Kinect Stream Recording, Point Cloud Extraction and Point Cloud Registration — Report

Master in Computer Science (MCS)
School of Computer Science and Statistics
University of Dublin
Trinity College

David Ganter
08562822
Supervisor: Dr. Rozenn Dahyot

Submitted to the University of Dublin, Trinity College
May 2013

Summary

This dissertation presents a tool to aid researchers who are using or plan to use to Kinect to record the data streams separately from their goal application. In cases where the application they plan to use Kinect streams for is not complete yet or it is not practical in terms of processing power to do online computation, this tool provides a method of taking data and working on it offline.

The tool is stand-alone, stable, reliable, without dropping frames. It is flexible in what streams and at what resolutions are recorded. The file structure used is open and non-complex. Due to the nature of the data it is easy compressed post recording to very manageable sizes. The tool is open source and is hosted on GitHub with documentation of the file structure. The tool is Windows platform only for simplicity with cross-platform compatibility being out of scope.

This report provides insight into what to expect in terms of data size from recordings at length. This hopefully will allow researchers to choose their resolution and streams carefully depending on their needs.

Futher to this, point cloud extraction from depth image is described and a point cloud registration method is outlined, which was an add- on to the dissertation, not initially being in scope.

Declaration

I, David Ganter, declare that the following dissertation, except where otherwise stated, is entirely my own work; that it has not previously been submitted as an exercise for a degree, either in Trinity College Dublin, or in any other University; and that the library may lend or copy it or any part thereof on request.

Signature

Date

Attestation

I understand the nature of plagiarism, and I am aware of the University's policy on this. I certify that this dissertation reports original work by me during my University project except for the following :

- The Windows application code and Kinect SDK code used in the Recorder discussed in Chapter 2 was created by Microsoft.

Signature

Date

Acknowledgements

I wish to thank my supervisor, Rozenn Dahyot, for providing me with outlook and insight into the dissertation and passing on a wealth of knowledge. To my classmates who have become friends and allies, charging through the challenges again and again, thank you. To my parents, thank you for encouraging me and putting up with me. And finaly to my girlfriend who has been so understanding and caring during the entire process and one of the best advisors there is, thank you dearly.

Contents

1 Introduction **1**

1.1 Laser Scanning 1

1.2 Uses of 3D Scanning 2

1.3 The Kinect 2

1.4 Dissertation Aim 3

1.4.1 Stream Recording 3

1.4.2 Point Clouds 3

1.5 Dissertation Tools 4

2 State of The Art **5**

2.1 3D Scanning 5

2.2 Stream Recording 7

2.3 Scene Reconstruction 7

3 Recording Streams **9**

3.1 Blocking and Non-blocking I/O 9

3.2 File Format 10

3.3 Threaded File Writer 12

3.4 Data Sizes 13

3.5 Compression 15

3.6 Kinect Wrapper 16

3.7 MatLab Function 17

4 Application Front-End **18**

5 Point Clouds **21**

5.1 Depth Field Projection 21

5.2 Point Cloud Registration 23

6 Future Work & Discussion **25**

7 Conclusion **27**

Chapter 1

Introduction

Typically, when we see a virtual 3D object in a game, an animated movie or most applications that deal with 3D models, we can most often assume that it was modelled mostly by hand by a skilled 3D artist in software like 3DS Max, Blender or Maya. For complex objects or characters this can be tedious work, manipulating the mesh vertex by vertex, or in small groups of vertices.

3D scanning can be used as a method of creating a virtual 3D structure of a real world object. In an ideal system, a user can place an object into a real world scene, scan it with a 3D scanner, and then use the 3D reconstruction of the object for whatever means. Due to the size and bulk of many 3D scanners, this is a cumbersome job and far from perfect.

This chapter will briefly cover some of the basics of 3D scanning without getting bogged down in too much detail; anything that would need to be covered in depth is discussed in Chapter 2: State of The Art.

1.1 Laser Scanning

One method of obtaining a depth field is a laser scanner; a high precision device which uses a scanning laser to accurately determine the distance of a point of an object relative to the device. One of the most common methods of laser scanning is to "steer" the beam across two dimensions over the object, generating a depth field. Given the intrinsic parameters of the scanner, this depth field can then be projected into a point cloud.

Scanning from different angles and applying some form of transformation to the data sets, we can retrieve a point cloud which can be used as a virtual 3D

representation of a real life object. This transformation between the data sets can be determined in a multitude of ways such as pre-determining the camera position at a certain time of scanning or using the data sets to estimate the extrinsic camera parameters[15, 2, 10] which can be done offline or online[9, 13].

1.2 Uses of 3D Scanning

3D scanning can be used in a multitude of ways. An example of 3D scanning in action is preserving cultural heritage, such has happened with Michaelangelo's Statue of David and other statues by the same sculptor[12], using precise laser scanning to make a virtual version of the statue which can be used for historical records and research. In this case, the researchers used laser a scanner accurate to 50 microns to not only get the general shape of the statues, but to pick up the individual chisel marks which add character to the visual appearance of the works. In this example the scanner used was cumbersome and most definitely not very portable.

Historical conservation aside, another area that 3D scanning could be considered is the sale of goods online, where currently it is commonplace to see pictures of an item for sale on websites such as Ebay or Amazon. In this case, the seller could 3D scan the object they are willing to sell with a 3D scanner and upload a 3D model, generated one way or another, to the website, allowing potential buyers to rotate and zoom on the model and get a better feel of the object they might purchase. This being said, the scanner used in this situation would have to be portable and cheaper than most laser scanners.

1.3 The Kinect

With the expansion of areas in interactive entertainment comes the creation of new technology, most of which is designed to enhance the experience of the end user. One such technology is the Microsoft Kinect, a device which provides a standard RGB camera, a microphone array and an infrared depth field camera, the latter of which is of most significance to this dissertation.

The Kinect is primarily a user interface designed for games, allowing users the freedom of using their own body as a controller, rather than some handheld device. Although this is not a new idea, with the likes of the Playstation Eye Toy predating the Kinect, the method of identifying the player was slightly different, as described

previously with an infrared depth field camera, as well as a standard RGB camera.

The Kinect provides a device which is a great deal more portable than traditional laser scanners, sacrificing some accuracy for this portability especially at distances reaching the edge of the device's maximum depth distance[11].

1.4 Dissertation Aim

1.4.1 Stream Recording

The first aim of this dissertation is to provide a flexible tool for researchers or anyone else who would intend to record the Kinect streams for offline processing. This tool should provide reliable recording without dropping frames or losing data[1]. The tool should also have an easy to use interface that allows the end user to select varying degrees of resolution, if any, of both the depth stream and the RGB stream individually of each other. The format that this tool records to should be non-complex so as to be relatively easily imported into any further applications that use the tool. A function for this file format should also be implemented for applications that work on the recorded data in MatLab.

The stream recording and file format of the recording tool is covered in chapter 3 and the user interface part of the tool along with the MatLab function is covered in chapter 4.

1.4.2 Point Clouds

When all recording is done, the next stage of the dissertation is to generate point clouds from the depth streams. Projecting the points from the depth field can be done in a few ways, the process of which is covered in chapter 5, section 5.1.

After projecting from depth field to point cloud the next stage is to reconstruct a virtual scene[2] from a sequence of point clouds. This process is called point cloud registration, where an algorithm is used to estimate the rigid transformation between two point clouds which can then be used to merge the two point clouds so they overlap. This process is documented in chapter 5, section 5.2. This registration process can then be used to extract objects and transform the point cloud data

[1]This could occur if lossy compression is used or loss of precision from type to type.

[2]To make a distinction, a real world scene is just called a scene and a virtual reconstructed scene will be called a virtual scene in the report.

into surface information to recreate a 3D mesh, which can then be used in some applications described in section 1.2.

1.5 Dissertation Tools

Over the duration of the dissertation a number of tools were used for development. The language of choice was C++, using the Visual Studio IDE and Visual Studio tools for creating dialogs. For version control git and GitHub[3] were used, with the GitHub wiki being used for documentation. Libraries used include the official Kinect SDK[4], Boost C++ Libraries[5], Eigen[6], PCL[7] and POSIX standard threads[8].

[3] *www.github.com/ganterd/dissertation*
[4] www.microsoft.com/en-us/kinectforwindows/develop/
[5] www.boost.org/
[6] eigen.tuxfamily.org/
[7] pointclouds.org/
[8] computing.llnl.gov/tutorials/pthreads/

Chapter 2

State of The Art

2.1 3D Scanning

Visual 3D scanning of an object can be classified into two groups; passive light scanning and active light scanning[6]. Passive stereo scanning techniques make use of cameras, or a single camera, at multiple positions and finding multiple correspondences between the images and extracting features and depth information from these images[8]. This passive scanning technique makes use of light already in the scene without projecting, although care has to be taken in making sure the scene is lit in such a way that the system has the best possible chance of finding features[6].

Active scanning systems project light information onto the scene to help determine depths. For example, structured light scanning project predetermined light patterns onto the scene and then determine from the distorted light pattern what the surface depths are[19]. One method of doing this is simply using a projector and a camera, with the projector lighting the scene with a black and white pattern[19] or a coloured pattern[7] to avoid occlusion issues. The Kinect is a structured light device, emitting a speckle pattern of infrared light in a determined pattern. The projected infrared light is then captured by the infrared sensitive camera on the device, extracting depth from the pattern's distortion.

As well as structured light another form of active object scanning is triangulation, where a dot of light is projected onto the object from a laser or other fine light emitting device and the dot of light is then triangulated using a camera[6]. In this scenario, the relation between the light emitter (e.g. laser) and the light receiver (e.g. camera) should be known or calibrated.

Finally an additional method of extracting depth in a scene is to use a time-

of-flight camera, where the emitter sends a pulse of light at the object and the round-trip time is measured[5, 6] and a distance determined by using the speed of light through air constant.

It should be noted that as the above techniques for 3D scanning all take time to complete one scan and, unless multiple devices are used at the same time, which is not a possibly for structured light scanners or time-of-flight scanners as interference from other scanners may throw off results, the scanning of a complete object from multiple angles takes time. In general, because of this fact, the scene should not be dynamic (e.g. scanning for fashion where the human model may not remain still), although depending on the scene reconstruction algorithm and how dynamic the scene is this may not pose a huge issue[13, 9].

Going into more detail about the Kinect, the device provides three streams; a depth stream, an RGBA stream and an audio stream. For the purpose of this dissertation the alpha channel of the RGBA stream was discarded for reasons explained further into the report and the audio stream was discarded as the aim of the dissertation was ultimately about point clouds, of which audio played no part. The depth stream comes in three resolutions:

- 80 by 60 pixels
- 320 by 240 pixels
- 640 by 480 pixels

The above resolutions are all at a frame rate 30Hz, providing a moderately high resolution range image at a fairly high frame rate. The RGB stream can be selected between two resolutions:

- 640 by 480
- 1280 by 960

The former resolution is as a frame rate of 30Hz and the latter, higher resolution is at a slower frame rate of 12Hz[1].

2.2 Stream Recording

The ability to take a depth an RGB recording offline to analyze the data and possibly reconstruct a scene should be an easy task to accomplish, and there are indeed some

tools to do this, however due do cost or constraints, they may not be useful for researchers.

For example, with the official Kinect SDK comes a tool called Kinect Studio which allows you to record the RGB, depth and audio streams from the Kinect. However in order to record these streams an application which is already using the Kinect has to be operational and streaming from the Kinect. The Kinect Studio SDK then taps into this stream, recording it for future use. Another drawback to using this tool is the need of having a Kinect device actually connected to play back recorded files[1], regardless of the use of said Kinect. Constraints like these render the Kinect Studio tool rather useless in situations where offline processing is done on a computer, cloud computing for example, which does not have access to a Kinect.

Aside from Kinect Studio, there are other tools which allow the recording of the Kinect streams, such as NUI[2] Capture[3], formerly known as KinectCapture, which seems to be a comprehensive program for recording Kinect streams, however this tool would not be considered cheap at $279 for an academic license[4].

At the time of writing a free tool that provided a method for recording streams in an easy manner and provided for free or that was open source could not be found.

2.3 Scene Reconstruction

With devices like the Kinect the view frustum is small relative to a scene like a room, and in order to reconstruct a such a scene, the point clouds must be registered, meaning the translation from one point cloud to another must be found to that they overlap correctly. For example, consider the scenario of scanning a room with a device like the Kinect. Sweeping from left to right a point cloud from frame i needs to be rotated and translated to line up with a point cloud from frame $i - 1$.

This process can be done by manually by rotating and translating point cloud i. However, when we consider the fact that a device like the Kinect can record the depth stream at up to 30Hz, this post-processing step would most likely take hours upon hours of manual work.

[1]This information can be found at msdn.microsoft.com/en-us/library/hh855396.aspx

[2]Natural User Interface is the term coined to interfaces which remove boundaries of the GUI, which in turn removed the boundaries of the command-line interface[17].

[3]nuicapture.com

[4]Prices at *nuicapture.com/purchase/* state that a single license is $399 and there is a 30% academic and student discount.

The alternative to registering point clouds manually is automatically with algorithms such as iterative closest point[4, 3]. ICP comes in many variants[14] which improve on the original ICP algorithm[18] but generally follows the same procedure; find a set of corresponding points between point cloud $i-1$ and point cloud i, then find a rigid transformation that minimizes a cost function between the two point clouds.

For example, Microsoft experimented with the Kinect for real-time applications of scene reconstruction with Kinect Fusion[13, 9] which took the ICP algorithm and parallelised the point correspondence part on CUDA implementing real-time SLAM[5]. In their version of the ICP algorithm they not only use the point clouds to find point correspondences but also the range images, assuming only incremental movement from one point cloud to another.

[5]Simultaneous Localization And Mapping

Chapter 3

Recording Streams

As discussed in chapter 1, the first part of the dissertation was to create a tool that provides researchers, developers and others for recording the depth and RGB streams from the Kinect in an easy fashion. The recorded data should be accessible and non-complex so as not to be a hindrance to further development. Further to that, the code for accessing the recording data should be well documented[1]. for any future developments with the application.

This section will go into detail about the development procedure used in back-end of the stream recording tool, the file format used for recording the streams, the file writing mechanism used, the wrapper around the Kinect SDK and file input and the data sizes to be expected from recording.

3.1 Blocking and Non-blocking I/O

There are several factors which should be considered when recording a data stream. Firstly, we shall look at the speed of recording data. This is one of the most important factors in writing out the data stream for two separate reasons.

On one hand, we can choose to *block* the program until what we wish to record is written. In the case of recording the data streams from the Kinect, if the program is blocked for a relatively large period of time there is a high chance that we may miss frames, which is not ideal. Although not very useful for this situation, it can be useful in preventing file corruption. For example, if the program sends data to be written to a non-blocking file writer and then crashes before the data has finished

[1]This documentation can be found on the GitHub wiki (*github.com/ganterd/dissertation/wiki*)

writing out, chances are the data written is not going to be complete, possibly destroying the file structure.

For this tool, time is of the essence as the round trip from sending data to be written must be shorter than the frame rate of the Kinect, otherwise frames will get backed up and eventually dropped by the maximum frame buffer limit of the Kinect.

In terms of libraries, the standard libraries for C++ are considered blocking I/O, as are the file writing mechanisms for OpenCV. Boost C++ libraries provide a method for asynchronous and non-blocking I/O, although when tested this method crashed randomly, but consistently. The final method for file writing is documented in section 3.3, as the file format needs to be covered before going into detail.

3.2 File Format

As part of the stream recording tool, two types file formats were considered for the application. Firstly seeing as a video stream was to be recorded, a video format would seem to be the obvious choice. For this selection care would have to be taken to record the depth stream, as the RGB stream is 8 bits per color whereas the depth stream is 16 bits per pixel. Using this method of stream recording, in order to use the file in future applications the language used must have some form of library to read a certain video type.

The method of writing video files chosen was to use OpenCV, as this has high level C++ functions to write to an AVI file using codecs found on the user's computer. There were a couple of issues with this approach:

1. The codecs on all computers may not be the same. This application is Windows based, so recording to a Microsoft standard AVI file may limit the libraries you can use cross-platform.

2. It assumes that whatever language the data will be used with has easy, if even existent, methods of reading from an AVI file.

The second choice was to use a custom file type using the raw data of the streams and no compression. In this scenario, the data is cast to 8 bit, unsigned chars. For example if we take one depth pixel from the Kinect stream which is in unsigned short format and cast to an unsigned char, the result will be two unsigned chars

which will be written out to disk. When reading in this data, the unsigned chars can be cast back to (or read in as, in the case of MatLab) unsigned shorts, without having to worry about codecs and additional frame fields.

The drawback of this method is coping with different stream resolutions and different amounts of streams, for example one file could have just depth recording at 80 by 60 resolution and another file could have both depth and RGB streams recorded at 640 by 480 and 1280 by 960 respectively, in which case not only will the presence of streams and their resolutions be different, but also the frame rate.

This now makes the file format more complicated, but still relatively simple. We can take a timestamp that the frame is written out at and also cast this to an unsigned char. The time stamp, taken from the standard library time function, is returned as an unsigned long. As there is no advantage to the application in being 64 bit, we can assume that the timestamp is 4 bytes, starting each frame data.

To solve the all-or-some-presence-of-streams-and-their-resolutions issue the start of this custom file contains a short varying length header. This varying length header contains none, one or two of a type and resolution specifier. This type and resolution specifier is exactly 13 bytes long and are plain text chars. The first four bytes of this specifier is either 'DEPTH' or 'RGB '. Note that there are two ASCII spaces after the RGB tag. The next 8 bytes of the specifier define the resolution of the stream, with the following tags:

- '80X60 '
- '320X240 '
- '640X480 '
- '1280X960'

To state the obvious, only the first three tags apply to a depth stream specifier and only the last two tags apply to an RGB specifier. Note that there are three ASCII spaces after the 80 by 60 tag and the two ASCII spaces after the 320 by 240 and 640 by 480 tags.

The frame data is a repeating set of data, with the depth data always coming before the RGB data for simplicity sake. The first 4 bytes of the frame data is the time code, then either the depth and RGB raw data, the size of which depends and is determined by the specified resolution. For example a depth stream of 80 x 60

will have a total of $80 \times 60 \times 2 = 9600$ bytes[2], plus the 4 byte time stamp. Putting this all together, the final custom file structure is as follows: A maximum of two 8-byte stream type and resolution specifiers, followed by the repeating frame data.

This raw data custom file is the final format used in the application.

3.3 Threaded File Writer

As discussed in section 3.1, care must be taken to avoid blocking I/O. To circumvent this issue, a custom threaded file writer was written. This file writer uses POSIX standard threads to run in parallel to the main process, using a queue.

The main process calls the file writer's write function which mutex[3] locks the queue, pushes data onto the queue, and unlocks the mutex. This file writer then, again, mutex locks the queue to avoid simultaneous and dangerous memory changes, takes the next set of data from the queue and releases the lock. This process is described in figure 1.

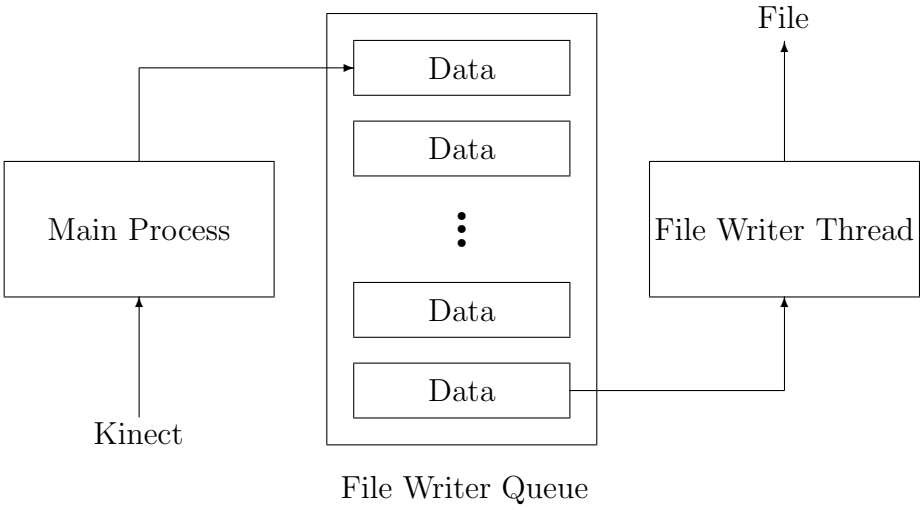

Figure 1: Main process and File Writer Thread run in parallel, using a file writer queue.

When the application starts, the user connects to the Kinect and then selects a new or existing file to write to. In the case of an existing file, the file is overwritten when the file is selected to be written to. The user then selects what streams to

[2]Unsigned short = 2 bytes.

[3]A mutex lock locks a certain section or sections of code so only one process can enter the mutex at any one time. This prevents race conditions and illegal simultaneous data changes.

record and at what resolutions. When the record button is pressed, the header information is written and the looping fetch, cast and send to be written loop is started until the stop button is pressed.

Depending on the system and the resolutions and streams selected the queue may begin to back up. In this case, the queue will continue to increase until the memory of the system is full and the program crashes, however during testing the time it takes to reach this limit is more that should be more than enough for most.

On a standard 7,200 rpm spinning hard disk system[4], recording depth at 640 by 480 does not back up the queue, with the file writer emptying the queue before more data being fetched. When RGB is added to the recording, the queue begins to back up at a variable rate of approximately 1 frame per second.

On a new solid state drive[5] the queue does not back up and the data is written to disk without delay.

3.4 Data Sizes

Aside from the file writer queue being backed up and memory being filled another constraint is the remaining space on the hard disk. This section will go into a small bit of explanation into what to expect size-wise when recording the Kinect streams.

The lowest storage usage is of course using the lowest resolution of the depth stream at 80 by 60. For 60 minutes of recording at this resolution the data size is just under 1 GB. However when we step up to the next highest resolution of depth stream, the data size increases dramatically from under 1 GB to over 15 GB, which can be seen in figure 2.

In comparison, increasing resolution from 640 by 480 RGB to 1280 by 960 RGB only increases the overall size of the recording by under half as can be seen in figure 3. This is due to the cut in frame rate from 30Hz to 12Hz.

This size increase ratio is even more diminished when recording with both RGB and depth. In figure 4, a comparison is made when depth at 640 by 480 is recorded with RGB at 640 by 480 and again at 1280 by 960. Again, this lesser increase is due to the cut in frame rate, but is also assisted by the additional data presented by the depth stream.

This presents a tradeoff between quality over quantity where on one hand you

[4]Western Digital Blue
[5]OCZ Vertex 4

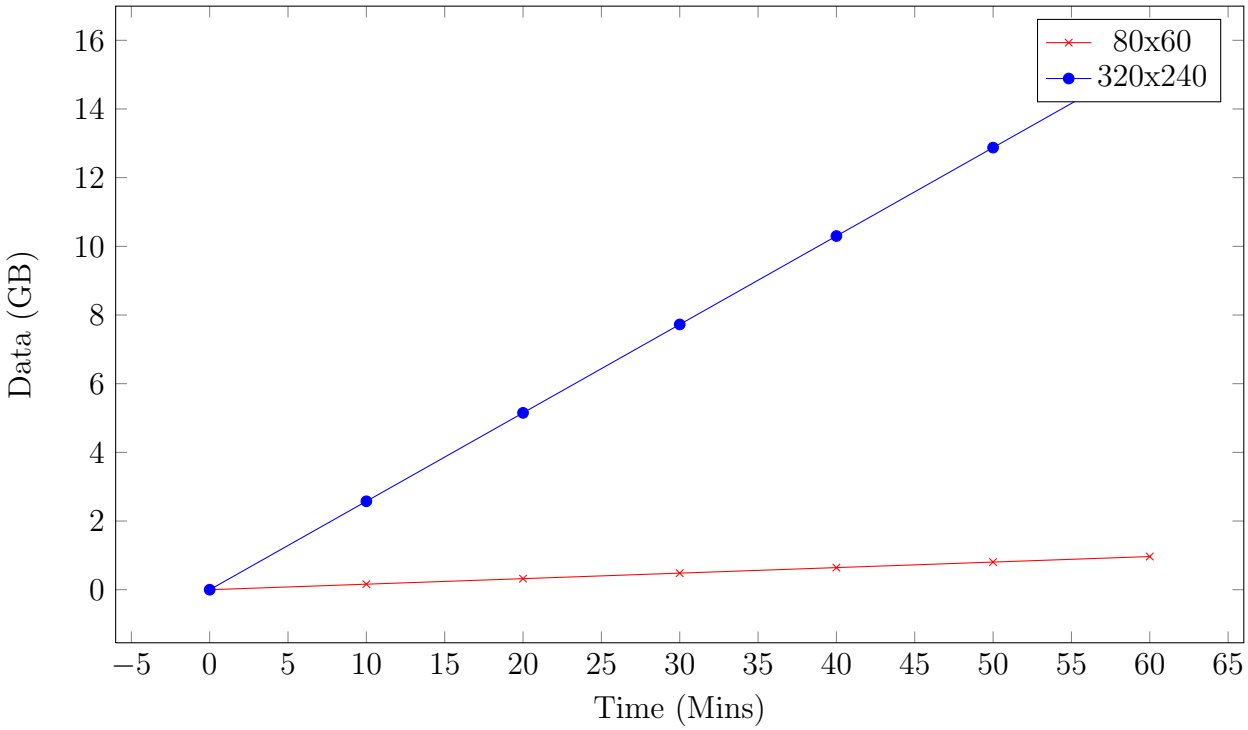

Figure 2: Increasing the depth resolution from 80 by 60 to 320 by 240 on a depth only recording results in an almost 16 fold increase in size from just under 1GB to over 15GB respectively for 60 minutes recording.

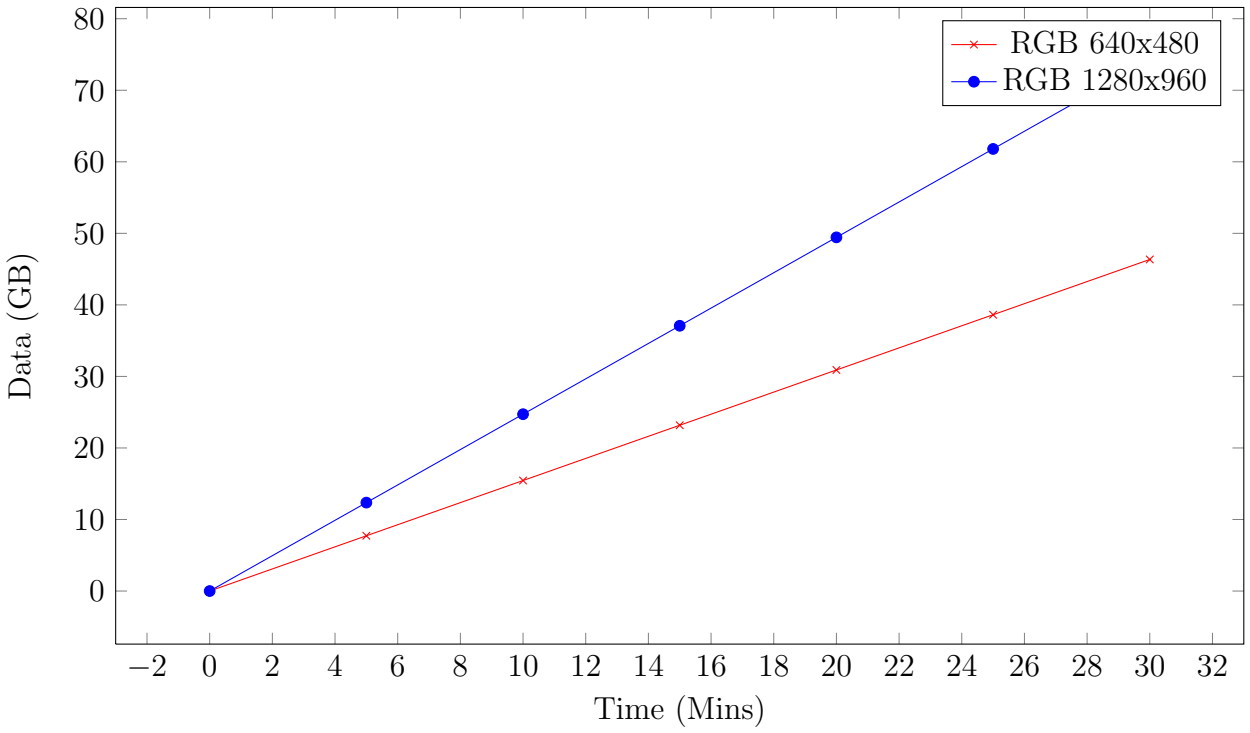

Figure 3: Increasing data sizes from 640 by 480 RGB to 1280 by 960 RGB.

have higher quality RGB capture at 12Hz for relatively the same size for lower quality RGB but at more than twice the frame rate. This tradeoff can be left to the user depending on their needs for the recording.

3.5 Compression

When we consider a depth field we should be able to envision that certain values in the data are similar and sometimes the same, which would lead to the idea that compression should work well on this data. Although no online compression was used in this dissertation it would indeed be considered as future work.

Offline compression on the other hand was run on the resulting recordings from the application, which quite good results. For example, a relatively small recording of only the depth stream at 80 by 60 reaching 9.5 MB was compressed with 7zip[6] using it's ultra compression setting down to 1.6 MB resulting in an almost 600% compression. A similar and even better result on a depth recording at 320 by 480 compressed from 115 MB to 12 MB.

[6] *www.7- zip.org*

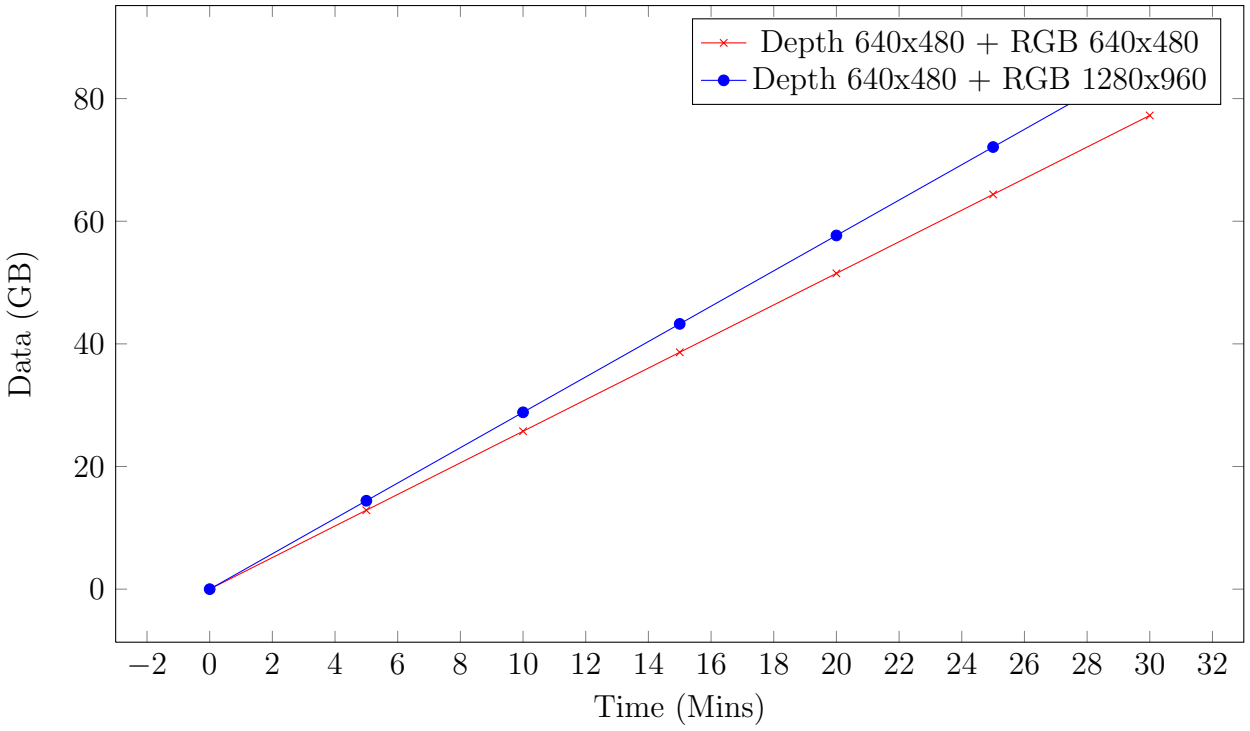

Figure 4: Relatively small data size increase from depth and RGB 640 by 480 recording to depth 640 x 480 and RGB 1280 by 960.

3.6 Kinect Wrapper

As part of the C++ code there is a wrapper around the Kinect and file reader which makes streaming from the Kinect and streaming from file transparent to the developer. This wrapper contains Methods to connect to a Kinect device, set the resolutions and start retrieving frames from the device.

The wrapper also contains a function to specify a file to read from, functions to retrieve the current resolutions and retrieve frames from the file in the exact same way as retrieving frames from the Kinect. This means that any loop which retrieves frames does not have to be changed when writing code for changing the stream source.

The function to retrieve a frame from the stream source contains two output variables rather than a returned value. The values are a pointer array of unsigned shorts and a pointer array of unsigned chars for the depth and RGB frames respectively. These memory for these arrays are allocated in the heap when the resolution is changed in the wrapper, so the developer should not delete or free these arrays.

To change the depth or RGB resolution two separate functions are defined, both

taking the Kinect SDK's NUI API enumeration for resolutions. The resolution code is checked to be valid not only in value but also makes sure the code is an option for the current stream before setting the appropriate resolutions and restarting the stream with this new resolution. Two sets of functions are defined for getting the resolutions of the depth and RGB stream; a set that returns the NUI API enumeration code and a set that modifies integer pointers for width and height.

Note that it is up to the developer to handle the representation of this data in terms of handling memory. For example, each element of the depth array is a separate pixel in the frame but the RGB frame pixels are made up of three elements of the array for red, green and blue.

3.7 MatLab Function

Along with the C++ wrapper provided there is a MatLab function to read recorded files only without the Kinect functionality. This wrapper provides access to the recorded data for offline processing with MatLabs powerful math functionality.

Chapter 4

Application Front-End

Shown in figure 5, when the application starts the user is presented with a window with two black viewports and a number of disabled buttons bar one. These buttons are organised into groups. From left to right the buttons are:

- Kinect Group - This group has buttons to connect to a device and enable/disable near mode.
- Record Group - Buttons to start and stop recording of Kinect stream. When no device is connected the buttons are disabled. When streaming from a file the buttons are disabled. When connected to a device but no output file is selected the buttons are enabled.
- Depth Recording Resolution Group - Buttons to set the resolution of the depth stream and enable/disable the depth stream. Much like the record group of buttons when no device is connected, when streaming a file or a device is connected but not output file is selected the buttons are disabled. However, when streaming from a file the buttons reflect the presence and resolution of the depth stream.
- RGB Recording Resolution Group - Buttons following the same pattern as the depth recording resolution group. Shows and selects the presence and resolution of the RGB stream.
- Camera Elevation Group - Buttons to change to tilt angle of the Kinect as the angle is changed by a motor in the device and physically changing the angle by hand may damage the device. These buttons are disabled when no device is connected or the application is streaming from at file.

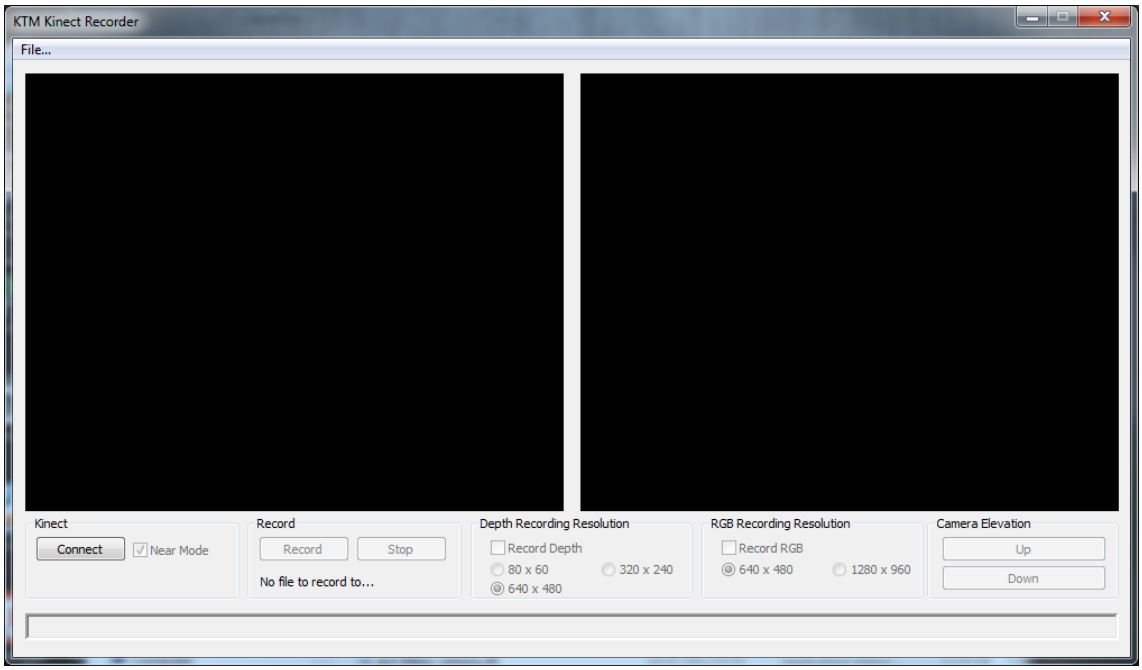

Figure 5: Application window at start up.

The two viewports in the application provide a preview of the depth stream and the RGB stream to be recorded on the left and right respectively.

The application front end is a Windows based form designed and implemented in Visual Studio 2010. When a button is pressed Windows passes a message to the program which then handles it in the appropriate manner. In the background a continuous update loop is run to fetch frames and pass it to the image viewer. These image viewers are Direct 3D viewports to which an updated bitmaps of the depth and RGB is drawn to.

In the case of the RGB been drawn to the viewport, it is a simple case of converting the data to a bitmap, which is handled by the system library. For the depth the scenario is a bit different as we need to show the data in a format compatible with bitmaps i.e using an RGB style where the maximum values are 255 each for the red, green and the blue channel.

A simple approach was chosen of dividing the depth pixel value by the maximum depth and multiplying by 255 yields a representation of the depth where white is closest and black is farthest with a 253 values in between. Values of zero are interpreted as black as the Kinect cloud not determine the depth for the pixel. This can be seen in figure 6 in the left viewport.

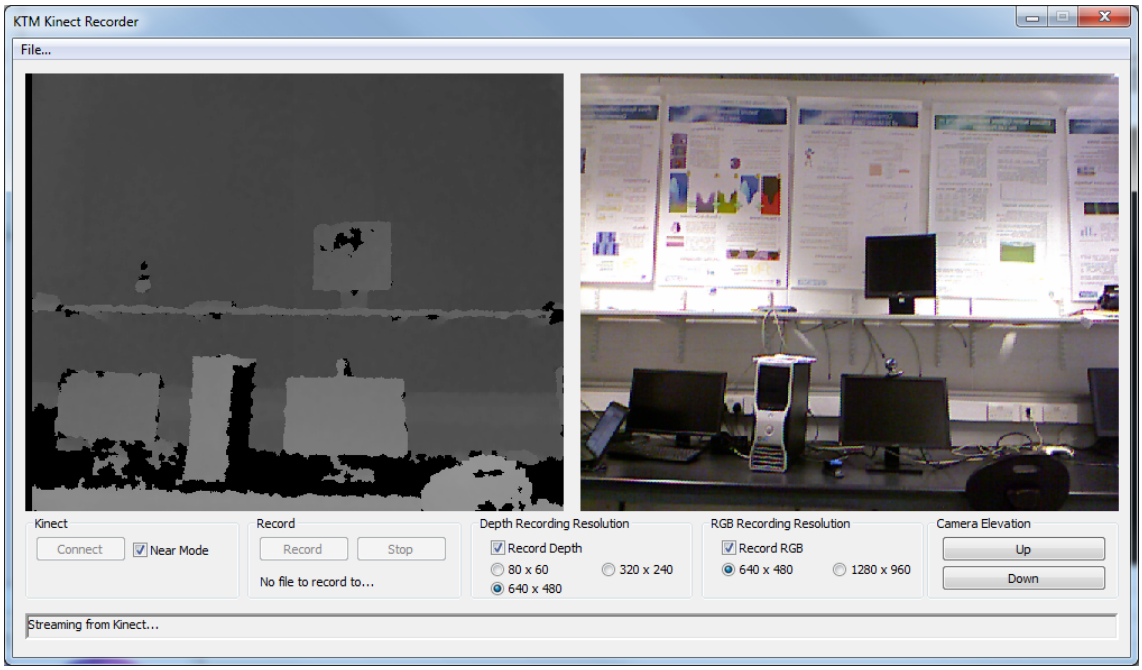

Figure 6: Application window connected to Kinect device streaming depth (left) and RGB (right).

A more complicated approach to this problem would be to use a heat map style image where the values range from red trough yellow to blue, red being the closest and blue being the farthest.

Chapter 5

Point Clouds

One of the most useful applications of depth cameras is the ability to extract point clouds and reconstruct a scene. Point clouds are surface-less 3D virtual scenes which can represent real world scenes.

If we use the Kinect to scan a room the sensor will only pick up a small portion due to its field of view, however if wanted a complete virtual scene if the room, multiple frames would have to be combined, rigidly transforming them into their correct respective places.

This chapter will cover the point cloud extraction from the depth view in section 5.1 and point cloud registration is discussed in section 5.2.

5.1 Depth Field Projection

Due the to fact that the depth sensor on the Kinect is effectively a pinhole camera the resulting image suffers from radial distortion. Imagine that the camera is faced at a flat wall at a perpendicular angle. The closest point on the wall is going to be the centre of the image but as the values gets farther away from the centre of the image, the distance increases.

Because of this effect if we were to directly transfer this data into a point cloud without any correction a flat wall would appear as a convex hull point cloud. In figure 7 a pinhole camera is faced at a perpendicular angle to a flat plane. The perpendicular distance $d_{height/2}$ is the shortest reading of the image with d_0 and d_{height}, the outer readings of the image, being the farthest. If these readings were used directly in creating a point cloud a convex shape would appear as shown in 8.

In order to compensate for this radial distortion two methods can be used; firstly,

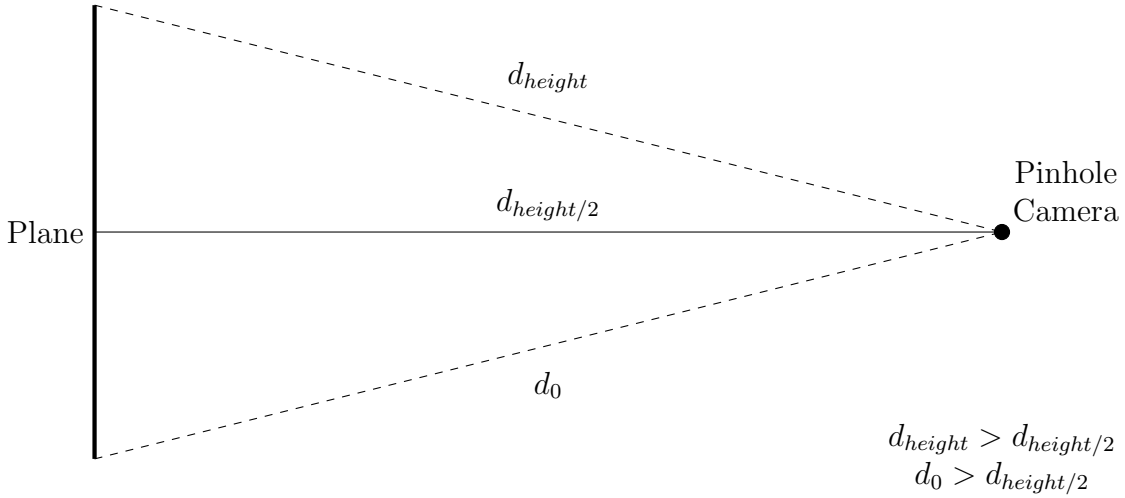

Figure 7: When measuring depth distances from a pinhole camera to a flat plane perpendicularly, the centre reading is the shortest with the depths increasing towards the edges of the image.

the complex solution is to calculate the intrinsic camera parameters. The intrinsic parameters compensate for the focal length of the camera and if an intrinsic calibration matrix is constructed from these parameters a simple matrix multiplication from the image plane to 3D camera space can be performed.

The second, much easier way is to use the Kinect SDK's depth to 3D space function. This function, given a u and v, the depth image coordinates, and a w, the measured depth, returns a 4 element vector with the x, y and z of the point, with the homogenous coordinate w representing a point. For simplicity and time constraints this was the method chosen.

In order to prevent repeated calls to this function, an array of vectors the dimensions of the depth stream was created from this function. For every u and v (width and height) in the depth stream resolution, use these u and v coordinates and a depth of 1 to get a vector. When perspective projecting from the depth image to a point cloud a lookup is performed for the current image pixel in the array of vectors and the vector is then multiplied by the depth value at that image coordinate, projecting the value into 3D space.

Because this perspective transformation would have to be applied to every point in the depth image, threading the process was an important step. The frame was divided up into 8 parts and each part taken by a thread to transform.

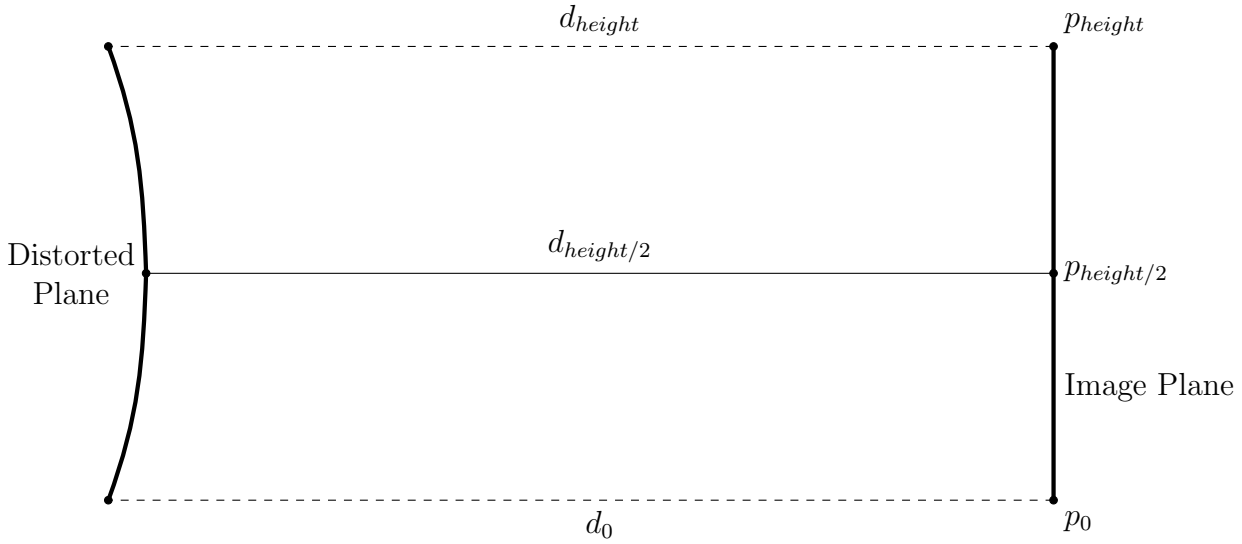

Figure 8: When reprojecting data from a pinhole camera care must be taken in making sure radial distortion is compensated for.

Using the standard library method for getting the current program ticks[1] before the process and after the process. The process was tested un-threaded and threaded, with the un-threaded version taking on average 150 ticks for 320 by 240 frame and the threaded version taking on average 50 ticks for the same frame.

5.2 Point Cloud Registration

After the point cloud is extracted from the depth image the next goal is to merge point clouds from different frames into one large scene. Further work could permit the extraction of 3D meshes from the scene. In order to reconstruct the scene, the transformation from one frame's point cloud to the next needs to be found.

The method I chose is outlined in the Kinect Fusion research[9]. This algorithm is based on iterative closest point by finding corresponding points and finding a rigid transform that minimizes an error function.

In terms of terminology a global transform is matrix which transforms a point or a point cloud into world space i.e the virtual reconstruction of the scene. This transform is a rigid transformation, meaning it only rotates and translates but does not scale. A point or a point cloud in camera space is a point or cloud relative to

[1]The standard library method for ticks was used instead of using system time as ticks represent the ticks of the application rather than absolute system time. If other applications are running using the system time is an unfair and unreliable method of timing code.

the camera. This point in camera space is the same as the point after perspective projection. A global point is a point in world space.

Starting off the algorithm, an estimated global transform is assigned. This estimated transform is set to the previous point cloud's transform. To find point correspondences the algorithm transforms the previous point cloud by it's global transformation inverse matrix. This brings the point cloud back to camera space. For each point in this point cloud the inverse of perspective projection is applied to find it's original point in the depth image. If this point in the current frame is greater than zero and if the point in the current point cloud corresponding to that point in the depth image is within certain distance threshold from the previous point cloud's *global* point and the dot product of the normals are within a threshold then we have found a point correspondence.

These correspondences are then used to determine an error cost, which is defined as the sum of the square of their distances. If the cost is below a certain threshold *or* the difference between the previous iteration's cost and the current iteration's cost is below a threshold, the estimated transformation can be considered the global transformation of the current point cloud.

On the other hand, if the cost is above the threshold a covariance matrix is derived from the corresponding points and then this matrix is decomposed with either Cholesky decomposition as in Kinect Fusion, or Singular Value Decomposition as in the current implementation.

Chapter 6

Future Work & Discussion

Due to time constraints all that has been outlined is all that has been completed. Having had more time many more goals and objectives would have been set and hopefully completed. Further to this, if the chance was given there would certainly be parts of the dissertation that would be changed, or that could be refactored in the future.

On top of this, new hardware with better specifications than current technologies could provide better quality and potentially more applications. For example if the stage is reached where a device like the Kinect with the same dimensions can deliver results as accurate as current laser scanners but in real time the historical archiving of statues could be a much shorter process for the same current quality as done with a laser scanner[12].

Microsoft's XBox reveal not only announced the new XBox, but also the new Kinect to accompany the console[1]. The device has a 1080p RGB camera and a microphone array (much the same as the current Kinect). These would be considered the next steps up from the current Kinect setup. The big difference however is the complete change of the depth sensor. Instead of going for a higher resolution structured infrared sensor they have opted for a time-of-flight sensor instead. The accuracy and noise of this sensor will be interesting to measure as time-of-flight depth sensors can be very noisy making feature extraction very difficult[16].

Keeping to the current dissertation topic, the point cloud section cloud absolutely be extended to perhaps extract 3D surface models from a scene. This could further be extended by using complex algorithms to get fine grain detail from the model

[1] Although no official link to the specifications of these devices has been established there are plenty of videos of the official reveal which go over the vague specifications

comparable to laser scans.

The tool itself can be used regularly for situations where processing power isn't in abundance or the actual application isn't complete but the time frame for recording is deadlines so recording the depth stream and color stream to be processed offline is a viable option. For example if a library has busts of influential figures but the library can only permit recording for a few hours on a specific day and the application is far from complete at that stage, using the recording tool to retain the data is a useful compromise.

In retrospect instead of having a varying size header for the file a fixed size header would have been more appropriate to prevent the slight complexity in determining the stream types in a file and where the data starts in the file.

Chapter 7

Conclusion

This dissertation presents a tool to aid researchers who are using or plan to use to Kinect to record the data streams separately from their goal application. In cases where the application they plan to use Kinect streams for is not complete yet or it is not practical in terms of processing power to do online computation, this tool provides a method of taking data and working on it offline.

The tool is stand-alone, stable, reliable, without dropping frames. It is flexible in what streams and at what resolutions are recorded. The file structure used is open and non-complex. Due to the nature of the data it is easy compressed post recording to very manageable sizes. The tool is open source and is hosted on GitHub with documentation of the file structure.

This report provides insight into what to expect in terms of data size from recordings at length. This hopefully will allow researchers to choose their resolution and streams carefully depending on their needs.

Bibliography

[1] Kinect SDK - Resolution Selection. http://msdn.microsoft.com/en-us/library/microsoft.kinect.colorimageformat.aspx. Accessed: 2013-02-11.

[2] Gérard Blais and Martin D. Levine. Registering multiview range data to create 3d computer objects. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 17(8):820–824, 1995.

[3] Gérard Blais and Martin D. Levine. Registering multiview range data to create 3d computer objects. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 17(8):820–824, 1995.

[4] Yang Chen and Gérard Medioni. Object modeling by registration of multiple range images. In *Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on*, pages 2724–2729. IEEE, 1991.

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

[6] Brian Curless. From range scans to 3d models. *ACM SIGGRAPH Computer Graphics*, 33(4):38–41, 1999.

[7] Philipp Fechteler, Peter Eisert, and Jürgen Rurainsky. Fast and high resolution 3d face scanning. In *Image Processing, 2007. ICIP 2007. IEEE International Conference on*, volume 3, pages III–81. IEEE, 2007.

[8] Martial Hebert. Active and passive range sensing for robotics. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 1, pages 102–110. IEEE, 2000.

[9] Shahram Izadi, David Kim, Otmar Hilliges, David Molyneaux, Richard Newcombe, Pushmeet Kohli, Jamie Shotton, Steve Hodges, Dustin Freeman, Andrew Davison, et al. Kinectfusion: real-time 3d reconstruction and interaction using a moving depth camera. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 559–568. ACM, 2011.

[10] Bing Jian and Baba C Vemuri. Robust point set registration using gaussian mixture models. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 33(8):1633–1645, 2011.

[11] Kourosh Khoshelham. Accuracy analysis of kinect depth data. In *ISPRS workshop laser scanning*, volume 38, page 1, 2011.

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

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

[14] Szymon Rusinkiewicz and Marc Levoy. Efficient variants of the icp algorithm. In *3-D Digital Imaging and Modeling, 2001. Proceedings. Third International Conference on*, pages 145–152. IEEE, 2001.

[15] Jonathan Ruttle, Claudia Arellano, and Rozenn Dahyot. Extrinsic camera parameters estimation for shape-from-depths. In *Signal Processing Conference (EUSIPCO), 2012 Proceedings of the 20th European*, pages 1985–1989. IEEE, 2012.

[16] Sebastian Schuon, Christian Theobalt, James Davis, and Sebastian Thrun. High-quality scanning using time-of-flight depth superresolution. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on*, pages 1–7. IEEE, 2008.

[17] Daniel Wigdor and Dennis Wixon. *Brave NUI world: designing natural user interfaces for touch and gesture*. Morgan Kaufmann, 2011.

[18] Wei Xin and Jiexin Pu. An improved icp algorithm for point cloud registration. In *Computational and Information Sciences (ICCIS), 2010 International Conference on*, pages 565–568. IEEE, 2010.

[19] Song Zhang and Peisen Huang. High-resolution, real-time 3d shape acquisition. In *Computer Vision and Pattern Recognition Workshop, 2004. CVPRW'04. Conference on*, pages 28–28. IEEE, 2004.