future work that could be pursued.

In the Appendices, information can be found on how to run the code on the accompanying usb drive as well as a description of what data can be found there.

Chapter 2

State of The Art

2.1 Current Technology

There are a wide variety of technologies available for acquiring 3D shape data of an object. This section will describe a few of these technologies while discussing their advantages and disadvantages.

2.1.1 Time of Flight

The time-of-flight 3D laser scanner is an active scanner that uses laser light to probe the subject. The camera uses a laser range finder to determine the distance of a surface by timing the round-trip time of a pulse of light. A laser is used to emit a pulse of light and measures the amount of time before the reflected light is detected by the device. Since the speed of light c is known, the round-trip time determines the travel distance of the light, which is twice the distance between the scanner and the surface. If t is the round-trip time, then distance is equal to $c \cdot t/2$[6].

The laser rangefinder only detects the distance of one point in its direction of view. In order to scan the entire field of view the range finder's direction is changed and each of the different points are measured one at a time. The view direction of the laser rangefinder can be changed either by rotating the range finder itself, or by using a system of rotating mirrors. The latter method is commonly used because mirrors are much lighter and can thus be rotated much faster and with greater accuracy.

Time-of-flight range finders are capable of operating over very long distances, in the

order of kilometers. Therefore, these scanners are suitable for scanning large structures like buildings or geographic features. The disadvantage of time-of-flight range finders is their accuracy. Due to the high speed of light, timing the round-trip time is difficult. Time-of-flight range finders are expensive and not ideal for indoor use.

2.1.2 Triangulation

Triangulation based 3D laser scanners are also active scanners that use laser light to probe the environment. Rather than measuring the round trip time of the emitted light, these scanners use a camera to determine the location of the laser dot. Depending on the distance of the subject at the target point, the laser dot appears on a different location in the cameras field of view. This technique is called triangulation because the laser dot, the camera and the laser emitter form a triangle. The length of one side of the triangle, the distance between the camera and the laser emitter is known. The angle of the laser emitter corner is also known. The angle of the camera corner can be determined by looking at the location of the laser dot in the camera's field of view. These three pieces of information fully determine the shape and size of the triangle and gives the location of the laser dot corner of the triangle. In most cases a laser stripe, instead of a single laser dot, is swept across the object to speed up the acquisition process[7].

Unlike time-of-flight range finders, triangulation scanners have a limited range of some meters but have a relatively high accuracy. The accuracy of triangulation range finders is in the order of tens of micrometers.

2.1.3 Kinect

Kinect is a motion sensing input device developed by Microsoft initially for use with the Xbox 360 video game console and later Windows PCs. The device features a depth sensor consisting of an infrared laser projector combined with a monochrome CMOS sensor, which captures video data in 3D under any ambient light conditions. The device is capable of recording depth data by using a structured-light technique. Structured-light 3D scanners project a pattern of light on the subject and look at the deformation of the pattern on the subject. In the case of the Kinect this is done using the devices infrared projector and sensor. The camera, offset slightly from the pattern

projector, looks at the shape of the pattern and calculates the distance of every point in the field of view by comparing the detected pattern against the projected pattern. The advantage of structured-light 3D scanners is speed and precision. Instead of scanning one point at a time, structured light scanners scan multiple points or the entire field of view at once. Scanning an entire field of view in a fraction of a second generates profiles that are exponentially more precise than laser triangulation. This reduces or eliminates the problem of distortion from motion. The Kinect offers a portable cheaper alternative to other 3D scanning devices, however the data collected is often noisy.

2.2 Common Concepts

Image: An image I of size w $\times$ h is a finite set of values I (x,y) $\in \mathbb{R}^d$ where x $\in$ {0,...,w - 1}, y $\in$ {0,...,h - 1}, d = 3 for a colour image, d = 1 for an intensity image. For a non-integer position (u,v) in the range of image, I (u,v) is computed as an interpolation of the value of pixels around (u,v). Therefore,I can be considered as a function in a continuous domain [0, w - 1]$\times$[0,h - 1].

Camera: Hartley and Zisserman[8] describe the pinhole camera; a popular model of the camera for use in multi-view stereo methods. This model is such that the camera's center, the 3D point of the model surface, and the corresponding pixel are collinear. Because the camera is placed on a flat surface, it assures that a straight line in the world space remains a straight line in the image. A pinhole camera is presented mathematically as a 3$\times$4 matrix and the projection is a formula of projective geometry. With two images I_1, I_2 taken from pinhole cameras, for a pixel p_1 in I_1, it is not necessary to search all pixels of I_2 to find its match,but only along a segment in I_2 known as the epipolar line.

Depth map/Image range: Each pixel p of an image, corresponding to a point $P \in \mathbb{R}^3$ the scene surface, the depth of pixel p is the distance between the camera center C to the perpendicular projection of P in the principal direction line of the camera. The depth map of an image is the set of depth values at all its pixels.

Point cloud/Point set: A set of points in $\mathbb{R}^3$ that is a sample of the object sur-

face. Depending on algorithm used to create this point cloud, it may contain many points that are a significant distance from the actual surface (called outliers). Each point should associate with images from which it is triangulated. The Point Cloud was first introduced by Rusinkiewicz[9].

Vertex: A vertex is a data structure that describes a point in either 2D or 3D space. Models are composed of arrays of flat surfaces (typically triangles or quads) and vertices define the location and other attributes of the corners of the surfaces.

Normal: A line or vector that is perpendicular to another object is said to be a normal to that object. In the two dimensional case, the normal line to a curve at a given point is the line perpendicular to the tangent line to that curve at that point. In the three dimensional case the vector perpendicular to the plane that acts as tangent to a surface is said to be the normal vector of the surface.

Patch: A patch is 3D rectangle shape segment of a surface. Normally it is a tangent patch to the surface. It comprises of a 3D point along with its normal vector.

Triangular Mesh: A Triangular Mesh is a polyhedron whose faces are triangles. Triangular meshes are commonly used in computer graphics and the games industry, where texture projection is easily computed with commodity hardware. Triangular meshes are an economical means of representing an object's surface. The main drawback of using a triangular mesh is the difficulty in handling topological change during surface evolution.

Voxel: A voxel is the smallest entity of a 3D volume. This is analogous to a pixel, which represents 2D image data in a bitmap. As with pixels in a bitmap, voxels themselves do not typically have their position explicitly encoded along with their values. Instead, the position of a voxel is inferred based upon its position relative to other voxels (i.e., its position in the data structure that makes up a single volumetric image). In contrast to pixels and voxels, points and polygons are often explicitly represented by the coordinates of their vertices. A direct consequence of this difference is that polygons are able to efficiently represent simple 3D structures with lots of empty or

homogeneously filled space, while voxels are good at representing regularly sampled spaces that are non-homogeneously filled.

Level Set: The objects surface is represented implicitly as the zero level set for a scalar function f in $\mathbb{R}^3$. For implementation, the function is typically dened over a volumetric grid, or over a small band near the surface. The main advantage of level methods is any deformation of the surface can be done by modifying the underlying function. Thus, it also handles natural topological change. However, it has many shortcomings including; large memory consumption for the grid that leads to expensive computation along with difficulty in tracking correspondence during the surface evolution.

Silhouette: The silhouette is the projection of the object surface on an image. Such projection shape can be easily estimated by segmentation, especially if the background color is homogeneous and quite different to surface color e.g. a white statue in a black background.

Visual Hull: If the silhouette of the object on an image is known, then the silhouette along with the camera viewing parameters defines a back-projected generalized cone that contains the actual object. This cone is called a silhouette cone. The intersection of two or more of these cones is called a visual hull, which is a bounding geometry of the actual 3D object. This concept was first introduced by Laurentini[10].

Registration: The problem of registration can be defined as follows: Given a fixed 'scene' point set S and a moving 'model' point set M where both models are subsets of a finite-dimensional real vector space R^d, estimate the mapping from $\mathbb{R}^d$ to $\mathbb{R}^d$ that best aligns the transformed model M to the scene S. The mapping may represent a rigid or non-rigid transformation[11].

2.3 Modeling with the Kinect

Microsoft's Kinect device is capable of capturing depth data from a scene. The quality of the depth sensing, given the low-cost and real-time nature of the device has made

the sensor instantly popular with researchers and enthusiasts alike. The Kinect camera uses a structured light technique to generate real-time depth maps containing discrete range measurements of the physical scene. This data can be projected as a set of discrete 3D points (or point cloud).

Even though the Kinect depth data is compelling, particularly compared to other commercially available depth cameras, it is still inherently noisy as described by Izadi et al.[12]. Depth measurements can fluctuate leaving areas in the depth map where no readings are obtained whatsoever. In order to generate high level 3D meshes, surface geometry needs to be inferred from this noisy point based data.

Cui et al.[13] discovered a similar problem when attempting to generate three dimensional models using a Time of Flight camera. They developed and successfully implemented an algorithm that compensates for noise. This algorithm involves a combination of a 3D super resolution method with a probabilistic scan alignment approach that explicitly takes into account the sensors noise characteristics. The results of their research proved that 3D models of a reasonable quality can in fact be generated using economical equipment regardless of noise.

A similar approach was adopted by Newcombe et al.[14] for The Kinect Fusion project. This paper presents a solution whereby a complete 3D model can be generated by capturing different viewpoints of a scene and fusing their interpolated depth data into a single representation. A user holding a Kinect camera can move within any indoor space and automatically reconstruct a 3D model of the scene within seconds. Using only depth data, the system continuously tracks the 6 degrees-of-freedom (6DOF) pose of the sensor using all of the live data available from the Kinect device rather than an abstracted feature subset, and integrates depth measurements into a global dense volumetric model. Tracking is performed at 30Hz frame-rate and is always relative to the fully up-to-date fused dense model. By using only depth data the system can work in complete darkness mitigating any issues concerning low light conditions, problematic for passive camera and RGB-D based systems. Fig. 2.1 displays the reconstruction of a static scene using Kinect Fusion.

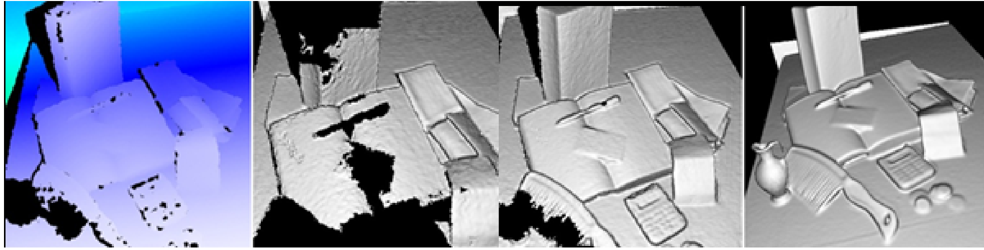

Figure 2.1: Kinect Fusion taking the depth image from the Kinect camera with lots of missing data and within a few seconds producing a smooth 3D reconstruction of a static scene [2].

The implementation of Kinect Fusion can be broken down into the following steps:

Depth Map Conversion: The live depth map is converted from image coordinates into vertices and normals in the coordinate space of the camera.

Camera Tracking: A rigid 6DOF transform is computed to closely align the current oriented points with the previous frame, using a GPU implementation of the Iterative Closest Point algorithm described in the next section.

Volumetric Integration: A volumetric surface representation is generated rather than fusing point clouds or creating a mesh. Oriented points are converted into global points given the pose of the camera and a single 3D voxel grid is updated. Given the global pose of the camera, oriented. Each voxel stores a running average of its distance to the assumed position of a physical surface.

Raycasting: The volume is raycast to extract views of the implicit surface, for rendering to the user. This raycasted view of the volume also equates to a synthetic depth map, which can be used as a less noisy more globally consistent reference frame for the next iteration of Iterative Closest Point. Live depth maps are aligned with the less noisy raycasted view of the model rather than merely the frame to frame depth maps.

2.4 Registration

2.4.1 Iterative Closest Point

The iterative closest point (ICP) algorithm first introduced by Besl and McKay[15] is one of the most well-known algorithms for point set registration. The idea of the ICP algorithm was motivated by the fact that there exist closed-form solutions for estimating 3D rigid body transformations, given a correspondence of point pairs [16]. Traditional ICP works as follows:

for each point m_i in the model set M, find its closest point, s_i, in the scene set S. The rigid transformation T that best aligns the $\{m_i,\ s_i\}$ pairs in a least squares sense is then calculated using the closed form solution. Then, all the points in M are transformed by T.

This establish-correspondence-then-register cycle is iterated until the specified stopping criterion is satisfied. The traditional ICP algorithm is intuitive and simple but has practical limitations due to its assumption that every closest point pair should correspond to each other. This assumption can easily fail when the two point sets are not coarsely aligned or the model set is not a proper subset of the scene due to possible occlusions in the scene.

2.4.2 Gaussian Mixture Model Registration

Jian and Vemuri[11] proposed a method of using Gaussian mixture models to represent the point set data with the intention of giving less credence to outliers and in some sense reflecting the uncertainty of feature extraction. The point sets are interpreted as a statistical sample drawn from a continuous probability distribution of random point locations rather than the discrete points themselves. By doing so, traditionally hard discrete optimization problems usually encountered in the point matching literature can be converted to more tractable continuous optimization problems. The implementation of this algorithm can be described as follows:

Given the model set M, the scene set S, and a parametrized transformation model T, output the optimal transformation parameter θ of model T that best aligns M and S.

1. begin

2. Estimate an initial scale σ from input point sets;
3. Specify an initial parameter θ, e.g. from the identity transform;
4. repeat
5. Set up the objective function $f(\theta)$ as the L2 distance between the Gaussian mixtures constructed from the transformed model $T(M, \theta)$ and the scene S with a scale σ. A regularization term can be added depending on the transformation model;
6. Optimize the objective function f using a numerical optimization engine (e.g. quasi-Newton algorithm when ∇f is available) with θ as the initial parameter;
7. Update the parameter $\theta \leftarrow \text{argmin}_T f$;
8. Decrease the scale σ accordingly as an annealing step;
9. until some stopping criterion is satisfied
10. end

This algorithm was found to perform well with 50% more outliers and was comparable in accuracy to an ICP algorithm using a Levenberg-Marquardt algorithm rather than the standard closed form solution. It was also found to perform slightly better at higher angles of disparity between depth images. Arellano and Dahyot[17] took this algorithm further and presented a Mean Shift Algorithm which built upon the aforementioned method. Using the Euclidean distance between the mixture of Gaussians it was shown that their algorithm was more robust due to the annealing framework implemented and the use of variable bandwidth for modelling the density functions.

Shape from X

There are many other ways to estimate the shape of objects with information like shading, texture and focus. If it is possible to control and adjust the light source, the resulting shadows on an objects surface can provide orientation information such as normal vectors. When the light source is a texture's pattern, the deformation of the pattern wrapped onto the surface can also assist in estimating the surface orientation.

The camera focus is also an indication of object depth.The amount of blur increases as the object moves away from the cameras focal distance. By adjusting the camera focus and measuring the blur of images, the depth of the object can be recovered.

Structure from Motion

The extrinsic and intrinsic parameters of the camera along with the shape of an object can be estimated successfully using structure from motion techniques. A popular and powerful method to solve Structure from Motion is Bundle adjustment, which is described in detail in many papers and textbooks such as Triggs et al.[18],Hartley and Zisserman,[8] and Snavely et al.[19]. First,the key points of the input images are extracted then matched together. Second,from the obtained matching, the initial cameras parameters are estimated and matching pixels are triangulated to obtain 3D points. These parameters are iteratively adjusted to optimize a sum of the squared distance between matching pixels and the computed projection of 3D points. Snavely et al.[19] implemented a method where input cameras were added incrementally during the execution. This incremental approach is quite time consuming and is improved by a direct initialization in Crandall et al.[20].

2.5 Tools and Libraries

2.5.1 Kinect SDK for Windows

The Kinect for Windows SDK and toolkit contain drivers, tools, APIs, device interfaces, and code samples to simplify development of applications for commercial deployment or academic research [3]. The SDK is frequently updated and is currently on version 1.7 as of 2013.

Kinect Studio

The Kinect Studio application which comes with the Kinect SDK is a useful tool that enables the recording and playback of depth and color streams captured from a Kinect [2]. The tool is intended to be used to read and write data streams to help debug functionality, create repeatable scenarios for testing, and analyze performance.

While this tool does in fact deliver on the concept of recording kinect data, it requires an application that uses the Kinect be operational before it will begin recording. It also requires a Kinect to be connected before it can playback the recorded files. The recorded videos are stored are .xed files which is a proprietary format controlled by Microsoft.

Kinect Fusion

An implementation of the Kinect Fusion algorithm also comes as part of the Kinect Developer Toolkit in the Kinect SDK. Kinect Fusion performs 3D object scanning and model creation using a Kinect for Windows sensor [2]. It allows users to paint a scene with the Kinect camera and simultaneously see, and interact with, a detailed 3D model of the scene. Kinect Fusion can be run at interactive rates on supported GPUs, and can run at non-interactive rates on a variety of hardware. The minimum hardware requirement for GPU based reconstruction is a DirectX 11 compatible graphics card. Four code samples come with the SDK to assist developers. The KinectFusionBasics-D2D and KinectFusionExplorer-D2D are in native C++ and the KinectFusionBasics-WPF and KinectFusionExplorer-WPF are in C#.

2.5.2 Kinect Stream Recording Tool

An open source tool for recording and managing depth data captured from Microsoft's Kinect device has been developed by David Ganter, Trinity College Dublin in 2013[1]. Much like Kinect studio it can capture depth and RGB data at multiple resolutions but stores the raw data as a custom open source file type, giving researchers access to that raw data. The application also has the advantage of being operable without the need for a Kinect device to be connected and it also runs independently of other applications. Fig. 2.2 displays the user interface of the Kinect Stream Record tool with the depth representation of the current video frame presented on the left side of the window and the RGB frame on the right.

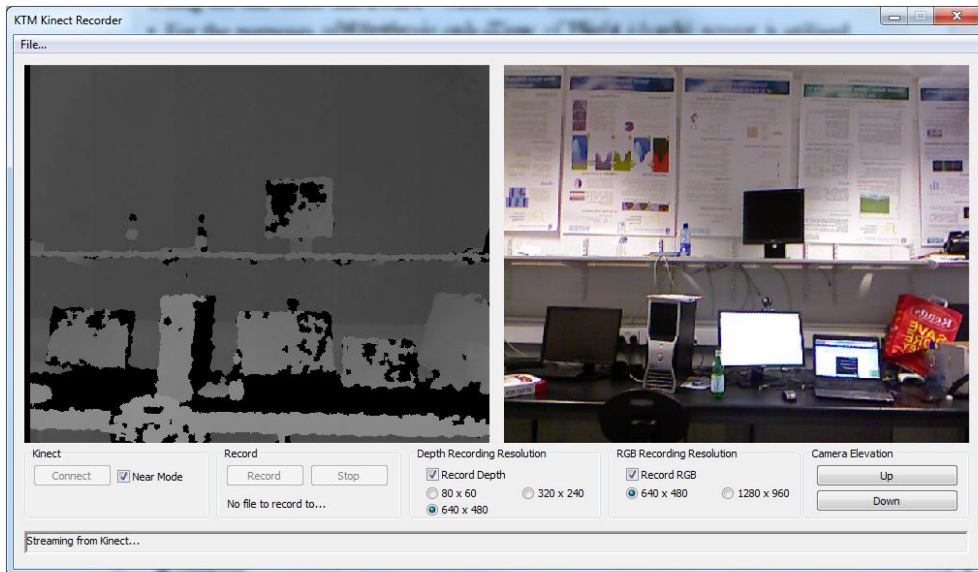

Figure 2.2: The user interface of the Kinect Stream Record Tool. Depth representation of the current frame is displayed on the left. RGB representation of the same frame is displayed on the right [1].

As well as merely storing the depth data the Mesh Creator application has the ability to load its own stored data files for repeated testing and displays a point cloud extrapolated from this data as it plays back the recorded frames. Much like Newcombe et al.[14], the system employs the Iterative Closest Point algorithm to determine the global camera position at each frame. However, this system merges the point clouds to generate one unified cloud consisting of all of the frames from the recording rather than using a volumetric grid of voxels. The Recording tool also interfaces with MATLAB allowing some registration methods to be run on the point cloud data and enhancing the research development environment. It is noted that there is a degree of radial distortion if the raw data is used to generate the point cloud directly. This occurs as a result of the Kinect acting as a pin hole camera and object to the periphery of the captured scene having a larger focal length than object directly in front of the camera. In order to overcome this issue a function known as Depth to 3D Space in the Kinect SDK was used to correct the distortion which takes in a raw depth value and returns a 4 element vector containing a point on the x,y and z axes as well as a depth value for that point and returns a new x,y and z coordinate without radial distortion.

2.5.3 Point Cloud Library

The Point Cloud Library (PCL) is a standalone open-source framework including numerous state of the art algorithms for n-dimensional point clouds and 3D geometry processing [21]. The library contains algorithms for filtering, feature estimation, surface reconstruction, registration, model fitting, and segmentation. It also provides structures for representing point clouds,k-d trees and quad meshes as well as libraries for file I/O and search algorithms. PCL is developed by a large consortium of researchers and engineers around the world. It is written in C++. The library is frequently updated and is on version of 1.7 as of 2013.

2.5.4 MeshLab

MeshLab is an advanced 3D mesh processing software system which is well known in the more technical fields of 3D development and data handling [22]. As free software it is used both as a complete package, and also as libraries powering other software. MeshLab is developed and supported by the ISTI - CNR research center. MeshLab was initially created as a course assignment at the University of Pisa in late 2005. It is an open-source general-purpose system aimed at the processing of typical unstructured 3D models that arise in the 3D scanning pipeline. MeshLab is oriented to the management and processing of unstructured large meshes and provides a set of tools for editing, cleaning, healing, inspecting, rendering and converting these kinds of meshes. MeshLab supports the input and output of many file formats including: PLY, STL, OFF, OBJ, 3DS, VRML 2.0, U3D, X3D and COLLADA as well as raw files of ascii values.

2.5.5 OpenCV

The OpenCv Library is an open source cross platform library which provides thousands of functions that focus on real time computer vision[23]. It was originally developed by Intel and is now free for both commercial and research use. It is currently supported by Willow Garage and as of this time it is currently on version 2.4.6. It has C, C++, and Python interfaces supported under Windows , Linux, Android and Mac. It provides many functions that focus on:

- Capturing video frames from a video source

- Loading and storing images
- Accessing image data through the use of matrices
- Performing transformations and manipulating images
- Outputting image and video data
- Drawing
- basic coordinate geometry

OpenCV is well regarded as one of the leading computer vision APIs and provides similar functionality to other libraries in the field such as Aforge and MATLAB.

2.5.6 MATLAB

MATLAB is a high-level language and interactive environment developed by MathWorks [24]. It is most commonly used for numerical computation and visualization while programming. MATLAB can assist in the analysis of data, development of algorithms and creation of models and applications. The language, tools, and built-in math functions are capable of providing solutions faster than with spreadsheets or traditional programming languages, such as C/C++ or Java. MATLAB is used for a range of applications, including signal processing and communications, image and video processing, control systems, test and measurement, computational finance, and computational biology. It is well regarded as being the language of technical computing and is at the forefront of scientific and academic research.

2.5.7 VXL

VXL is a large collection of C++ libraries designed for computer vision research and implementation[25]. It was created from TargetJr and the IUE with the aim of making a light, fast and consistent system. VXL is written in ANSI/ISO C++ and is designed to be portable over many platforms. The core libraries in VXL are:
vnl (vision numerics library): This library provides numerical containers and algorithms such as matrices, vectors, decompositions and optimisers.

vil (vision imaging library): This library focuses on the loading, saving and manipulation of images in many common file formats, including very large images.
vgl (vision geometry library): This library focuses on geometry for points, curves and other elementary objects in 1, 2 or 3 dimensions.

The vsl (vision streaming I/O library), vbl (vision basic templates library) and vul (vision utilities library) provide miscellaneous platform-independent functionality. As well as the core libraries, there are libraries covering numerical algorithms, image processing, co-ordinate systems, camera geometry, stereo, video manipulation, structure recovery from motion, probability modelling, GUI design, classification, robust estimation, feature tracking, topology, structure manipulation and 3D imaging. As of 2013 it's current release is version 1.17.

2.6 Project Description

While the research tool developed by David Ganter[1] does successfully record Kinect depth data, the raw file format used to represent this data is not ideal in some respects. The file format consists of taking the depth information of each pixel as an unsigned short and casting this value to two unsigned char values. These two char values are then output to a file. In order to recover the depth information when loading in a file, every pair of unsigned char values are cast back to their original unsigned short values. This custom file format used, while it is not a proprietary format is still only readable by the application itself or alternatively it would require someone to write a custom application to read the format. The research tool also only dealt with point cloud vertices ignoring the edge data between those vertices. The edge data is important for visualisation of the individual frames and may be relevant during the registration process, depending on the algorithm to be implemented.

The first goal of this project was to extend the MeshCreator.exe program so that the individual point clouds along with their edge data were written to the hard disk in a known non-proprietary format. Two file formats were researched and eventually one representation was chosen as an alternative to the custom format. This will be discussed in the next chapter. The second goal of this project was to explore the implementation of an alternative registration algorithm, specifically the gaussian mixture model method

developed by Jian and Vemuri[11] described in the previous chapter. This will be detailed in the next chapter. Fig. 2.3 shows the work carried out previously in blue and the work to be carried out in this project in green.

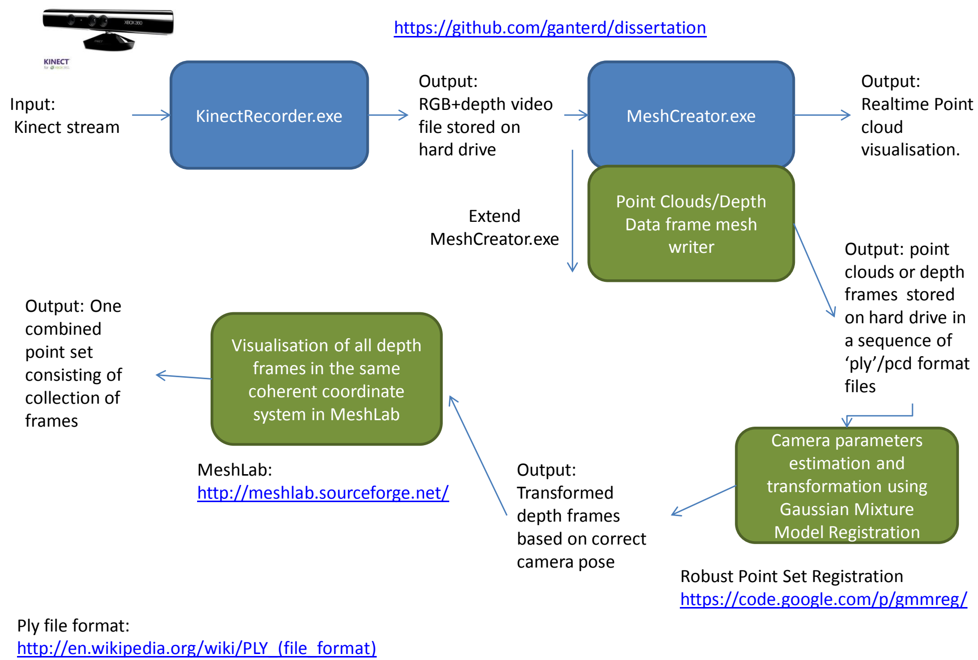

Figure 2.3: An Overview of the Project. Work carried out by David Ganter is highlighted in the blue textboxes. Work to be carried out by Ronan O'Mullane is highlighted in the green textboxes

Chapter 3

Point Cloud File Formats & Kinect Fusion

3.1 Point Cloud File Formats

3.1.1 PCD Format

The PCD or Point Cloud Data format is a file format supported by the Point Cloud Library. It is used to store the data of 3 dimensional point cloud data structures in either ASCII or binary representation. The format consists of a header followed by the point cloud's vertex list denoted in the specified representation. The header consists of ten entries:

- VERSION: specifies the PCD file version
- FIELDS: specifies the name of each dimension/field that a point can have. The format can represent

    x, y, z positional data

    RGB data

    normal data

    moment invariants.

- SIZE: specifies the size of each dimension in bytes.

- TYPE: specifies the type of each dimension as a char. The currently supported types are

  I - represents signed types int8 (char), int16 (short), and int32 (int)

  U - represents unsigned types uint8 (unsigned char), uint16 (unsigned short), uint32 (unsigned int)

  F - represents float types

- COUNT: specifies the number of elements each dimension has.
- WIDTH: specifies the width of the point cloud dataset in the number of points. In unorganised point clouds this represents the entire number of points in the cloud, whereas in organised point clouds this represents the number of points in a row.
- HEIGHT: specifies the height of the point cloud dataset in the number of points. If the cloud is organised then it represents the number of rows, otherwise it is set to 1 in which indicates that the point cloud is unorganised.
- VIEWPOINT: specifies an acquisition viewpoint for the points in the dataset.
- POINTS: specifies the total number of points in the cloud.
- DATA: specifies the data type that the point cloud data is stored in. As of version 0.7, two data types are supported: ascii and binary.

Initially the PCD format seemed to be the most viable option for storing the point cloud data produced by the Kinect Research tool on disk. It allowed for the point cloud produced at each frame to be stored as a separate PCD file in an easy to read ASCII format. This format also seemed ideal as the Point Cloud Library used to develop the Kinect Stream Recording tool already has functions that are capable of writing point clouds as PCD files as well as reading those files in as point cloud objects. Initially the research tool was altered to write each frame's point cloud as a PCD file and work had begun on a program capable of reading those files in using the savePCDFileASCII and loadPCDFile functions of the Point Cloud Library.

However, some drawbacks were discovered in that the PCD file format was not broadly recognised by third party applications such as MeshLab. While MeshLab

provides a means to extend its supported file formats through the use of plugins, no PCD plugin exists at this time and writing one would be outside the scope of this project. The only means of visualising these PCD files was through the use of the Cloud Viewer application that comes with the Point Cloud Library. This application is quite restrictive and does not offer the host of features other model viewing applications such as MeshLab provide. Another major disadvantage is that the PCD file format is incapable of storing edge data. Depending on the registration algorithm that was to be used later on when merging the depth frames, the edge data could be key in their implementation. For these reasons an alternative file format was to be employed.

3.1.2 PLY Mesh Format

The Polygon File Format (PLY) or Stanford Triangle Format is very similar to the PCD format in its simplicity but is in fact a regonised file format supported by MeshLab. It is quite similar to the PCL library's PCD format in that it consists of a header followed by data represented in either binary or ASCII but rather than just storing a point cloud, it is capable of storing a complete mesh consisting of both vertices and edges. In order to achieve this an additional face list is stored along with the vertex list which describes which points are connected. Each entry in the face list first consists of the number of points that are to be connected in that face (e.g 2 for an edge, 3 for a triangle, 4 for a quad) followed by the indexes of the faces to be connected as they appear in the vertex list. The header of a ply file has the following entries:

- format: specifies whether the format is binary big-endian,binary little-endian or ascii followed by the version number.
- element vertex: specifies the number of vertices in the file as an integer.
- property float x: specifies that each vertex has a property x that is a float.
- property float y: specifies that each vertex has a property y that is a float.
- property float z: specifies that each vertex has a property z that is a float.
- element face: specifies the number of vertices in the file as an integer.
- property list uchar int vertex_index: specifies the vertex indices are a list of ints.

- end_header: delimits the end of the header.

The PLY format was a better choice in that it did not have the drawbacks that accompanied the use of the PCD format mentioned above. The point cloud data of each frame along with the edges could be converted to a mesh object and then written as individual PLY files.

3.2 Kinect Fusion & ICP

The traditional ICP algorithm used in Kinect Fusion to determine the camera position is intuitive and simple but has practical limitations due to its assumption that every closest point pair should correspond to each other [11]. This assumption can easily fail when the two point sets are not coarsely aligned or the model set is not a proper subset of the scene due to possible occlusions in the scene. This problem is evident in the Kinect Fusion example KinectFusionBasics-D2D which comes as part of the Kinect Developer Toolkit in the Kinect SDK (see Fig. 3.1). The example code was run and it was noted that when moving the Kinect around a scene the program would often return an error stating that the tracking had failed and requesting that the camera be realigned to the last tracked position.

Because it is quite difficult to determine where exactly the last tracked position was this often lead to the system failing and the construction being reset. This reaffirms the findings of Jian and Vemuri[11] and presents a case that their registration algorithm using Gaussian Mixture models may be suited to such an application. This example was designed to run on hardware that features a DirectX 11 compatible GPU ideally with 2GB of onboard memory. Because such hardware was unavailable, minor changes to the example code were made. In order to run the code in non realtime CPU mode the parameter

m_ processorType = NUI_FUSION_RECONSTRUCTION_PROCESSOR_TYPE_AMP
must be changed to
m_processorType = NUI_FUSION_RECONSTRUCTION_PROCESSOR_TYPE_CPU

Because the frame rate is slower in CPU mode another parameters has to be al-

tered. The cResetOnTimeStampSkippedMilliseconds stores the length of time before the program determines that the frame timestamp has skipped a large number and resets the generated 3D construction. This is is to enable playback of a .xed file through Kinect Studio and reset the reconstruction if the video loops. However, due to the slow frame rate when running on the CPU if this time is shorter than the length of time between frames then the reconstruction resets at every frame. Therefore the cResetOnTimeStampSkippedMilliseconds parameter needs to be increased until it is longer than the time between frames.

This example was tested on a machine with a 2.53Ghz 64bit Intel i5 processor with 4gb of RAM in CPU mode and produced output between 0.09 and 0.25 frames per second. Because the machine only featured an Intel integrated graphics card the realtime GPU mode could not be tested. The program was also run on machines in the IET Lab and produced similar results.

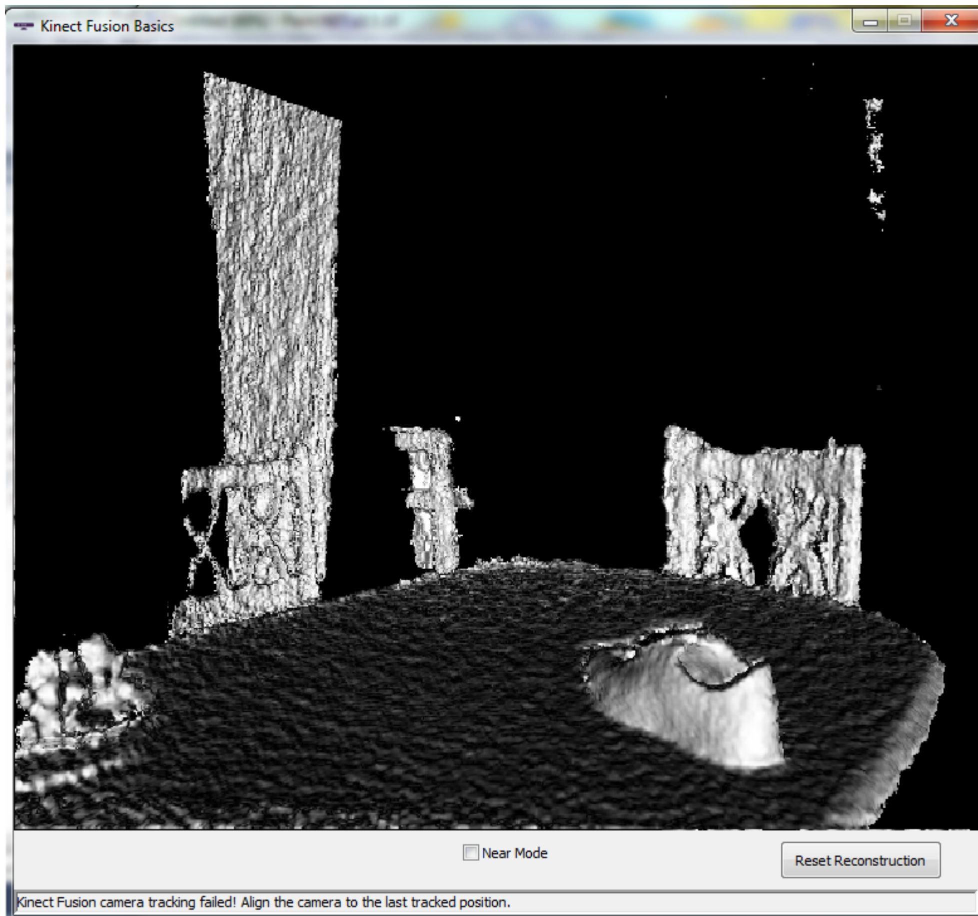

Figure 3.1: The KinectFusionBasics-D2D example capturing a scene of a shoe on a table with chairs. Note that the program is returning the following error at the bottom of the window 'Kinect Fusion camera tracking failed. Align the camera to the last tracked position'

Chapter 4

Recovering Edges for better PLY

4.1 Writing Meshes Using the Point Cloud Library

The point cloud library offers some functions for outputting depth data as a PLY mesh, specifically the savePLYFile function which takes in a string to denote the file name, a mesh object and an integer to signify the ASCII precision. In order to save the point cloud data as a PLY mesh using this functions, the point cloud object must first be converted to a mesh object.

4.1.1 Greedy Projection Triangulation Mesh

The first method used to achieve this was a greedy projection triangulation algorithm implemented in the point cloud library. This algorithm assumes locally smooth surfaces and relatively smooth transitions between areas with different point densities when creating meshes. In order to run this algorithm, the point clouds normals are first estimated and stored in alongside the point cloud data. A kd-tree object is generated using the point cloud vertices along with the estimated normals. The greedy algorithm generates a triangular mesh by connecting neighbouring points in the tree based on the following parameters:

- Maximum Nearest Neighbours: Sets the maximum number of nearest neighbors to be searched for.

- Maximum Surface Angle: Disregards points for triangulation if their normal deviates more than this value from the query point's normal.
- Minimum Angle: Sets the minimum angle each triangle can have.
- Maximum Angle: Sets the maximum angle each triangle can have.
- Normal Consistency: A flag that determines if the input normals are oriented consistently.
- SearchRadius: The nearest neighbors search radius for each point and the maximum edge length.
- Mu Value: The nearest neighbor distance multiplier to obtain the final search radius.

When this algorithm was run it successfully produced a mesh object for each of the point clouds presented as input. These mesh objects were then written as PLY meshes using the savePLYFile function to produce a 3D mesh of each frame. However regardless of how the parameters were altered (i.e increasing the Maximum Nearest Neighbours, Search Radius and Mu Value) the meshes that were output always contained holes (see Fig. 4.1 and Fig. 4.2).

Figure 4.1: A PLY mesh of a frame's point cloud with a resolution of 320×240 generated using the greedy projection algorithm. This algorithm results in holes appearing in the mesh.

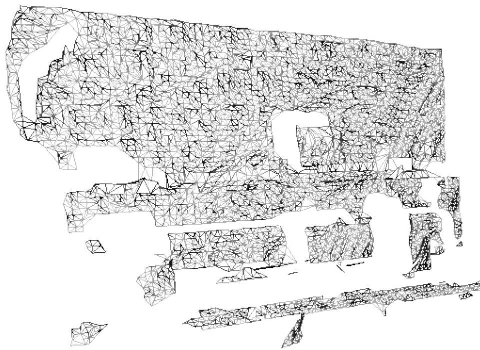

Figure 4.2: A PLY mesh generated from the point cloud of the same frame as 4.1 but recorded with a resolution of 80×60. This mesh was also generated using the greedy projection algorithm. The holes are also evident at this resolution.

This algorithm was also found to be quite slow considering the fact that it was designed for unorganised point clouds rather than the organised dataset that was being worked on in this instance. The algorithm disregarded the fact that point clouds x and y values were increasing from their lower left corners to their upper right. To perform a greedy search for neighbouring points was unnecessary.

4.1.2 Organised Fast Mesh

The second method to create a mesh object from the point cloud was to use the OrganisedFastMesh object which is applied to the point cloud in much the same way as the Greedy Projection Triangulation algorithm. Again this algorithm is featured as part of the Point Cloud Library. This method does not require a kd-tree to be created and relies on neighbouring points being incremental along the x and y axes to generate the mesh. The following parameters are required to be set to generate the mesh from the point cloud:

- Triangle Pixel Size: Sets the edge length (in pixels) used for constructing the fixed mesh.
- Store Shadowed Faces: A flag which determines whether shadow faces are created or not.
- Triangulation Type: Denotes which triangulation method should be used (e.g Quad mesh).

While this function did speed up the conversion from a point cloud to a mesh significantly, the problem of the mesh containing holes persisted. It was also noted that because the depth data for each frame was put through the Kinect SDK's Depth to 3D Space function to alleviate radial distortion(see chapter 2); depth pixels with null values were being given x,y and z positional data of value (0,0,0). This caused a scenario where any holes in the mesh were being connected back to the origin, creating a cone effect(see Fig. 4.3 and Fig. 4.4).

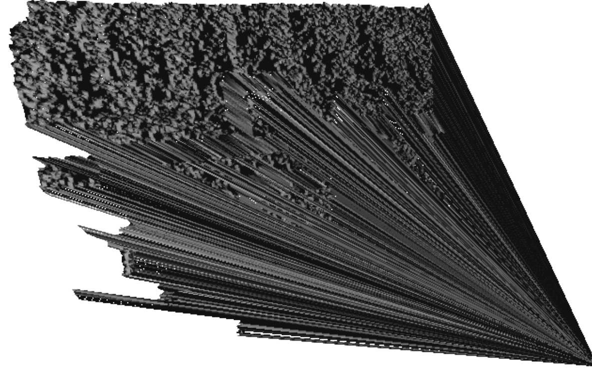

Figure 4.3: A PLY mesh generated from a frame's point cloud using the Organised Fast Mesh algorithm. This mesh has a resolution of 320×240. The 'cone' effect is evident here where points bordering holes are being connected to the origin.

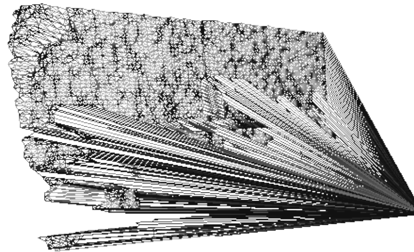

Figure 4.4: A PLY mesh generated from a frame's point cloud using the Organised Fast Mesh algorithm at the 80×60 resolution. Again, the 'cone' effect is evident at the lower resolution.

This function was also applying a transformation based on the cameras position which was altering the orientation of the meshes produced. In order to remedy these issues a new custom PLY writer function was created.

4.2 PLY Mesh Writer

4.2.1 Point Cloud Data vs Depth to 3D Space

A custom PLY mesh writer function was initially written in order to write point clouds as PLY meshes by connecting neighbouring points with the intention of avoiding holes appearing in the meshes. When it was noted that the Kinect SDK's Depth to 3D Space function was causing the problems mentioned above it was made clear that the raw depth data should be used to generate point clouds and should be stored as PLY meshes rather than the point clouds produced by the Depth to 3D Space function. While these frames would contain radial distortion, that issue could be tackled later. To generate the PLY mesh, the depth value at each pixel is taken as the z coordinate value and the x and y values are directly related to that pixels position in the frame.

4.2.2 Process Description

Before the data can be written as a PLY mesh it must first be processed correctly based on the frames resolution. The program has the capability to write depth frames captured at resolutions of 80×60 and 320×240 as PLY meshes of quads. It also performs basic hole filling to avoid the problem mentioned above. Firstly the depthArraytoPointCloud function determines the resolution of the frame and sets a maximum width variable to the width of the resolution. This function then cycles through the depth pixels taking their values and storing them as the z component of an array of 3D points contained in an output point cloud object. The x value to be stored is incremented each time until it reaches the maximum width value at which point it is reset to zero and the y value is incremented. This ensures that the output array begins at position (0,0) populating each row incrementally from left to right until it reaches position (80,60) or (320,240) depending on the resolution.

It is during the population of this output array that a basic hole filling check is

implemented. If the depth value at a certain pixel is equal to zero then the value of the previous pixel is put in its place. This gives generally positive results. The depth values are also multiplied by a different multiplier value depending on the resolution in order to keep the depth ratio the same for the different resolutions. This function produces a point cloud object with the raw depth data stored incrementally.

This point cloud is input as a parameter of the writePLY function along with the resolution of the frame and the current frame number. This function uses the c++ 'iostream' library, and 'boost' library function 'lexical cast' in order to efficiently output the depth data as a PLY mesh file. The file name simply consists of the word 'frame' concatenated with the frame number cast to a string and the characters '.ply'. The function keeps track of three string variables; 'header', 'body' and 'faces' as well as two integers for the face count and vertex count . The function is limited to only one 'for' loop that generates the vertex list and faces list simultaneously, concatenating each entry to the the body string and faces string respectively. The face count and vertex count are incremented each time a face or vertex is concatenated to their respective lists. Once the loop reaches the last entry in the point cloud's array the header is generated using the face count and vertex count. The complete header, body and faces strings are then output in the correct order to the file. By generating the PLY file in this manner and limiting the function to a singular 'for' loop, the task of generating the two lists along with the header file is completed in *O(n)* time.

When generating the list of faces, each vertex[i] is connected to its neighbours to the east[i+1], southeast[i+frame width+1] and south[i+frame width]. To stop a situation where vertex[i] on the edge of a row is connected to the first vertex of the next row[i+1], the following check is imposed: if the modulus of [i+1] and [frame width] is not equal to zero then do not connect [i to i+1]. Fig. 4.5 displays a mesh written using the writePLY function without hole filling where Fig. 4.6 displays the same mesh with the holes filled.

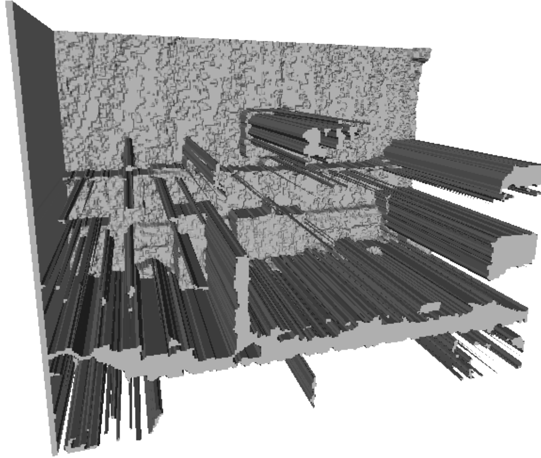

Figure 4.5: A PLY mesh of a point cloud generated from a depth frame with a resolution of 320×240 without the simple hole filling implementation. Areas with no depth value appear closest to the screen

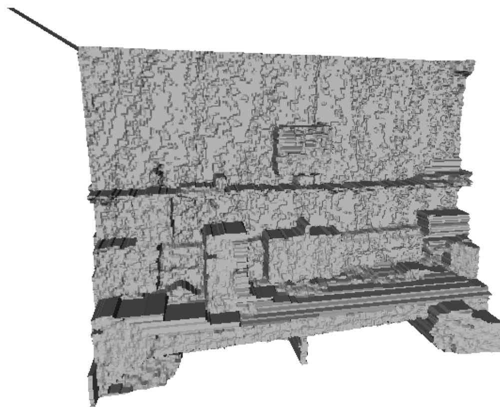

Figure 4.6: A PLY mesh of a point cloud generated from the same frame as Fig. 4.5 with a resolution of 320×240 with the hole filling implementation. Problem areas are no longer visible

Chapter 5

Registration

5.1 Using Gaussian Registration Code

Implementations of the robust point set registration algorithm using gaussian mixture-models as described by Jian and Vemuri[11] are available online[26]. Full source code is implemented in Python, MATLAB and C++. As an alternative to the Iterative Closest Point algorithm, this implementation was demonstrated on the vertex data of the ply meshes that were output from the modified MeshCreator.exe tool described in the previous chapter.

5.1.1 Installing VXL library

It is a requirement of the C++ implementation that the VXL library be installed before that code can be run. The C++ implementation makes heavy use of the data structures and functions provided by the vnl library in particular in order to execute the algorithm. The vnl_matrix and vnl_vector structures and function such as the determinant and iterate functions are used throughout the code. Installing the VXL library proved to be quite difficult. The latest release of the library's source can be downloaded from its google code repository using an Apache subversion client such as TortoiseSVN. The library then needs to be built using the CMake cross-platform, open-source build system [27], and the Microsoft Visual Studio 2010 compiler. This process is quite straight forward and results in a visual studio .sln file that should allow the different components of the VXL library to be built in Visual Studio. While the

VXL core libraries claim to be independent of one another this was found not to be the case in practice and some libraries could not be built as one file had a bug where it could not be built using Visual Studio 2010.

The bug was eventually discovered in the vcl_cstudio.h file on line 28. The line #elif defined(VCL_VC_7) || defined(VCL_VC_8) || defined(VCL_VC_9) must be appended with || defined(VCL_VC_10) in order for the library in question, vil.lib to build in Visual Studio 2010 and in turn for the rest of the libraries to build. Once built, the libraries can be linked to the gaussian mixture registration Visual Studio project which is also generated using Cmake.

5.1.2 Finding Correspondence

The first of the C++ demonstration samples that was successfully run was the extract correspondence algorithm. While finding the correspondence between point clouds is not a requirement for this type of registration, the data produced is still of interest and could be used as possible control points for the gaussian mixture model registration. This will be discussed further in the next section. The extract correspondence program takes two point cloud files (one model and one scene) consisting of ASCII coordinate values to denote the vertexes as well as a threshold value. These ASCII files were created simply by stripping the ply meshes produced by the MeshCreator.exe program of their header and faces, leaving only the vertices. These stripped files were given the .asc file extension and can be visualised in MeshLab. The program computes the square distance matrix between the model and scene and simply returns pairs with a distance below the specified threshold.

5.1.3 Registration on Point Clouds

Attempts were made to run the C++ source code implementation of the gaussian registration code on two .asc point cloud files. However, the program could not accept the input arguments when run in command prompt or in Visual Studio. Ultimately the pre-built gmmreg_demo.exe program that comes with the source code was run using MATLAB as described by the README file. The program takes a .ini configuration file and a string to denote which transformation model to perform. The two types of transformation that were tested were the standard rigid transformation and the non

rigid transformation using thin plate splines. In order to run this code the following commands must be entered in the MATLAB command window while in the directory of the gmmreg_demo.exe file :

exe_file = '.\gmmreg_demo.exe'
followed by either:
gmmreg_demo(exe_file,'configuration_file.ini','rigid') for the rigid transformation or
gmmreg_demo(exe_file,'configuration_file.ini','TPS_L2')
for the transformation using thin plate splines.

In the configuration file the .asc model and scene input files are specified. For non rigid transformations an optional third input file can be specified featuring control points to speed up the transformation, otherwise all of the values in the model file are used as control points. The correspondence between the scene and model file described in the last section could potentially be used here, this was not tested however. Optional initial transformation parameters can also be provided but if left blank, default parameters corresponding to the identity transform are used. These parameters were left blank during all testing. Finally the output models can be specified along with files in which to output the transformation either rigid or the spline transformation and affine transform if 'TPS_L2' is selected.

5.1.4 Results

Rigid Transformations

The gaussian mixture model registration using a rigid transformation was performed on two 80×60 point clouds generated from frames of depth video four frames apart. The transformation took 40.981435 seconds to perform. The results of this transformation can be seen in Fig. 5.1. Fig. 5.2 shows the MATLAB output of the transformation with frame 850 coloured in blue and frame 854 coloured in red before and after the transformation.

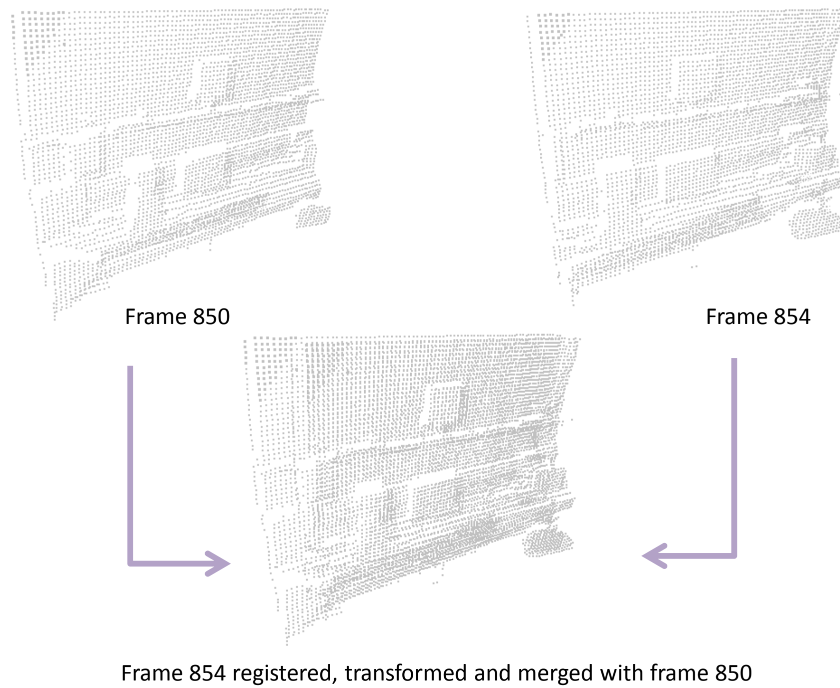

Figure 5.1: Gaussian registration and a rigid transformation performed on two 80 $\times$ 60 point clouds generated from frames 850(scene) and 854(model) as visualised in MeshLab

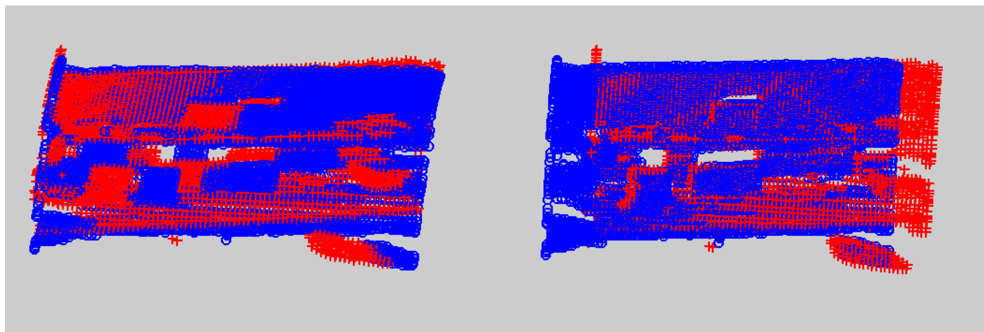

Figure 5.2: The MATLAB output of the rigid transformation seen in Fig. 5.1 from a slightly aerial perspective. Frame 850 is coloured in blue frame 854 is coloured in red. Frames before the rigid transformation are seen on the left; frames after the transformation are seen on the right.

The gaussian mixture model registration using a rigid transformation was also performed on two 320 $\times$ 240 point clouds generated from frames of depth video seven frames apart. This transformation took 9433.589060 seconds (˜ 157minutes) to perform. The results of this transformation can be seen in Fig. 5.3. Due to how densely populated the meshes are, the lighting in MeshLab could not display both clouds properly, therefore the point set data from the frame 77 model appears much darker in this figure. Fig. 5.4 shows the MATLAB output of the transformation with frame 70 coloured in blue and frame 77 coloured in red before and after the transformation.

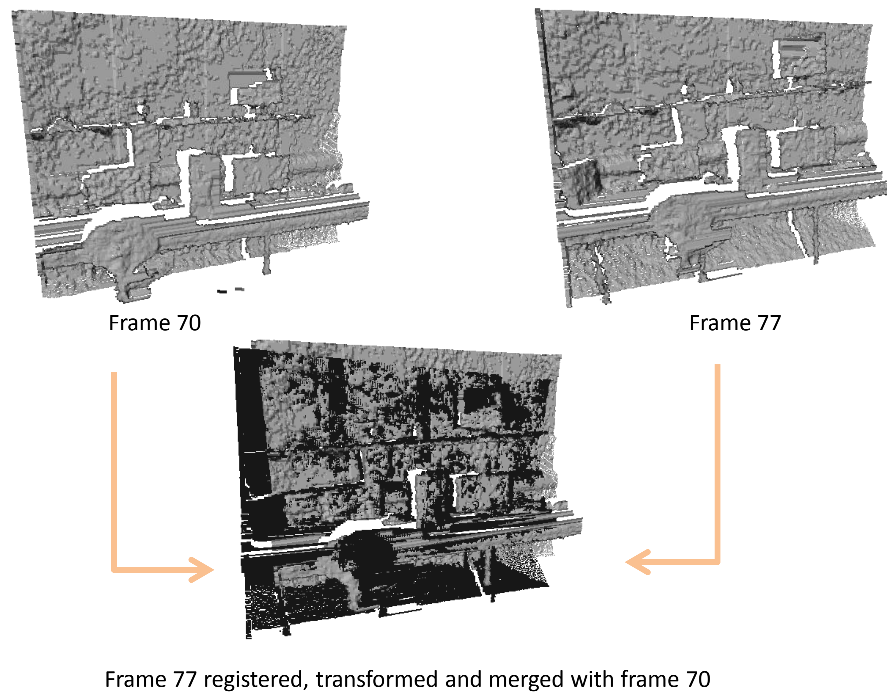

Figure 5.3: Gaussian registration and a rigid transformation performed on two 320 $\times$ 240 point clouds generated from frames 70(scene) and 77(model) as visualised in MeshLab. The transformed frame 77 model appears darker in the merged point cloud.

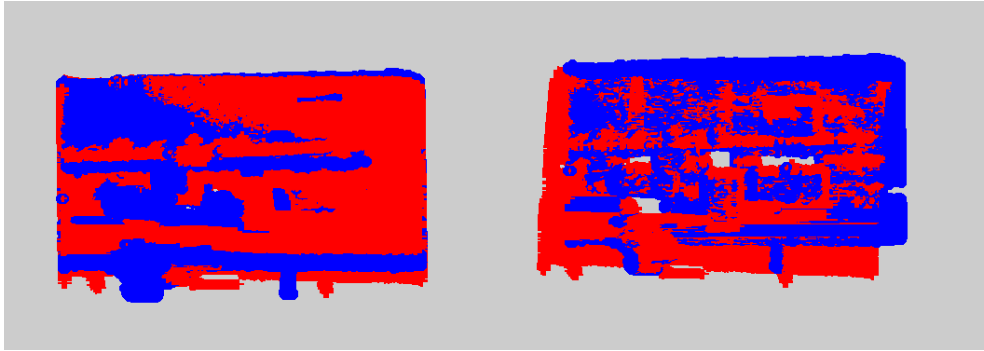

Figure 5.4: The MATLAB output of the rigid transformation seen in Fig. 5.3 from a slightly aerial perspective. Frame 70 is coloured in blue frame 77 is coloured in red. Frames before the rigid transformation are seen on the left; frames after the transformation are seen on the right.

Non-Rigid Transformations

The gaussian mixture model registration using a non-rigid thin plate spline transformation was performed on two 80 × 60 point clouds generated from frames of depth video four frames apart. This transformation took 15466.190239 seconds (˜258minutes) to perform however. The results of this transformation can be seen in Fig. 5.5. Fig. 5.6 shows the MATLAB output of the transformation with frame 850 coloured in blue and frame 854 coloured in red before and after the transformation.

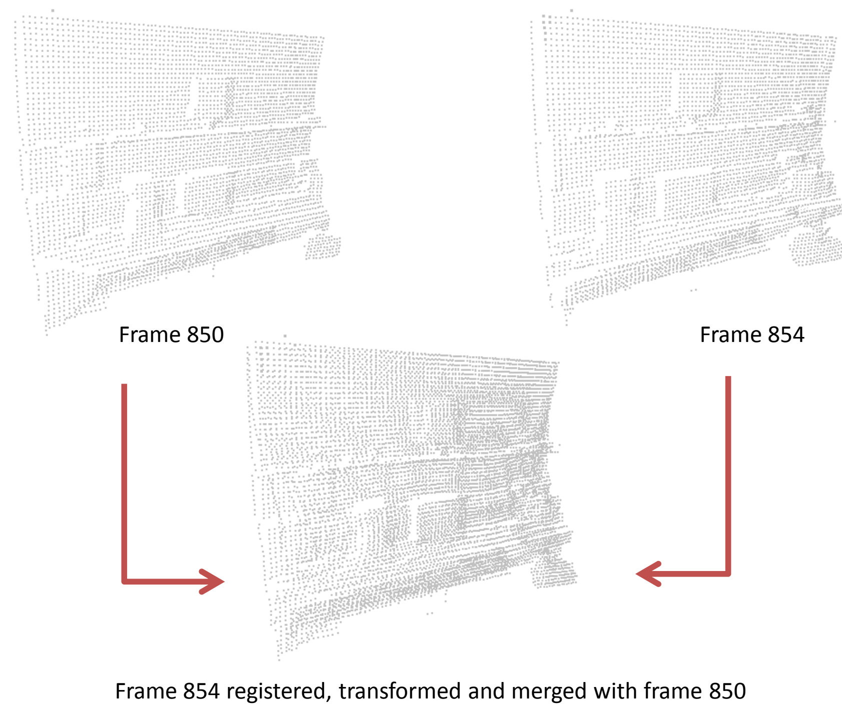

Figure 5.5: Gaussian registration and a non-rigid transformation using thin plate splines performed on two 80 $\times$ 60 point clouds generated from frames 850(scene) and 854(model) as visualised in MeshLab. The model point cloud is distorted slightly in order to align with the scene as expected with this type of transformation

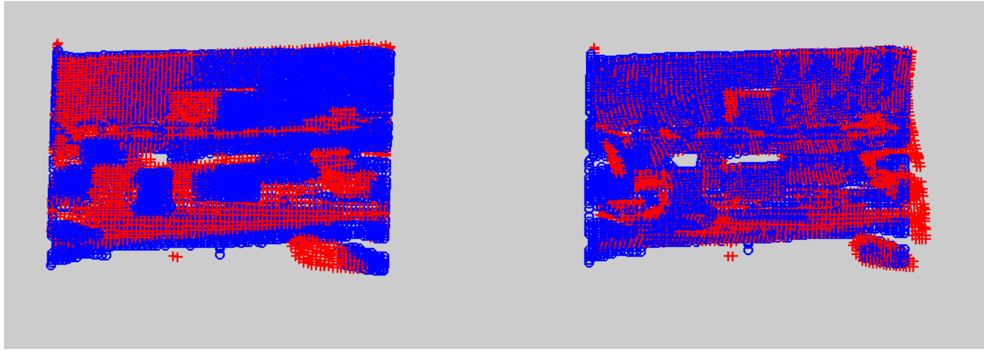

Figure 5.6: The MATLAB output of the non-rigid transformation seen in Fig. 5.5 from a slightly aerial perspective. Frame 854 is coloured in blue frame 850 is coloured in red. Frames before the rigid transformation are seen on the left; frames after the transformation are seen on the right.

The gaussian mixture model registration using a non-rigid thin plate spline transformation was again performed on two 80×60 point clouds but this time were spaced 76 frames apart. This time the transformation failed to align the point clouds correctly as the disparity between the frames was too great. The frames also had some information repeating such as the monitors on the desk and the chairs which most likely contributed to the failure of the algorithm. The results of this transformation can be seen in Fig. 5.7. Fig. 5.8 shows the MATLAB output of the transformation with frame 4 coloured in blue and frame 80 coloured in red before and after the transformation.

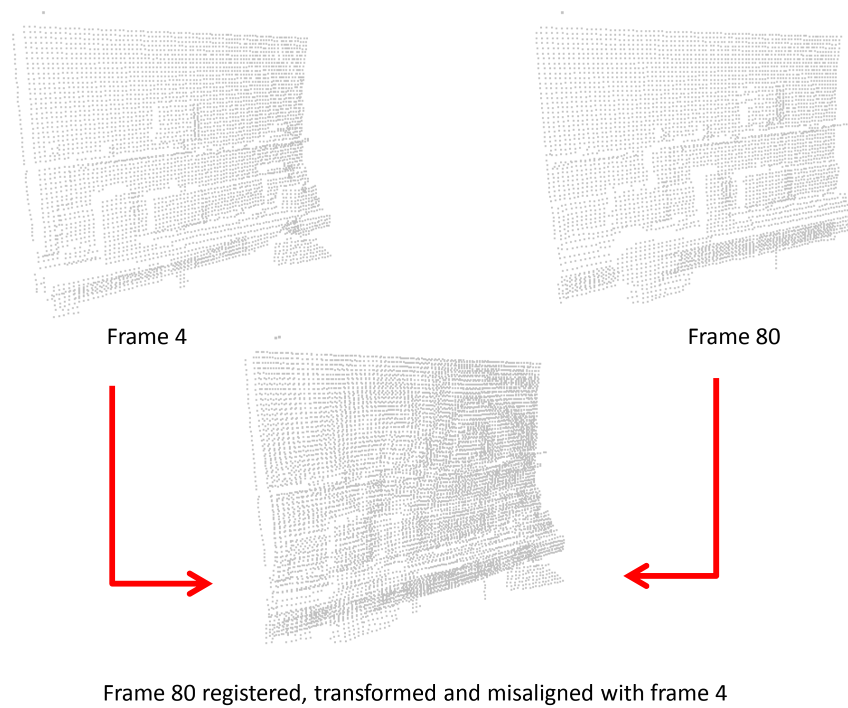

Figure 5.7: Gaussian registration and a non-rigid transformation using thin plate splines performed on two 80 × 60 point clouds generated from frames 4(scene) and 80(model) as visualised in MeshLab. The registration algorithm failed to align the point clouds correctly

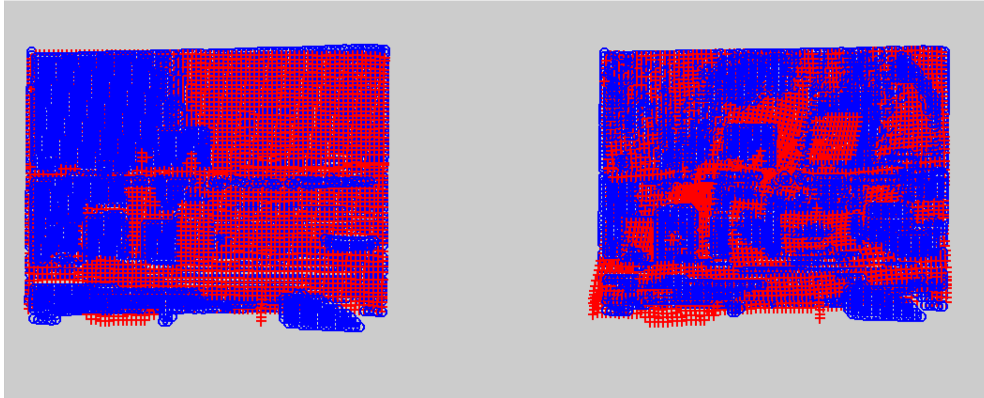

Figure 5.8: The MATLAB output of the non-rigid transformation seen in Fig. 5.7 from a head on perspective. Frame 4 is coloured in blue frame 80 is coloured in red. Frames before the rigid transformation are seen on the left; frames after the transformation are seen on the right.

5.1.5 Registration Completion Time

Due to the fact that the completion time for the non rigid transformation appeared to be taking exponentially longer depending on the number of vertices in the model, a non rigid transformation was not carried out on two 320 $\times$ 240 sized point clouds as it was estimated that it would likely take days to complete on the available hardware. Table 5.1 displays the length of time it took to complete each of the registration computations. One way to speed up the algorithm could be to use control points. Rather than comparing the gaussian values at each of the model point locations, key points could be identified and used instead, thus lowering the overall number of calculations and speeding up the algorithm. While identifying good control points remains outside the scope of this project, the correspondence mentioned earlier in this chapter could prove useful.

Vertices/Point Cloud	Transformation	Completion Time(seconds)
4800	rigid	40.981435
76800	rigid	9433.589060
10	TPS_L2	0.791154
275	TPS_L2	5.440258
800	TPS_L2	470.611227
4800	TPS_L2	15466.190239

Table 5.1: A table displaying the length of time taken in seconds in order to run the registration algorithms with point clouds of different sizes.

Chapter 6

Conclusion

6.1 Summary

The MeshCreator tool in the Kinect Stream Recording application was successfully extended with the functionality to export point clouds as ply Meshes while the tool is running. Three separate executable versions of the MeshCreator.exe program now exist that output the point clouds as ply meshes using the Point Cloud Library's Greedy Projection Triangulation algorithm, the Point Cloud Library's Organised Fast Mesh algorithm and the custom PLY Mesh writer that was developed respectively. Basic hole filling was also implemented in the PLY mesh writer. Each of these versions of the program are capable of outputting the point clouds along with their edge data, as meshes when given depth videos as input with resolutions of 80×60 and 320×240. It was demonstrated that these ply meshes could be visualised in MeshLab.

It was shown that at least two of these ply meshes could be stripped of their faces and headers and used as input for the Gaussian Mixture Model Registration. During this process one of the two point clouds was treated as a model and the other a scene. The model was accurately transformed using a rigid transformation based on the output produced by the Registration code. In all cases when a rigid transformation was used the code was successful in aligning the model point set with the scene point set. The registration algorithm failed to align the model with the scene on one occasion when using the non-rigid transformation using thin plate splines. This transformation also appeared to warp the model point set in order to align it with the scene which does not

produce good results when attempting to build an entire point cloud scene comprised of different point clouds. The final aligned point clouds were successfully visualised and merged in MeshLab.

6.2 Discussion of Results & Future Work

On reflection having imlpemented hole filling at the meshing stage, it is clear that this is not the best course of action. In order to create a full mesh, it is desirable to fill the holes based on the point cloud data of the other frames that possibly contain that missing data rather than the neighbouring vertices of those for which there is no value. The problem of meshing the final merged cloud was not addressed and left for future work. One suggestion would be to use the Greedy Triangulation Algorithm on the merged scene point cloud. The likelihood of holes appearing in a mesh produced with this algorithm would decrease with each additional point cloud registered and merged with the scene.

The gaussian mixture model registration code was not successfully implemented in the MeshCreator tool due to time constraints. It was also found that should the algorithm be implemented it would be too slow to run in a real time application unless control points were used. As suggested previously the correspondence between the model and scene point clouds could be used as robust control points but this would need to be studied further. A corner detection algorithm could be used to pick out key points in the model and scene and these could potentially be used as control points. Alternatively an algorithm much like RANSAC [28] could be implemented that iteratively takes a series of random points relying on the probability that most will be inliers. These random point sets could be used as control points for a certain amount of iterations and the best transformation could be used. Considering that the point set data is being interpreted as mixtures of gaussians rather than the discrete points themselves, the influence in the objective function of the outlying points is naturally being suppressed so this may be a good course of action.

The problem of radial distortion caused by the pinhole camera and its affect on the reliability of the registration algorithm was never addressed. An alternative to the Kinect SDKs depth to 3D space function could be explored or a calibration matrix could be generated manually for each point cloud in order to eliminate the distor-

tion. Alternatively each point cloud could be treated as a convex hull and a specific registration algorithm could use this fact when determining the correct transformation.

Finally it was an intention to explore the concept of adding colour to the meshes but due to time contraints this never came about. The Kinect Stream Recording application records both the depth and RGB values of pixels in each frame. The RGB values of many frames could be taken into account when determining the correct colour for a certain vertex. Vertex and Pixel shaders could be written to apply these colours to the mesh or large textures could be generated in tandem with the mesh and applies at a later stage.

Appendix

.1 USB Flash Drive Contents

The attached USB flash drive contains two folders '**code**' and '**data**'

.1.1 code

The '**code**' folder contains four sub-folders for each of the three versions of the code:

- **OrganizedFastVersion**
- **GreedyVersion**
- **PlyWriterFinalVersion**
- **DaveOriginalVersion**

Each of these subfolders contains the source code, a Release folder that contains the .exe and a '**datadump**' folder that contains the output produced by that version. The '**code**' folder also contains the Visual Studio Dissertation.sln file which runs the PlyWriterFinalVersion of the code and the PCD Reader that was written but abandoned.

.1.2 data

The '**data**' folder contains two sub-folders called '**interestingdata**' and '**olddata**'. The '**olddata**' folder contains some old pcd point cloud files as well as some other old meshes produced from early in development and can most likely be ignored.

The '**interestingdata**' folder comprises of a '**meshes**' folder and a '**registration**' folder

The '**meshes**' folder contains a single example of all the possible outputs of each of the programs and is well labelled. These .ply meshes can be viewed in MeshLab.

the '**registration**' folder contains the following folders:

- **non rigid transform that didnt work**
- **rigid transform 320x240**
- **rigid transform 80x60**
- **non rigid transform 80x60**

Each of these folders contain the MATLAB output figure of the transforms, aswell as a .asc file for the point cloud before transformation, .asc file for the point cloud after transformation and a .asc for the scene point cloud.

The best way to view these files is to load all three .asc files into MeshLab at once and click on view - show layers dialogue. Then the different point clouds can be hidden by clicking on the eye icon. To view the fully merged scene just leave the transformed model and the scene model visible.

Bibliography

[1] Ganter D. Kinect stream recording, point cloud extraction. *M.Sc. Dissertation*, 2013.

[2] Microsoft. Msdn. *http://msdn.microsoft.com/en-us/library/dn188670.aspx*, 27-08-2013.

[3] Microsoft. Kinect for windows sdk. *http://www.microsoft.com/en-us/kinectforwindows/*, 29-08-2013.

[4] Open Source. Openkinect collaboration. *http://openkinect.org/wiki/*, 29-08-2013.

[5] Herrera C. D., Kannala J., and Heikkil J. Joint depth and color camera calibration with distortion correction. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 34:2058 – 2064, 2012.

[6] Schuon S., Theobalt C., Davis J., and Thrun S. High-quality scanning using time-of-flight depth superresolution. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, 2008.

[7] Franca J.G.D.M., M.A. Gazziro, Ide A.N., and Saito J.H. A 3d scanning system based on laser triangulation and variable field of view. *Image Processing, 2005. ICIP 2005. IEEE International Conference on*, 2005.

[8] Hartley R. and Zisserman A. Multiple view geometry in computer vision. *Cambridge University Press*, 2004.

[9] Rusinkiewicz S. and Levoy M. Qsplat: a multiresolution point rendering system for large meshes. *Siggraph 2000. ACM*, 1:343–352, 2000.

[10] A. Laurentini. The visual hull concept for silhouette-based image understanding. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 1:150–162, 1994.

[11] Bing Jian and Vemuri B. C. Robust point set registration using gaussian mixture models. *Pattern Analysis and Machine Intelligence, IEEE Transactions*, 33, 2011.

[12] S. Izadi, D. Kim, O. Hillige, D. Molyneaux, R. A. Newcombe, P. Kohli, J. Shotton, S. Hodges, D. Freeman, A. J. Davison, and A. Fitzgibbon. Kinectfusion real-time 3d reconstruction and interaction using a moving depth camera. *ACM symposium on User interface software and technology*, pages 559–568, 2011.

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

[14] Newcombe R. A., Davison A. J., Izadi S., Kohli P., Hilliges O, Shotton J., and Molyneaux D. Kinectfusion real-time dense surface mapping and tracking. *Mixed and Augmented Reality ISMAR*, pages 127 – 136, 2011.

[15] P.J. Besl and N.D. McKay. A method for registration of 3d shapes. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 14:239–256, 1992.

[16] K.S. Arun, T.S. Huang, and S.D. Blostein. Least-squares fitting of two 3d point sets. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 9:698–700, 1987.

[17] Arellano C. and Dahyot R. Mean shift algorithm for robust rigid registration between gaussian mixture models. *Signal Processing Conference (EUSIPCO), 2012 Proceedings of the 20th European*, pages 1154–1158, 2012.

[18] Triggs B., McLauchlan P., Hartley R., and Fitzgibbon A. Bundle adjustment - a modern synthesis. *Proceedings of the International Workshop on Vision Algorithms: Theory and Practice ICCV*, 1999.

[19] Snavely N., Seitz S. M., and Szeliski R. Modeling the world from inter-net photo collections. *International Journal of Computer Vision*, 2008.

[20] Crandall D., Owens A., Snavely N., and Huttenlocher D. Discrete-continuous optimization for large-scale structure from motion. *Conferenceon Computer Vision and Pattern Recognition (CVPR)*, 2011.

[21] Willow Garage. The point cloud library. *http://pointclouds.org/*, 29-08-2013.

[22] Visual Computing Lab of ISTI CNR. Meshlab. *http://meshlab.sourceforge.net/*, 29-08-2013.

[23] Willow Garage. Opencv - the open source computer vision library. *http://opencv.willowgarage.com/wiki*, 27-08-2013.

[24] MathWorks. Matlab. *http://www.mathworks.co.uk/products/matlab/*, 29-08-2013.

[25] Open Source. Vxl - c++ libraries for computer vision research and implementation. *http://vxl.sourceforge.net/*, 29-08-2013.

[26] Bing Jian and Vemuri B. C. Robust point set registration using gaussian mixture models. *https://code.google.com/p/gmmreg/*, 27-08-2013.

[27] Open Source. Cmake, the cross-platform, open-source build system. *http://www.cmake.org/*, 27-08-2013.

[28] Fischler M. A. and Bolles R. C. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24:381–395, 1981.