5-2012

Three-Dimensional Scene Reconstruction Using Multiple Microsoft Kinects

Matt Miller
University of Arkansas, Fayetteville

THREE-DIMENSIONAL SCENE RECONSTRUCTION USING MULTIPLE MICROSOFT KINECTS

THREE-DIMENSIONAL SCENE RECONSTRUCTION USING MULTIPLE MICROSOFT KINECTS

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science

By

Matthew Thomas Miller
University of Arkansas
Bachelor of Science in Computer Science, 2009

May 2012
University of Arkansas

ABSTRACT

The Microsoft Kinect represents a leap forward in the form of cheap, consumer friendly, depth sensing cameras. Through the use of the depth information as well as the accompanying RGB camera image, it becomes possible to represent the scene, what the camera sees, as a three-dimensional geometric model. In this thesis, we explore how to obtain useful data from the Kinect, and how to use it for the creation of a three-dimensional geometric model of the scene. We develop and test multiple ways of improving the depth information received from the Kinect, in order to create smoother three-dimensional models. We use OpenGL to create a polygonal model combining the RGB camera image and depth values. Finally we explore the possibility of combining the three-dimensional models from two Kinects to create a better representation of the scene.

This thesis is approved for recommendation
to the Graduate Council.

Thesis Director:

_____

Dr. John Gauch

Thesis Committee:

_____

Dr. Craig Thompson

_____

Dr. David Andrews

THESIS DUPLICATION RELEASE

I hereby authorize the University of Arkansas Libraries to duplicate this thesis when needed for research and/or scholarship.

Agreed _____
Matthew Thomas Miller

Refused _____
Matthew Thomas Miller

ACKNOWLEDGEMENTS

I would like to thank Dr. John Gauch, for guidance and unfaltering optimism when it came to getting my thesis finished. Without his support my time as a graduate student and masters thesis would have surely spiraled into an abyss.

I would also like to thank, Dr. Craig Thompson, and Dr. David Andrews, for serving on my committee.

To my parents, well, life and school certainly wouldn't have been possible. You shaped my life, and did the best you could with what you had to work with, me.

Finally, my deepest thanks, and love go to my dear fiancé, Minghua, who was the one who encouraged me to enter graduate school in the first place, and the one who dragged me to the finish line.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

Scene reconstruction is an area of research in computer vision that uses a visual form of input, generally RGB cameras, and sometimes multiple inputs, to reconstruct the scene, or what the cameras see, as a three-dimensional geometric model. Normally at least two cameras are used, a set distance from each other, in order to recreate stereo vision [4]. This is analogous to how the brain works when it combines images from its two eyes to understand a scene.

When the Microsoft Kinect [5] was introduced as an inexpensive depth sensing camera, it caught the attention of researchers as a great tool for computer vision applications. The Kinect makes use of structured light to obtain depth information [2]. It works by sending out a pattern of infrared light. This light then proceeds to hit objects, once it has done so it will generally bounce back. The Kinect then uses its infrared sensor, to determine at that specific point in the pattern, there was an object a certain distance away. This is different from another method of estimating distance, where you try and detect the same object in multiple images, and use a form of triangulation to estimate the depth. Kinect depth sensors, along with a coupled RGB camera a fixed distance away allow you to determine the depth of a specific pixel in the RGB image.

In our thesis we decided to try to use a Kinect on a computer to collect the depth and RGB data. Once we had the data, we needed to analyze this data, to determine if it could be used for scene reconstruction. Although most data is good, we found several problems with the collection of depth information. Structured light is does not work on certain surfaces, and may not return the same results twice in a row. Also between two or more Kinects there can be interference that causes even more errors in the data. We implemented and tested various methods that attempt to reduce the amount of errors in the depth image; such as time based

average, as well as multiple strategies at pixel based averaging. What we found is that while these methods work well when viewing only the depth information, when combined to make a three-dimensional model, they are not always better than the original.

In order to create the three-dimensional model, we decided to draw a collection of polygons, one for each pixel in the RGB input image. For the Z values of polygon vertices we used the depth values at the same location as the RGB pixel. This turned out quite well, and gave us a smooth model based on the depths. A few problems we had were camera calibration, which we attempt to fix through manual alignment, and jittery polygons. These are polygons that move back and forth over time and appear to be caused by our attempted fixes on the depth errors.

Our next step is to attempt to combine the models of two Kinects by manual translation and rotation. We let OpenGL handle polygon collisions, which works in certain restricted situations.

The rest of this report is organized as follows: first we will provide background about the hardware and software as well as some similar works using the Kinect. Next we describe our approach and our development path: what we did, and the results of each different method. Finally we provide a conclusion, and future work.

2. BACKGROUND AND RELATED WORK

In this section we examine some of the hardware, and projects that make working with the Kinect possible. We also look at some Kinect projects that have been developed since the Kinect was first "hacked." In section 2.1 we describe the Kinect itself, and some of its capabilities. In section 2.2, we will talk about the company PrimeSense and its Kinect software, NITE, and OpenNI. In section 2.3 we describe OpenKinect, and LibFreenect. Finally in section 2.4 we review a similar Kinect project that also does scene reconstruction.

2.1 Microsoft Kinect

The Kinect is an addition to the Xbox 360 gaming console, that allows complete controller free motion control. It is said to be able to track up to six people, although this is only limited by the Kinect's field of view [1].

The Kinect has an 8 bit VGA RGB camera with resolution 640 x 480 with a Bayer color filter, 50% green, 25% red, 25% blue, RGBG. Also it has an 11bit 640x480 (632x480 usable, last 8 columns are always no data) depth sensor, that allows for up to 2048 levels of sensitivity. The depth sensor consists of an Infrared Projector and CMOS sensor. There is a filter on the sensor that is supposed to filter out ambient light, allowing it to operate in any condition.

The software allows the Kinect to track the skeleton, and joints, of each player, allowing for complex gesture recognition and control. Multiple gestures can be mapped to actions, as well as the possibility to create new gestures on the fly.

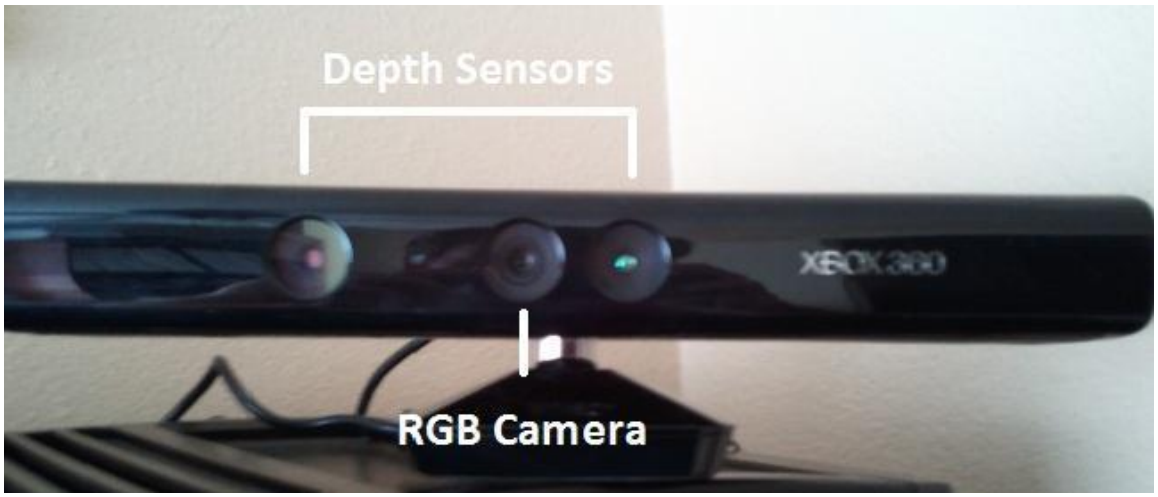

Figure 1: Shows the Microsoft Kinect with RGB camera and Depth Sensors labeled.

2.2 PrimeSense, NITE, and OpenNI

Kinect technology is based on PrimeSense's range camera technology [1], which interprets 3D scenes from continuous infrared structured light. This system called Light Coding uses a technique variant of image-based 3D reconstruction.

PrimeSense released open-source source drivers and motion tracking software called NITE [1]. NITE is the software Microsoft used as a reference for the tracking of features and gestures in the Kinect. It allows for what is called Natural Interaction. Combined with PrimeSense's sensor technology, it makes up the bulk of the Kinect.

PrimeSense also launched OpenNI (Open Natural Interaction) [1] an open source project that is designed to build natural interaction into devices that support it, as in those with PrimeSense like technology, RGB-D cameras.

2.3 OpenKinect and LibFreenect

OpenKinect is an open source project that allows for use of the Microsoft Kinect on a PC [2]. The main project libfreenect is a library that provides the ability to use all the Kinect's abilities. It allows for retrieval of the RGB camera data, depth data, as well as the ability to control the devices motor and LEDs. Libfreenect is an asynchronous C interface to the Kinect, however wrappers for other languages such as C++, C#, Java, Python Ruby are available.

Our thesis depends upon the libfreenect library to easily retrieve the data from the Kinects. Much of the work draws inspiration from the C++ wrapper, and device examples that come along with this library.

2.4 Oliver Kreylos Kinect Hacking

Oliver Kreylos, a UC Davis visualization researcher, released one of the first big hacks for the Kinect in the form of a YouTube video [3]. This was a project that, similar to our thesis, combined the depth and RGB information from the Kinect to create a three-dimensional recreation of the scene. The goal of his project was to have real objects rendered along with digitally created objects, allowing for a kind of augmented reality. This sparked huge attention in the Kinect from a wide range of researchers, and showed that it was possible to do this sort of work for Kinect.

Figure 2: Oliver Kreylos' Kinect Hack, 3D scene reconstruction.
http://www.youtube.com/watch?v=KW9of1Ud0uo

3. APPROACH

Overall our approach is relatively straightforward. First, we use LibFreenect from the OpenKinect project to get the image and depth data we need. Then, we analyze the data and apply various methods to attempt to minimize errors. Next, we combine the depth and RGB images to produce a 3D geometric model using OpenGL. Finally, we combine two 3D models, to attempt to create a better reconstruction of the scene.

3.1 Setting up and Using Kinect

The first thing to do was to get a single Kinect up and running on a Linux PC. This involved downloading and installed a driver that allows use of the Kinect over USB. Afterwards, we chose to use the OpenKinect software to access that Kinect. We chose to go this route rather than rewriting code to access it via USB. Code would have to be written using libusb to interact, and all the appropriate video callbacks, making it more work than it was worth. With OpenKinect all this is already built in, along with the wrappers for any language, including the one we needed which is C++. After it was installed to test we used a binary called cppview, which simultaneously shows the RGB stream, and depth stream using OpenGL.

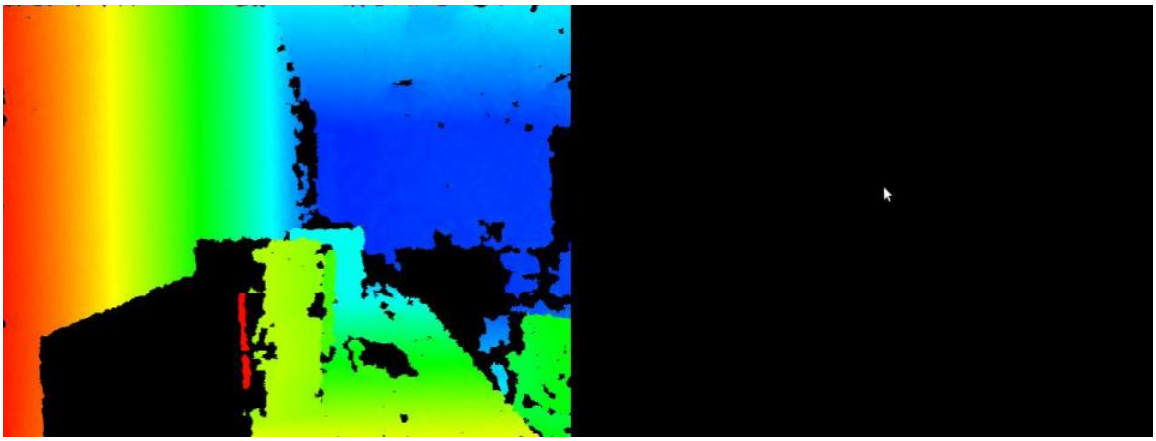

Figure 3: Shows the binary cppview that comes with libfreenect. The left shows the depth image, and the right shows the RGB image, which does work due to improper initialization in their code.

3.2 Integration of Kinect and IM libraries

In order to learn how to interact with the library and be able to make changes quickly, we used the source for cppview as a basis. This source code was very unruly. It included such things as a class called MyKinectDevice, which provided the video and depth callbacks, as well as an asynchronous way to obtain the latest image of each in the form of a vector. Cppview also contained a lot of code that got the OpenGL window up and running, as well as some initialization code.

The first step was to integrate Dr. Gauch's extensive digital image manipulation library with this code. His library provides many useful classes, that allow you to manipulate any kind of image, in any way you want, such as conversation to the frequency domain, filters, scaling, and others too numerous to name. Obviously having all this code already made would be a huge help in further direction of the thesis.

To do this we decided to make the changes in the MyKinectDevice class, so that all the other code could continue to operate without change. We started with the video and depth

callback methods. These accepted raw data, void*, as their video streams. For video the stream was interleaved as red, green, blue, red, green blue, and so on. Originally this was then stored into a vector in the class. Instead we wanted to store it into a class in Dr. Gauch's library called im_color. This is the class that holds color images. It is made up of three im_short classes which are simply arrays of shorts that hold a monochrome image, one for red, green, and blue. We replaced a vector with the im_color, and simply de-interleaved it to convert it to this type of storage.

The depth stream is monochrome 11-bit, but was being converted to color, and stored in a vector. Instead of doing this we decided to store it in an im_short, and leave the conversion to color as an option to be performed later, this cuts down on initial work done for each callback, and allows access to the raw depth.

The only thing left was to convert the private vector variables to their respective form, and change the getRGB, and getDepth, the way to retrieve the last image of the camera, to instead support these new formats.

Unfortunately some code outside of the class did need to be modified for cppview to continue to work. To display the images, OpenGL was used. They were getting the image, in the form of the vector and simply converting this to a 2D texture that was displayed on the window. The way the conversation to a 2D texture works however, is that you need an interleaved RGB image. To accommodate this required new function that convert back from an im_color or im_short image to a vector. This is one more conversion than the original version; however, this is the last step to be performed before display and should not have much of a computational effect, while allowing access to a vast library of image manipulation functions.

Finally, we added a function to convert the depth back to a colored depth. We also added

the ability to switch back and forth between the monochrome depth and the colored depth by pushing a key.

Figure 4: Shows the update cppview with working RGB image, and im_short depth on left.

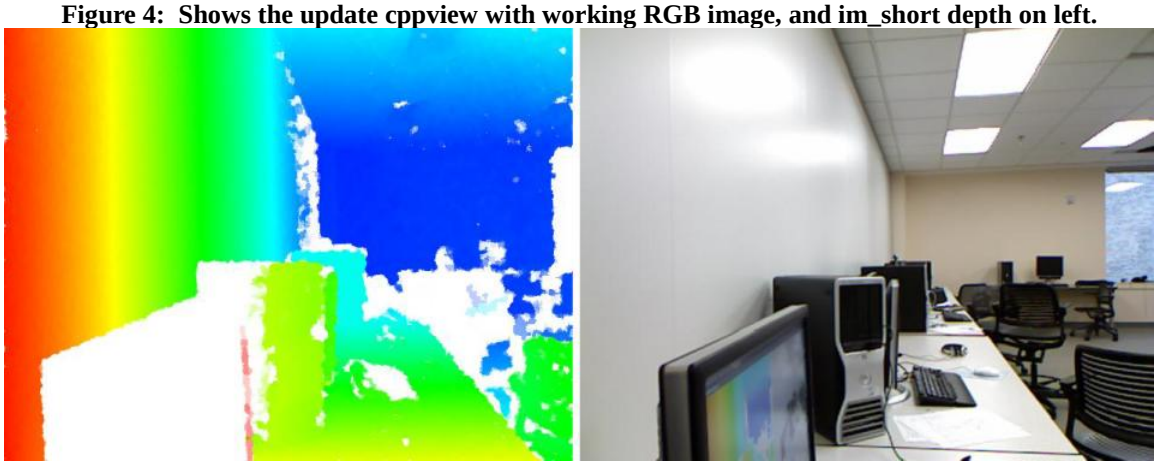

Figure 5: Shows the update cppview with depth converted to color on the left.

3.3 Creating a Cleaner Kinect Device Class

At this point there had been enough changes made to MyKinectDevice class to warrant making our own class. This would allow us to clean it up as well as implement some new features. The class was called Kinect. Our main goal was to clean up the code, allow for portability, use outside just cppview, and to hide some of the pains of use.

One of our main issues with creating an instance of the class with libfreenect is that you

also needed an instance of another class called libfreenect. This was troublesome as we just wanted to keep track of one single variable. Also the way to create an instance used the address to this libfreenect instance, had it call a create class and needed to use templates.

```
Libfreenect::Libfreenect freenect;
Kinect* dev=&libfreenect.createDevice<Kinect>(0);
```

This type of instantiation is unnecessarily complex, and we wanted to hide it not only from ourselves, but from anyone else that happened to ever use this class. To do this we decided to make that libfreenect object a static variable inside the class. The reason we think it is a good idea make it static is because it contains a mapping of all Kinect devices plugged into the machine, and what their index is. Through this, you give it an index to get back the specific device you need. If one of these was created for each instance of the Kinect class, the mapping would be incorrect due to it not having a complete listing of every Kinect, and you woudld not be able to access the Kinects properly. We then created a method called createDevice that takes the Kinect index as a parameter and simply returns a Kinect instance. This may be further simplified by pushing the code to the constructor as follows:

```
Kinect* dev=Kinect::createDevice(0);
Kinect* dev=new Kinect(0);
```

We also simplified the code in the callbacks, and get methods as much as possible. The previous constructor was also a problem. They were using a method of instantiating private variable that was inelegant and unnecessary for most of the variables. It did allow the use of its constructors however and was necessary for the freenect variable.

In early stages, due to the old constructor design, it would not work in a separate .h, and .cpp file design. With the new fixes it does, which makes it more portable. Another key

feature of the class is that it now allows for the use of two different Kinects on the same computer by specifying which one you want, and creating a separate object for each device. Using this we modified cppview to take in a command line argument specifying which Kinect device you wanted to connect to.

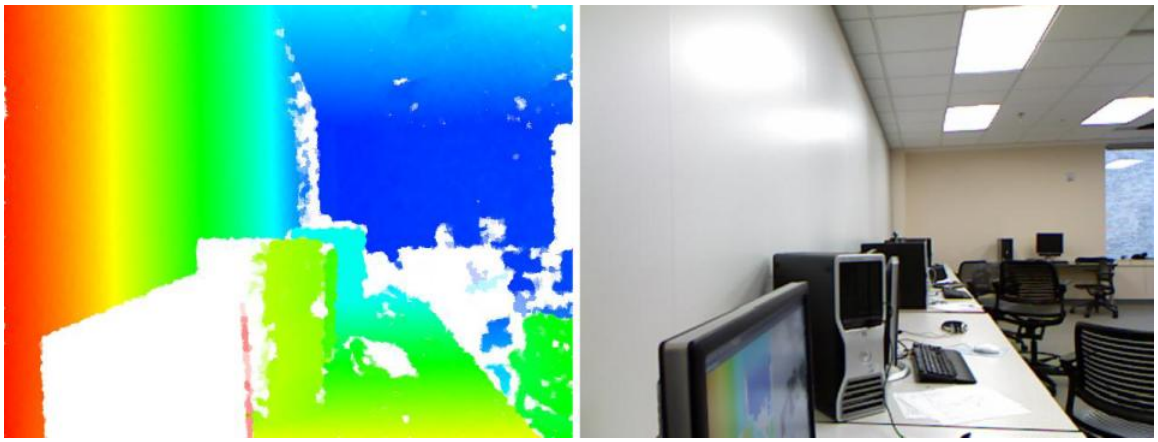

Figure 6: Shows cppview using the Kinect at index 0.

Figure 7: Shows cppview using the Kinect at index 1, and my smiling face.

3.4 Two Kinects

Creating the Kinect class earlier allowed the use of multiple Kinects per computer. Each Kinect is specified by an integer index starting with 0. The Kinect class takes this index, looks up a mapping it keeps of USB devices, and returns the specified devices. Using this we can open two instances of cppview, one with index 0, and one with index 1, to get two different OpenGL

windows showing the two different viewpoints of each Kinect.

We wanted however to be able to open multiple Kinects per window, to allow for ease of use, and to easily see the differences between different effects applied to each Kinect, and any effects of both of them working at the same time.

To accomplish this we made a struct, which could and should be made into a class in future work, that allows an easy way to handle all the different data items needed to display one Kinect device. Using this we could create two of these structs, one for each device, and display them. This required expanding the window, creating more 2D textures, and generalizing functions. We implemented a function that would place the 2D texture for each device at a certain point on the window based on its device number. Right now the number of Kinect devices is hard coded to be two, but in future this could be changed so you can have any number of Kinects devices per window.

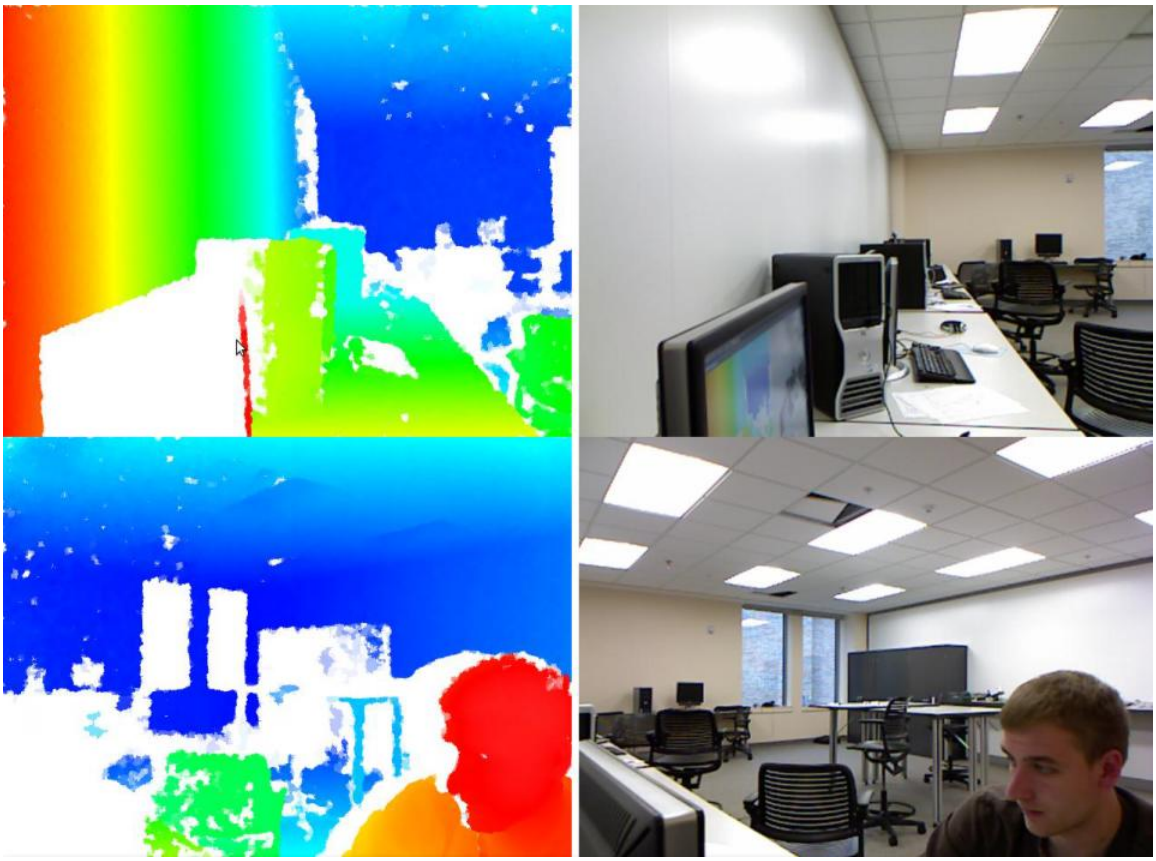

Figure 8: Shows the use of two Kinects together

3.5 Averaging Images

3.5.1 Reasoning Behind Averaging and Holes

Because the Kinect is a relatively inexpensive depth sensor, there is often noise or other artifacts in the depth data it produces. These problems can often be corrected using spatial or temporal image averaging. For example, for stationary objects like walls or tables, the depth values returned by the Kinect may vary slightly from image to image. By averaging values from successive images together, we get a more accurate depth model of these objects.

Another reason to do averaging is to remove black or white "holes" that occur in the depth data due to how infrared light interacts with objects in the scene. Holes can be caused

from a surface absorbing the light, or reflecting light away from the sensor. They can also occur because of sensor error or ambient light interference. As long as the infrared light sent out does not get back, a hole will be created. Looking at a sequence of images, holes tend to jump around a bit. Averaging will help in alleviating this hole problem, by keeping more of the old data, that is potentially good, and not keeping as much as the new potentially bad data. Of course this is not a complete solution and other steps need to be taken.

3.5.2 Averaging First Attempt

Our first attempt at temporal averaging is probably the most obvious, but slowest way to do it. We implemented a class called AveQueue, using an array-based queue of the last N images. Each time it gets a new image; the oldest image is removed from the queue and subtracted from the temporal average. Then the new image added to the queue and the temporal average. This turned out to be complex, programmatically and computationally, due to having to store and keep track of multiple images in an array. The more images you kept in your temporal average, the more "ghost" images were produced of moving objects.

3.5.3 Infinite Impulse Averaging

There is a much better way to do this, which is called infinite impulse response [7]. Thinking about this, it is really quite brilliant, and is much faster computationally and easier to implement. Instead of keeping track of the previous N images, all we need to keep is an average of the previous images. When a new image appears, it is given a certain weight W and combined with the previous average, which is weighted with (1-W) to create a new temporal average. By adjusting the weight W, we can control how much of the new or old image you want. Increasing

W reduces the amount of averaging that is applied, and decreasing W increases the amount of averaging. This is illustrated in the figures below. This approach never removes data; although the older the data is the smaller its effect on the average has and eventually gets so small it has almost no effect.

This not only works faster, but also allows an easy and dynamic way to control the image. We implemented in cppview a key press that allows you to change the weight W given to the new image, which of course alters the weight (1-W) applied to the average. In this way you can dynamically alter the video and depth streams. Implementing this averaging technique paves the way to further work in combining the video and depth images.

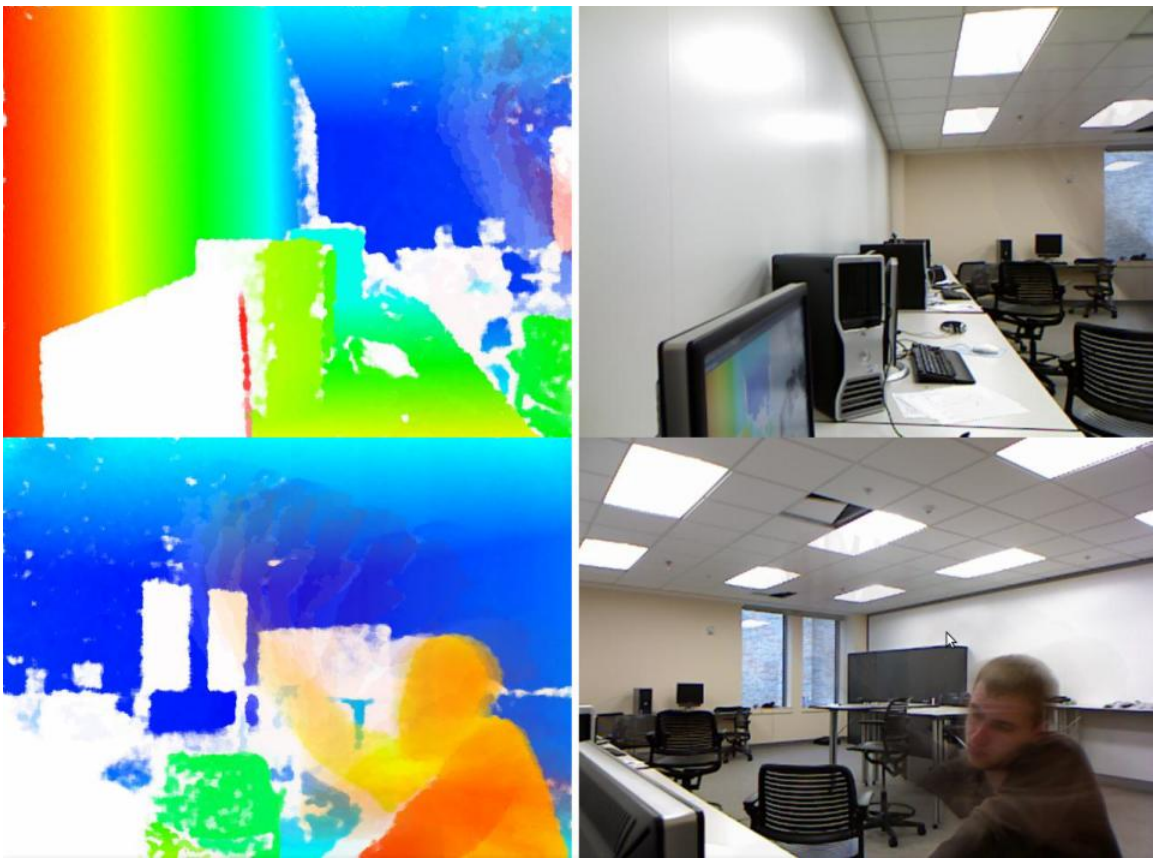

Figure 9: Infinite impulse response averaging with 10% of the new image and 90% of the average.

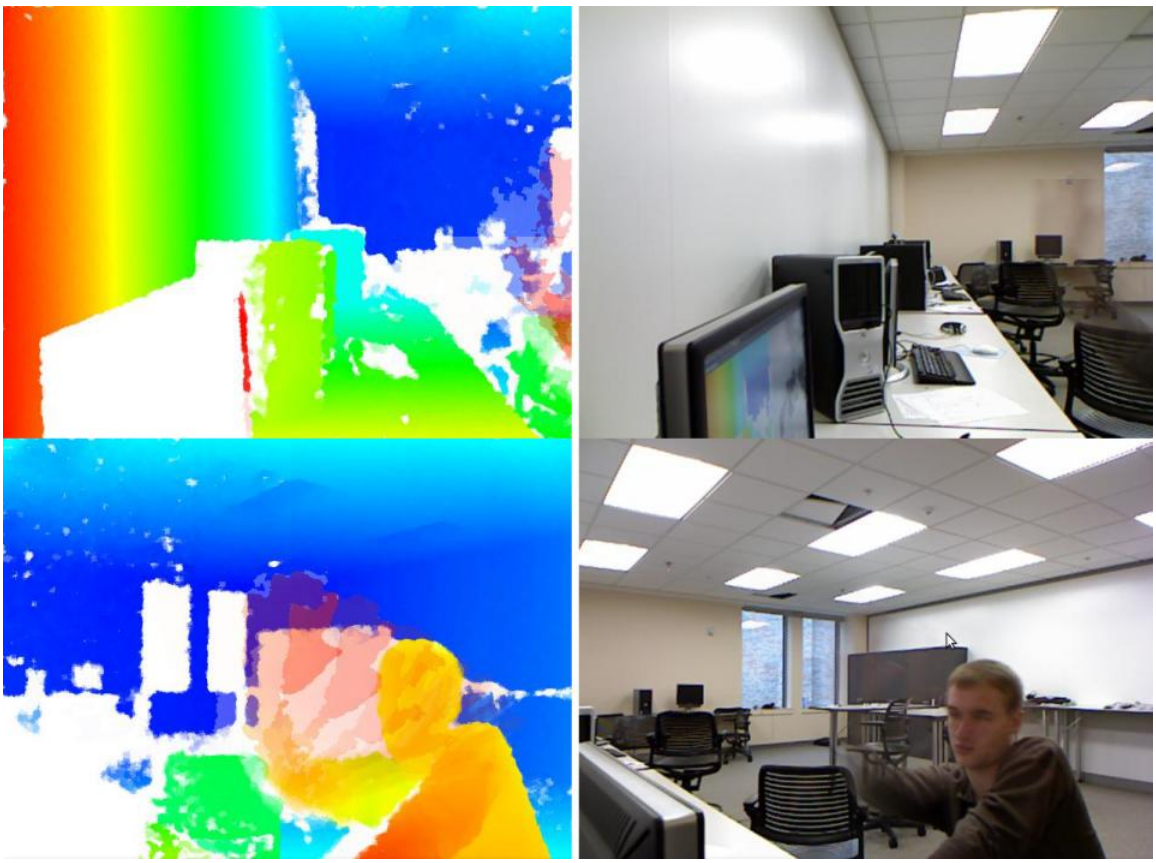

Figure 10: Infinite impulse response averaging with 30% of the new image and 70% of the average.

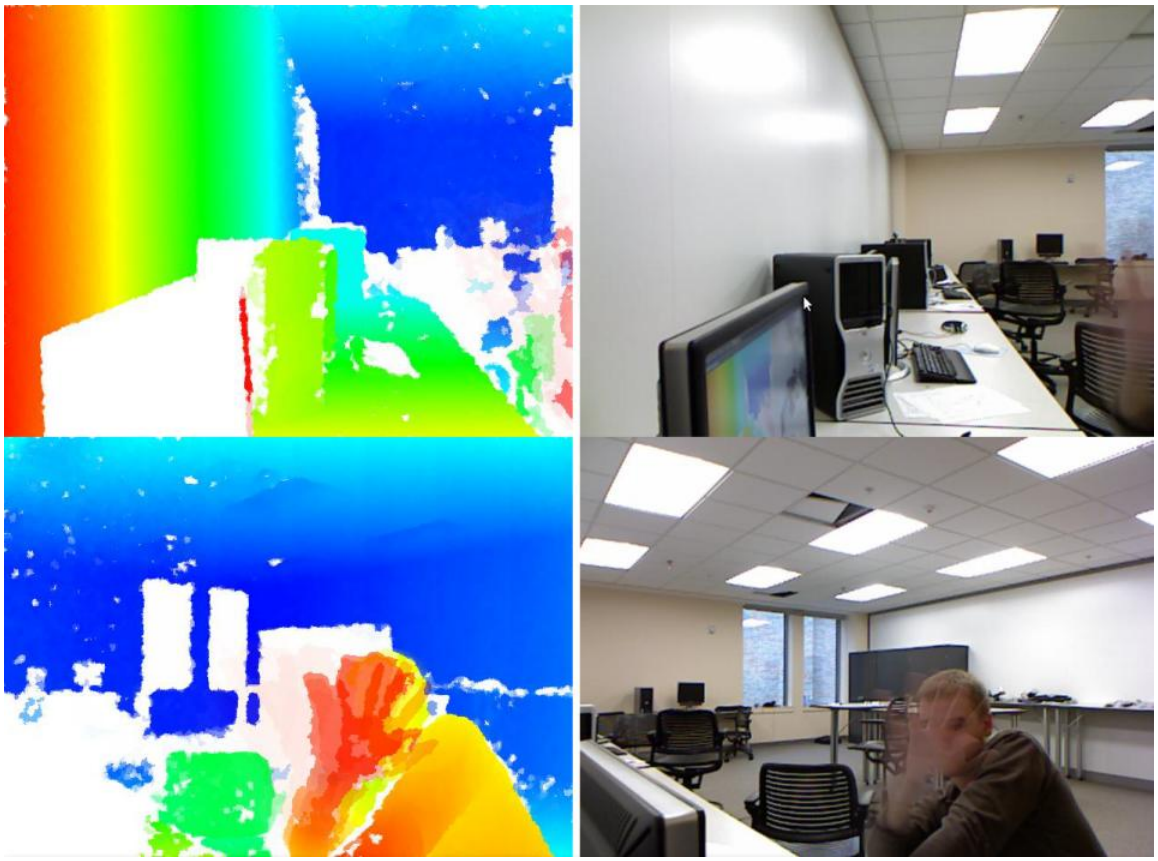

Figure 11: Infinite impulse response averaging with 50% of the new image and 50% of the average.

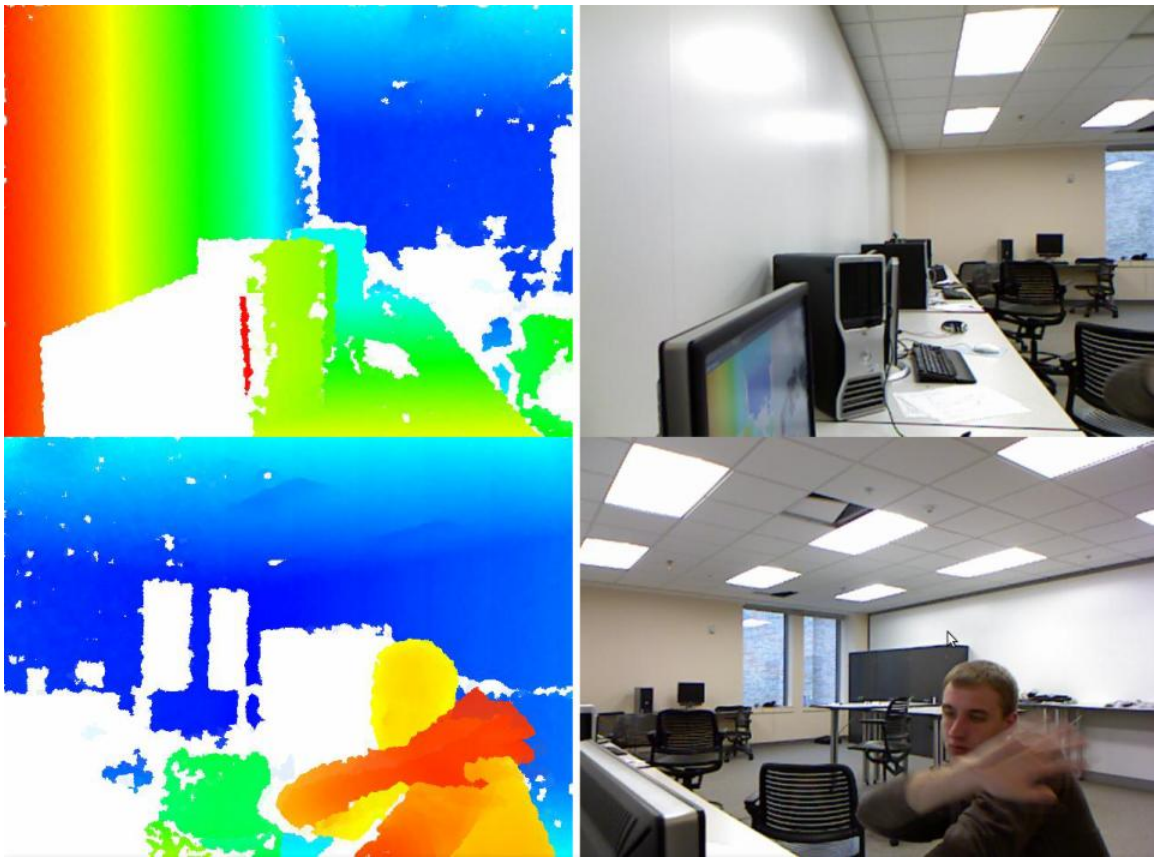

Figure 12: Infinite impulse response averaging with 70% of the new image and 30% of the average.

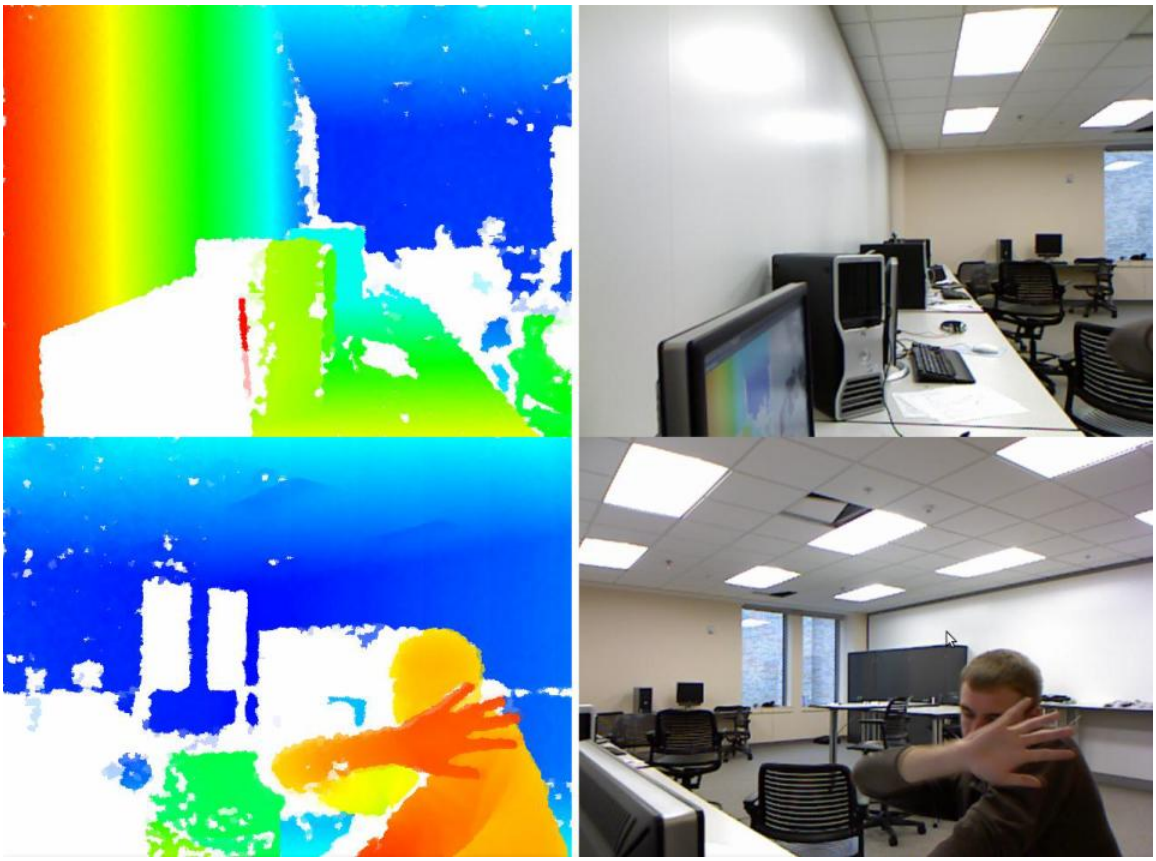

Figure 13: Infinite impulse response averaging with 90% of the new image and 10% of the average.

3.6 Holes

3.6.1 IR Interference from Multiple Kinects

Seeing two Kinect depth images at the same time allowed us to see clearly how they affected each other. Since both Kinects are on, sending out infrared light, with this light bouncing back all over the place, and each device's sensors taking this light in and giving a depth feed, this causes interference problems. Having both on at the same time creates a lot more holes in the depth image. Each of the two devices depth streams gets noticeably worse with the other on sending out infrared to the same area.

While this is a problem, we do not believe it to be a significant problem, because where

one Kinect may not have data because of interference, the other Kinect will have that missing data. At least that is what we observed. So we believe if both depth streams were to be combined, that a complete set of data would be available. Any data loss that occurs should be able to be overcome by any of the other data we obtain from the Kinects.

3.6.2 Ignoring Bad Data

It became apparent, that to get the best results going forward some sort of scheme for filling holes would be needed. Most of the holes were random errors, for some reason at that frame the infrared light did not make it back, or it could not recognize the data from the infrared pattern it uses. There was no reason that if we once had good data there, to replace it with bad data.

First we needed to determine which value represented no data. To do this we added a feature to the OpenGL window, where no matter where you click, it will print out the current RGB values of that particular location. Once we had this tool, all we had to do was click on the holes as they appear to figure out that the value of 2047, depth is an 11 bit value, represents no data in the depth image.

At first we decided to just not use the data from the Kinect if it was 2047. This worked, and was fast, however, it filled it all spots that did not have data, rather than just the holes that would appear and disappear. It was decided that the areas of no data that are not always there is what we wanted to focus on rather than the area that always are empty, polarized windows for example. To correct the problem of holes we added two more methods, one called streak, and another called previous averaging.

3.6.3 Streak

These two functions were incredibly fast due to them being able to be implemented as you get the data from the camera, which resulted in almost no more work. To implement streak, whenever you came upon a pixel with NODATA, you just take the valid pixel to the left of it and streak it across the hole, or use it to fill in all subsequent pixels in a horizontal line. While this is fast, and easy to implement it left visual streaks in the image which were undesirable.

3.6.4 Previous Average

Previous averaging is similar to streak in that it is very fast, however it also displays better results. Instead of just taking the previous good value, you take the pixel above it, to the left of it if they are not equal to NODATA, and average them for your new pixel value. This files in the holes with much better quality. However, a problem, this and streak both have is that not only do they fill in holes; they also fill in those large areas of NODATA that never change, which is undesirable. To tackle this, it became apparent that some form of a size requirement for the hole was needed.

3.6.5 Modified Flood Fill

The first method to fill holes with only a certain size we implemented was a modified form of flood fill, which is a classic region filling technique [6]. To do this we used a queue, and a linked list. Whenever there was NODATA, it was added to the queue. Then until the queue was finished, it would look at each one, and with two different schemes determine if its neighbors needed to be added to the queue. The easiest, was just to examine the pixels four neighbors, and if they were no data, add them to the queue. However, this turned out to pretty

slow, about 0.057s per frame.

A slightly faster method was to loop to the left and right of that pixel, adding each to the queue if they were no data, and stopping when a valid pixel is found. This is slightly faster at 0.033s per frame. To do this we had to modify this flood fill method, to keep track of the valid pixels around it as well. Whenever it would stop when it found a valid pixel, it would add this to a total, and keep track of the count. This way when it was done it could calculate the average of all the pixels surrounding the hole. Also we needed to change each examined pixel to a different value, we chose -1, so it would not re-add them to the queue, and finally to keep track of them for later, we kept them in a linked list structure to easily loop through and change their values to the average.

Because it can easily keep track of the size of the linked list, which contains the number of pixels in the hole, it can easily determine if it is a big hole, or a small one, and whether or not it needs to be filled in or not. For this we set a threshold level, and played with it trying several sizes in the range of 20-150, to determine which one produced a better quality. In the end we found a threshold of around 50 produced the best results.

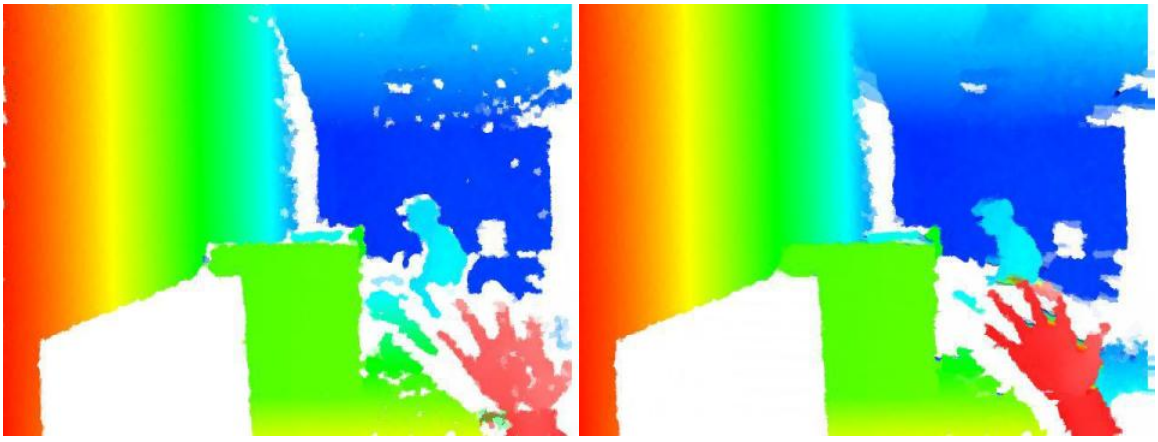

Figure 14: On the left, without fillHole. On the right, with it.

3.6.6 Horizontal and Vertical Interpolation

Another way to remove small holes is to loop through your image, upon finding some no data, keep going until you find the end of this no data. One you find the beginning and the end, you can get the good pixel before it and after, and linearly interpolate between them, filling in the no data. This will give you better results than streaking because you incrementally change each pixel value along the way, resulting in a smooth transition. This also allows you to determine the size of the hole, and decide if it was too big to be filled in. We did this in two ways, horizontally and vertically.

Horizontally interpolating was the fastest with an average time of 0.0007s per frame, however the quality seemed somewhat lacking. Vertically interpolating seemed to have the best results for its speed. Quality wise, it seemed on par with the flood fill method, but much faster with an average of 0.0015s per frame. The size threshold is set for both horizontal and vertical interpolation ended up about the same at 35 and 30 pixels respectively. Overall we think that for speed purposes and quality purposes vertical interpolation is the best method of the several that were tested. See figures below.

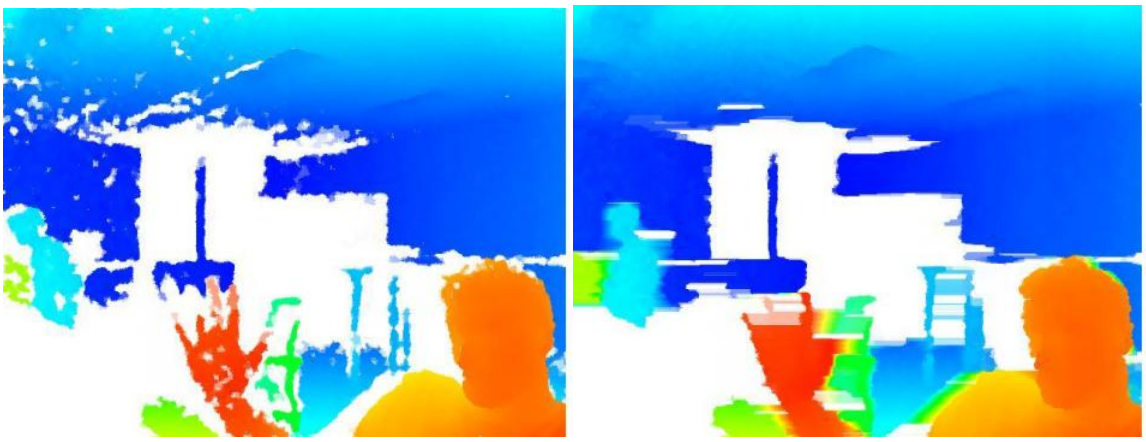

Figure 15: On the left, without horizontalInterpolate. On the right, with it.

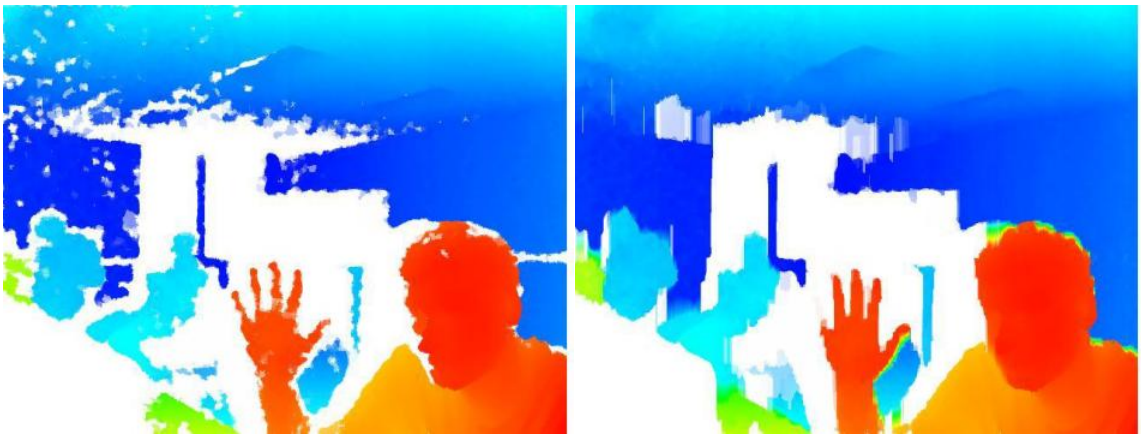

Figure 16: On the left, without verticalInterpolate. On the right, with it.

3.7 Three-Dimensional Model

In order to make the transition from RGB and depth images into a three dimensional model it was decided that drawing polygons and using texture mapping to place an image on top of these polygons was a good place to start. The texture to be used would be the RGB image repeatedly coming from the Kinect. How to draw the polygons then became the issue. We decided to just use the depth values as the Z positions of the polygons. What we did is draw one polygon for every pixel, so there will be 640 polygons in the X direction and 480 in the Y direction. This makes their x and y positions of each polygon easy. For the depth, we simply use the depth value of that pixel, and the pixel to the right and below it to get the four corners.

When you first view this, you only see the RGB image; it is not until your rotate the image that you can see the depth that stands out. This worked surprisingly well, and once a bug was fixed the polygons all lined up pretty well. Still there were many issues that needed to be cleaned up such as stretched polygons, texture offsets, and usability.

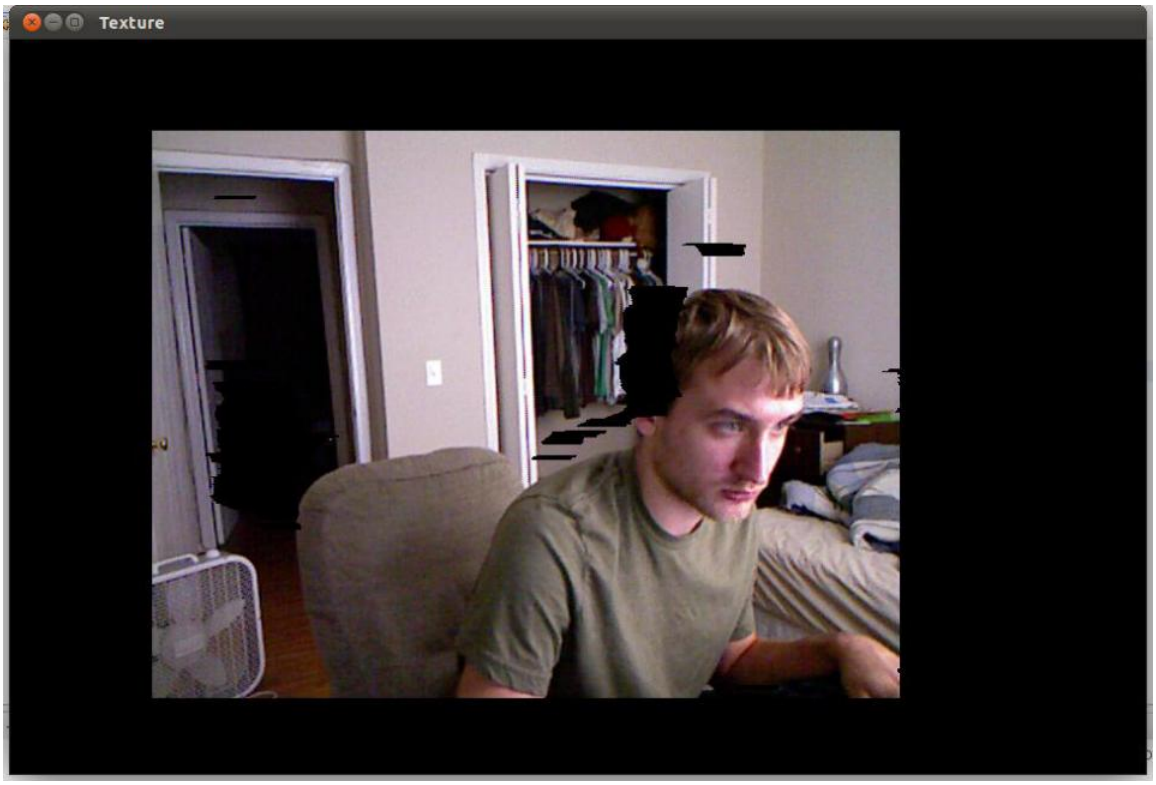

Figure 17: Three dimensional model

3.7.1 Usability

The next thing we decided to do was make it more useable. It is hard to tell how well it is working if you cannot see the depth of the polygons. Therefore we needed to rotate it. We made two modes rotate, and translate. You can change between them by pressing 'r' or 't'. Once in rotate mode, you click your mouse on the point you want and drag the mouse. The scene rotates around this point. Once in translate mode, it works the same click and drag to move the scene to the position you want. You can also, at any point, zoom in and out by using the mouse scroll wheel. These three tools also you to inspect the model in great detail, allowing for much better idea of what is being displayed.

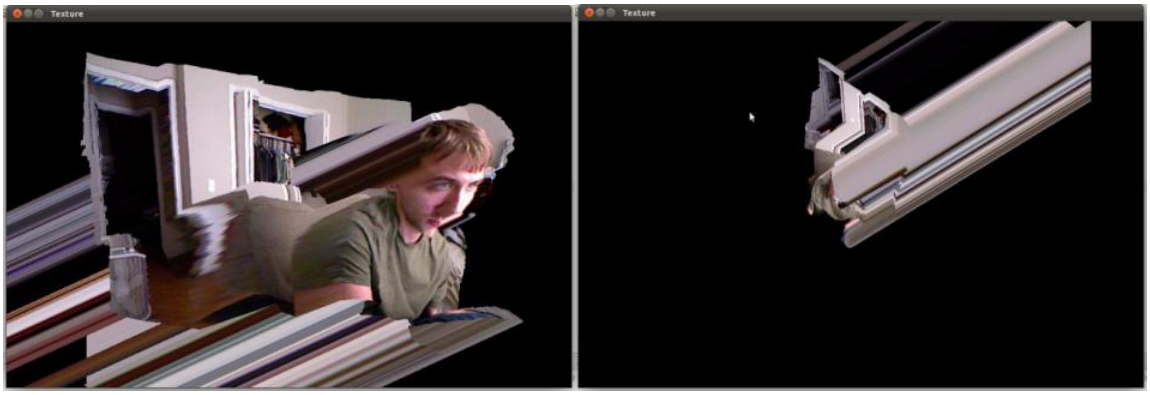

Figure 18: Showing Rotate, Translate, and Scale

3.7.2 Improvements to the Model

The first thing you start to realize when inspecting the image is that while it does depth properly, you get polygons that are stretched way back, causing the scene to look distorted. The reason for this is that in those areas, there is no depth information. It is on an edge of an object, and the next object is much farther back, so when it draws the polygons it attempts to stretch way too far. To combat this we decided the best way is to simply not draw those polygons. To do this, whenever it is about to draw a polygon, you simply get the maximum of the four depth values, and the minimum. You can then determine if the maximum minus the minimum is greater than some threshold. What this does is it allows us to determine how far that polygon stretches, if it is too far, simply do not draw that polygon.

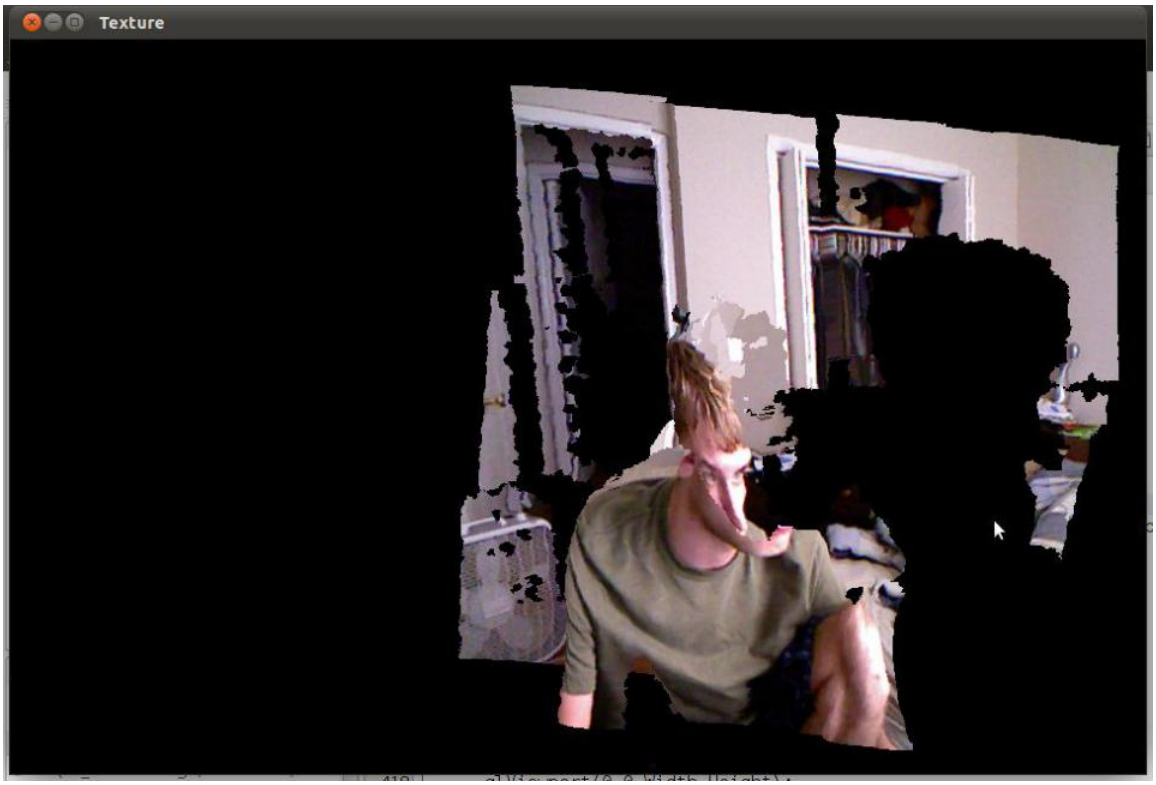

Figure 19: Not drawing stretched polygons resulting in a more comprehendible image.

3.7.3 Texture Alignment

The sensor for the depth and the RGB camera are not in the exact same spot. This results in the texture and depth, not perfectly lining up. You could at this point go through a lengthy calibration procedure to make sure that everything works perfect. However, through observation, it seems that the texture was simply off to the right and above where it should be. It did not seem as if there was any scaling. To counter this we chose to simply allow for manual correction. You can move the texture until it seems to fit, these positions are then saved to a file, and read in the next time the program is opened, allowing the texture to be in the correct position, and these can be further modified, if it seems they are not quite correct.

3.7.4 Combining Two Models

In order to tackle the combination of the two models we took a similar approach to how we adjusted the texture. You start off with two models, you then choose which model you want to be able to control, or if you want to control both models at once. The user then will manually translate, and rotate both models, until they are in the correct position. Once there, these translations and rotations will also be stored in a file, to be read at the start of the program.

Obviously there will be points where these models collide, where they both have the same information. Our initial way of handling this is to simply allow OpenGL Z buffering to handle this. Z buffering will only allow the closest polygon, which is hopefully the best polygon, to display. A problem with this is that obviously this will only work for your current set-up. Once you move the Kinects, they will no longer be in the correct positions.

What we found from experimenting with this, it is possible to merge the models from two different Kinects and produce a combined model with more information in it. Such as, one Kinect will see one side of the room, and the other will see the other half of the room, together you will have a much wider view of the room. This can be seen in the figures below.

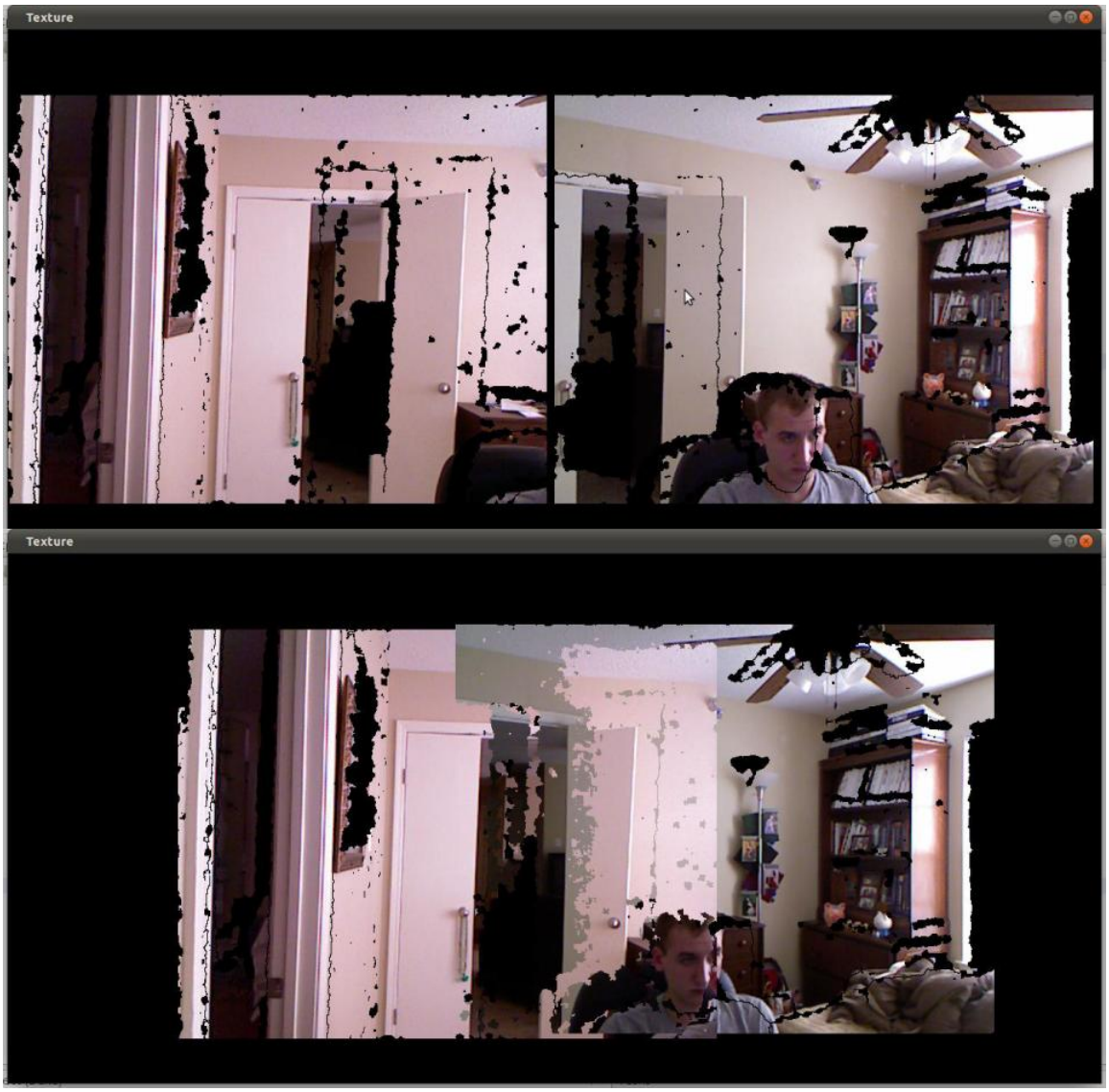

Figure 20: Original two models on top, combined on the bottom.

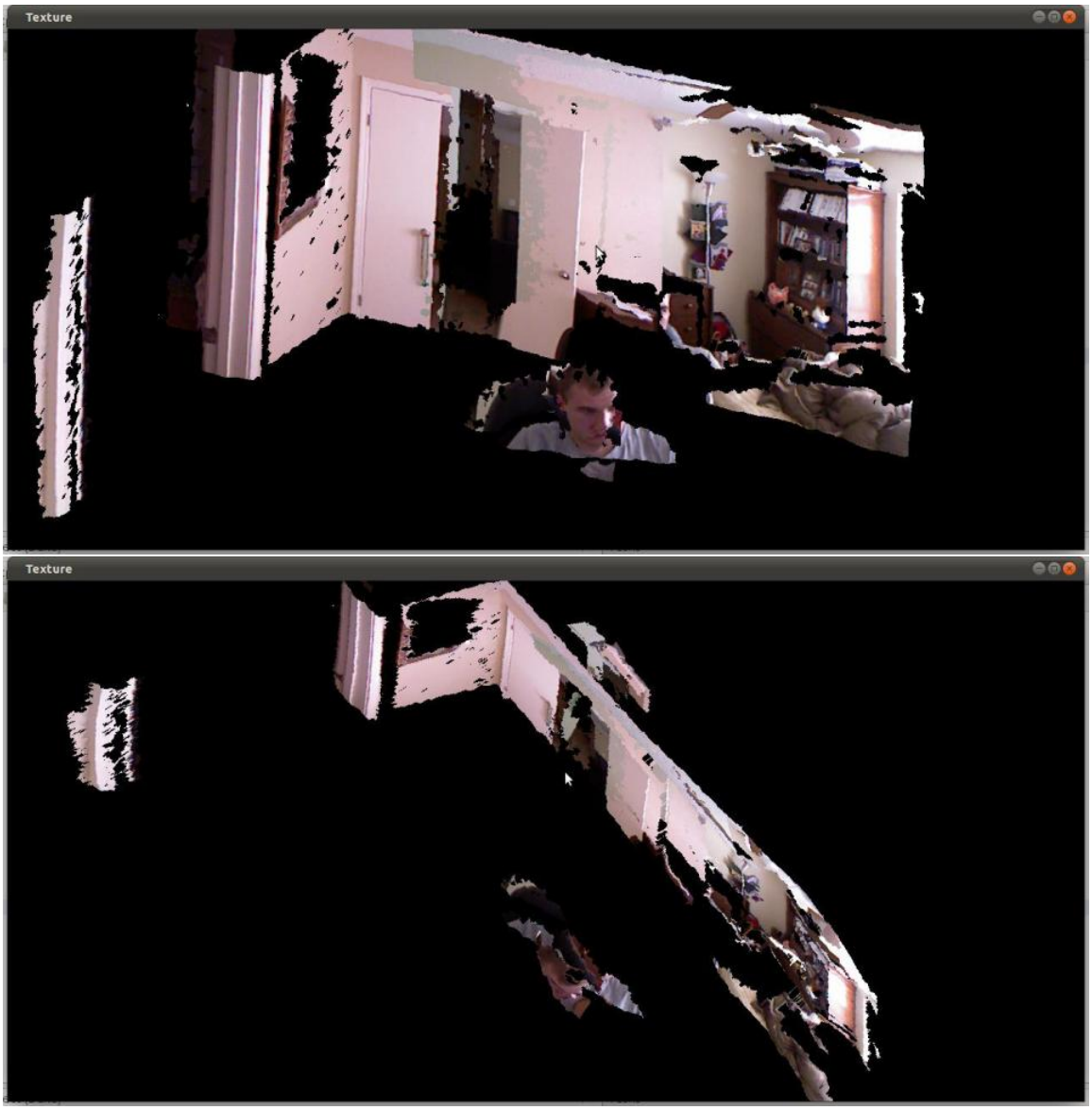

Figure 21: Combined model from two different angels, showing improved visibility.

There are some noticeable problems with this approach however. For one the Z buffer, and the distance values from the Kinects are not perfect, and you can get a lot of polygons appearing and disappearing which can be distracting. The other is, this only works well depending on your Kinect setup. It becomes very difficult to line up the models perfectly and in some cases near impossible. Two models with vastly different angles of view on things make it hard to line

straight lines such as doors up. This can be seen in the figure below. For this approach to work, the Kinects need to have as little in common, as far as their contents, as possible. Yet they also need to be on near the same level, as well as close together to avoid alignment issues. This defeats one of the goals of actually having a more complete model of objects that overlap.

Overall this method works, but needs significant overhaul to produce good results. Augmenting it with other techniques, such as more powerful monitoring of polygons, and mathematical alignment methods are necessary.

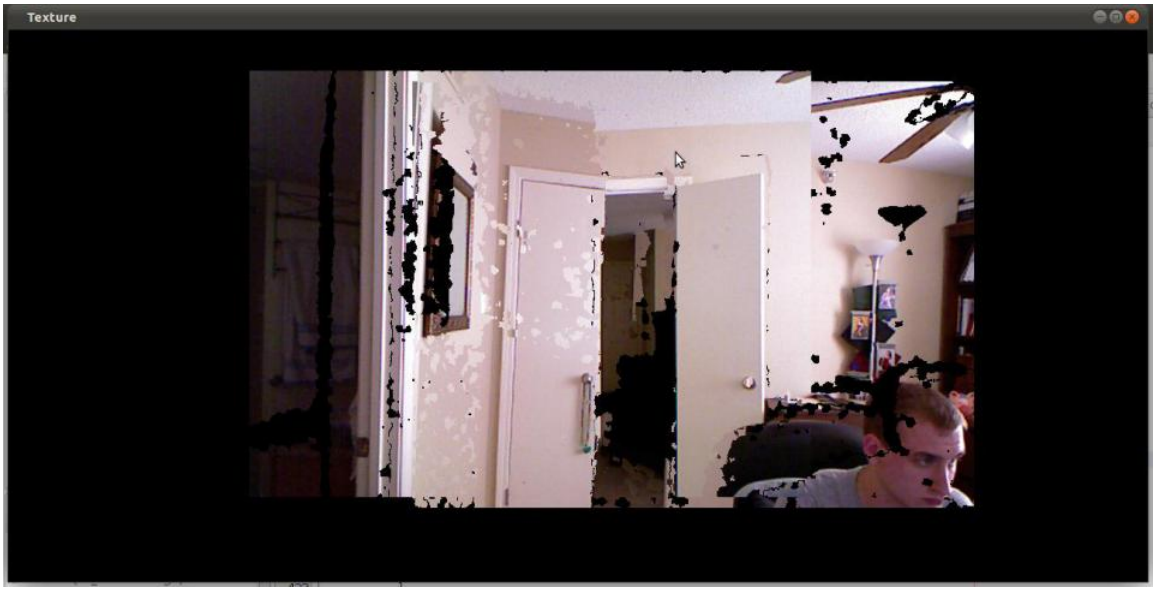

Figure 22: Combined model with too much in common and different angles resulting in alignment issues.

4. CONCLUSIONS

The Kinect presented the opportunity to get not only the RGB image associated with the scene, but also the depth information along with it. In this thesis, two of these devices were tested and used in order to create a 3D reconstruction of the scene.

We implemented classes and methods for the retrieval of Kinect information using the OpenKinect API but in the format that allowed the greatest usability. We explored ways of improving the depth information we received from the Kinect in the form of time-based averaging, and location-based averaging, comparing the advantages, disadvantages, and speed of each.

Finally, we used the depth information to create a 3D reconstruction of the scene using one Kinect, and presented an implementation that allows you to combine two such models. Once rotated, translated, and scaled correctly, these models could line up and provide an improved scene reconstruction. However, more powerful techniques are needed to produce better results and more robustness. Currently our thesis is limited to certain physical setups with the Kinects in order to have a combined model with more information in it than a model from a single Kinect.

5. FUTURE WORK

While the work that was done on this thesis successfully reconstructs a three dimensional model from a Kinect on a computer there is more that can be done. Right now the thesis is very limited, in that it requires manual calibration. In the future it would be best to use the wealth of data to do a form of automatic calibration. One way to do this is to calculate the depth of objects in the classic way, find objects in two images and guess at their depth, and then use this in conjunction with the real depth information. Another is the use access the Z buffers of OpenGL and rotate these around until the two Z buffers have the best match.

REFERENCES

[1] "PrimeSense, NITE Middleware." *PrimeSense, NITE Middleware*. Web. 12 June 2011. <http://www.primesense.com/?p=515>.

[2] "Open Kinect." *Open Kinect*. Web. 12 June 2011. <http://openkinect.org/wiki/Main_Page>.

[3] Kreylos, Oliver. "Oliver Kreylos' Research and Development Homepage - Kinect Hacking." *Oliver Kreylos' Research and Developement*. Web. 12 June 2011. <http://idav.ucdavis.edu/~okreylos/ResDev/Kinect/>.

[4] SangUn Yun; Dongbo Min; Kwanghoon Sohn; , "3D Scene Reconstruction System with Hand-Held Stereo Cameras," *3DTV Conference, 2007* , vol., no., pp.1-4, 7-9 May 2007

[5] Microsoft. "Kinect - Xbox.com." *Xbox 360 - Official Site - Xbox.com*. Microsoft. Web. 19 June 2011. <http://www.xbox.com/en-US/kinect>.

[6] Hoon Kang; Seung Hwan Lee; Jayong Lee; , "Image segmentation based on fuzzy flood fill mean shift algorihm," *Fuzzy Information Processing Society (NAFIPS), 2010 Annual Meeting of the North American* , vol., no., pp.1-6, 12-14 July 2010

[7] Levy, M.; , "The Impulse Response of Electrical Networks, with special reference to the Use of Artificial Lines in Network Design," *Electrical Engineers - Part III: Communication Engineering, including the Proceedings of the Wireless Section of the Institution, Journal of the Institution of* , vol.90, no.12, pp.153-164, December 1943