2013

Augmented Reality Using the Kinect Camera

Jonathan Alaric De Wolf
Bard College

Augmented Reality Using the Kinect Camera

A Senior Project submitted to
The Division of Science, Mathematics, and Computing
of
Bard College

by
Jonathan De Wolf

Annandale-on-Hudson, New York
May, 2013

Abstract

This project was originally inspired by Google Glass, a pair of glasses that present the wearer with a digitally augmented view. Applications include projecting information about someone you are talking to, checking weather, and translation services via the glasses [1].

The goal of my project was to create an Augmented Reality using the Kinect Camera. This program takes a live feed from the Kinect RGB Camera and augments it with images of additional objects. These objects are placed in the feed in a realistic manner, so that they obscure the images of real objects that lie "behind" the added object but not of objects that lie "in front" of the artificial object. The depth sensor of the Kinect is used to distinguish which objects in the real world should be obscured. This project explored various ways of using the Kinect, manipulating images based on data, and working with the Windows Presentation Format.

Contents

List of Figures

Dedication

To my family for their immense interest and support in my studies during my time at Bard College.

Acknowledgments

I would like to thank my adviser Rebecca Thomas for her help with this project. I would also like to thank my parents for their constant support.

1
Introduction

This project involves using the Kinect Camera's RGB (red/ green/ blue) and depth cameras as a basis for an augmented reality. Using the depth camera, I augmented a live camera feed from the RGB camera with images of additional objects.

1.1 Augmented Reality

An Augmented Reality is a live view of the real world that is augmented with virtual objects. One of the most well known uses of augmented reality today is the yellow line that marks the line of scrimmage in televised football. The yellow line isn't actually on the field, but when shown on television, the live video feed of the game is augmented with this line as seen in Figure 1.1.1. Augmented reality allows objects that wouldn't normally be seen together to be able to be seen together. James Fahey describes Augmented Reality as "the artificial, seamless, and dynamic integration of new content into, or removal of existing content from, perceptions of reality." [2] This broad definition hints at the variety of possible applications of augmented reality. A reality that is digitally altered positively or negatively can be considered an augmented reality.

Figure 1.1.1. Line of Scrimmage in Televised Football [2]

1.2 Software: Microsoft Visual Studio 2010 C#

For this project, Microsoft Visual Studio 2010 C# Express Edition was used as the programming platform. It is free software that allows for coding using the Kinect hardware via the Kinect for Windows Software Development Kit (SDK). The Microsoft.Research.Kinect version of the Kinect for Windows SDK requires use of the .NET Framework, a popular development platform for Windows applications which requires the use of either Visual Basic or C#; for this project, Microsoft Visual Studio C# was used. A Windows Presentation Format (WPF) application was created in Visual Studio. WPF uses XAML, which is an XML based language, to define user interface elements of the program.

1.3 Hardware: The Kinect Camera

The Kinect Camera is a camera developed by Microsoft for the Xbox 360 gaming console. The Kinect Camera has a red/ green/ blue camera, infrared emitter, depth camera, tilt motor and microphones. The Kinect has a USB port and a power port. It was developed

in order to allow users to play games on the Xbox 360 without a hand held controller. In order to do this, the Kinect camera can track humans via skeletal tracking, get depth information regarding the view of the camera, and recognize voices. The physical setup is described in Figure 1.3.1. The IR emitter sends out IR frequencies that the depth camera can then read to calculate depth. The tilt motor allows for motorized tilting of the Kinect 27 degrees up and down. The tilt motor requires additional power, which is why the Kinect camera has both a USB port and a power port. The microphones allow for voice control and recognition [3]. For this project, the RGB camera, IR emitter, and depth camera were used.

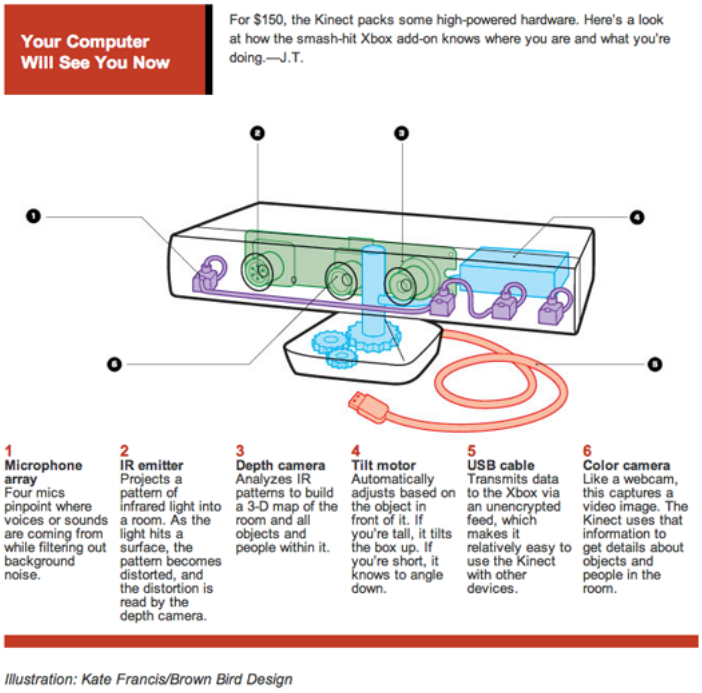

Figure 1.3.1. Breakdown of the Components of the Kinect Camera [4]

1.3.1 The Depth Camera

The Kinect depth camera is essentially an IR sensor. The IR emitter emits infrared beams that the depth camera can capture when they reflect back to the sensor. The reflected

beams are converted to depth information measuring the distance between the camera and the reflecting surface. The Kinect depth camera is located next to the RGB camera as seen in Figure 1.3.1. The depth camera outputs depth information at a resolution of 640 x 480 pixels by default. It can be sharpened to 1280 x 1024, but at the cost of a reduced frame rate. The frame rate is adjustable depending on resolution; this project uses 30 frames per second and a resolution of 640 x 480 pixels. The depth camera can accurately calculate the depths of most pixels in the view of the camera from 800 millimeters to 4000 millimeters (approximately 2.6 feet to 13.1 feet) [3]. The depth information can become inaccurate after around 13 feet, but it can still calculate depth up to roughly 18 to 20 feet and as close as 2 feet with reduced accuracy. The field of view is 43 degrees vertically and 57 degrees horizontally [3].

1.3.2 The RGB Camera

The Kinect RGB camera is located in the center of the camera, between the IR emitter and depth camera. The RGB camera outputs a video stream via three channels of data: red, green, and blue. The default resolution for the video stream is 640 x 480 pixels, but resolutions of up to 1280 x 1024 are supported. However, a resolution of 640 x 480 is ideal for a fast running program with lots of computation. The frame rate is adjustable from 9 to 30 frames a second; this project uses a frame rate of 30 frames per second and a resolution of 640 x 480 pixels.

1.4 Why use the Kinect?

The Kinect camera was used for this project due to its small resolution, depth camera, and software development kit. Since the Kinect camera has a small default resolution of 640 x 480, image computation can be done relatively quickly. This is very important, because image computation is being done in real time so any delay is noticed by the viewer. The

depth camera allows for separation of objects, and makes it easier to augment the RGB camera feed with virtual objects. The Kinect camera has one of the cheapest available depth cameras, so it was an ideal choice. The Kinect for Windows SDK is free and allows for users to create programs that use the Kinect in any way they wish.

1.5 Related Work

Many other projects are being conducted using the Kinect for augmented reality. The Kinect is being used to review training for sports in virtual environments. In the augmented reality being created, a user's position and movement is captured and the Kinect allows the user to interact with the virtual world. It was found that users training for sports in an augmented reality could successfully transfer this skill only if similar practice conditions were met in the real world [5]. Another related project involves using puppets for 3D animation. The user can load in a 3D image of an object. The user can move the objects around in real time and an animation is displayed on a screen. The user becomes a puppeteer for a live 3D animation. The Kinect is used to track the real world objects being augmented with the virtual background [6]. This is in contrast to my project where a real world video feed is being augmented with virtual objects. Another project is an augmented reality sandbox. In this sandbox, you can use a real shovel to move around virtual sand and water particles. Using a digital projector and Kinect camera, a 3D virtual sandbox was created and virtual particles can be moved via a real object [7].

2
Augmented Reality using the Kinect Camera

This chapter describes my method for creating an augmented reality with the Kinect camera. First, the virtual images must be created and coded in XAML. Next, the camera sensors and camera feeds must be initialized. Following this, depth calculation was completed. Then, the images defined in XAML must be manipulated. After this, the RGB camera feed has to be augmented with those images. Finally the user interface had to be generated.

2.1 Image Creation

The main window of the application (called MainWindow) was created as a 700 x 1200 pixel window with a Grid in XAML. XAML is a XML based language that is used to create and display images and the overall structure of the interface of the program. Images are created at specified positions in the Grid. In XAML, a Grid is a layout panel that arranges controls by position. Controls can be anything such as an image, button, or camera feed. Figure 2.1.2 shows the Grid structure. You can see the outline of the camera feed with smaller boxes (other images) inside it. Based on the location of the images in the XAML

```
<Window x:Class="WpfApplication1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="700" Width="1200" Loaded="Window_Loaded"
        Name ="mainWindow" Closed="Window_Closed">
    <Grid>

        <Image Height="480" HorizontalAlignment="Left" Margin="256,140,0,0"
               Name="depthImage" Stretch="Fill" VerticalAlignment="Top" Width="640" />
        <Image Height="480" HorizontalAlignment="Left" Margin="256,140,0,0"
               Name="cameraFeed" Stretch="Fill" VerticalAlignment="Top" Width="640" />

        <Label Content="Augmented Reality with the Kinect Camera" Height="96"
               HorizontalAlignment="Left" Margin="50,12,0,0" Name="label1"
               VerticalAlignment="Top" Width="1073" FontWeight="Bold" FontSize="64" FontFamily="Monotype Corsiva" />

        <Button Content="Click to add Table" Height="28" HorizontalAlignment="Left"
                Margin="50,280,0,0" Name="buttonTable" Click="tableClicked" VerticalAlignment="Top" Width="127" />
        <Image Height="120" HorizontalAlignment="Left" Margin="50,314,0,0"
               Name="table2" Stretch="Fill" VerticalAlignment="Top" Width="160"
               Source="file:///C:/Sproj/SeniorProject_Jonathan_DeWolf/SeniorProject_Jonathan_DeWolf/table.jpg" />
        <Image Height="480" HorizontalAlignment="Left" Margin="256,140,0,0"
               Name="image2" Stretch="Fill" VerticalAlignment="Top" Width="640" />
```

Figure 2.1.1. XAML Image Creation

code, these images are either in front of or behind other images by default. Images farther down in the code are in front of images that are farther up. For example, the video feed cameraFeed is in front of the depth feed depthImage in Figure 2.1.1. These two images are created in XAML at the same position, and have the same size, making the cameraFeed fully overlay the depthImage so that it is hidden. The depth image is only used for computation in this project.

Figure 2.1.1 shows how a table is loaded into the program via XAML beginning with <Button Content=“Click to add Table”...>. The next two lines of code create a button that will, when pressed, call a tableClicked method. This button is called buttonTable. The following two lines create an image under the button that shows the source file on the user interface. The next two lines create a blank image in which the source will be defined later. In order to define an image in XAML, the image height, width, alignment, margin, name, and stretch must be defined in XAML. The images used are green screened, for easy cutting of the image from its background. This means that the image background is green instead of white, and since none of the images have green in them, the green will be removed cleaner than white would be. The margin is the location of the image, and the

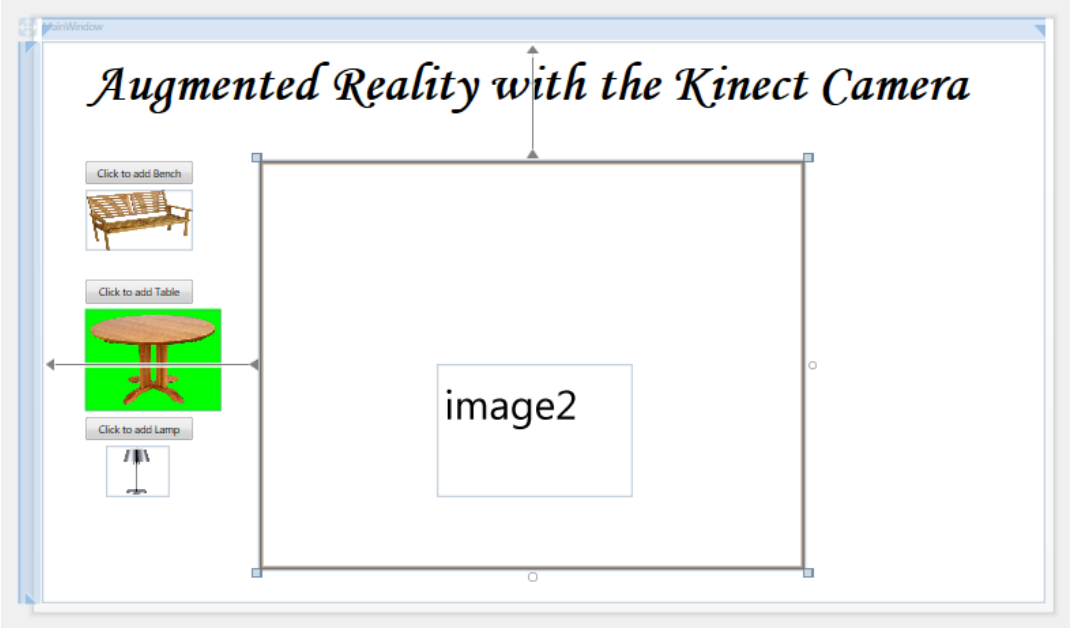

Figure 2.1.2. Designer: The Basic Design of Interface

stretch is how the image fills the specified dimensions given in XAML. The source of the image can be defined in the C# code or XAML. The image table2 can be seen in Figure 2.1.2 on the left under the Click to add Table button. The source of image2 is undefined in XAML (it is defined later in C#), thus image2 is not shown in the designer [3]. The image2 image coded in XAML is shown as image2 in the designer as seen in 2.1.2.

2.2 Initializing the Kinect Camera

In order to initialize the Kinect Camera's sensors, a runtime must be created. This runtime will then allow us to open video streams for both the depth and RGB camera on the Kinect. The runtime is used to initialize these streams and also handle the updating of frames, which is done via an event handler. Separate XAML code will define the placement of the two video streams. In XAML, two 640 x 480 images need to be specified with their sources as the respective RGB and depth Kinect sensors. The Window_Loaded event is called when the WPF application window is loaded and shown on the screen. This event

creates the window and loads the user interface. When the runtime is initialized, only the color and depth sensors are needed. The video and depth streams are updated frame by frame via an event handler. Both streams are set at 640 x 480 pixels, the same size as specified in the XAML, and the streams are opened. If you are looking at just the two streams it will look something like Figure 2.2.1.

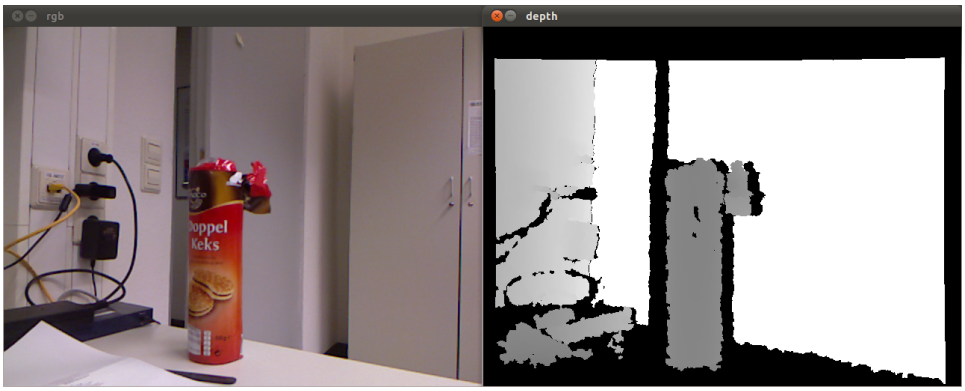

Figure 2.2.1. RGB and Depth Images side by side [9]

In both the video stream event handler and the depth stream event handler, an image is created from the image frame from the Kinect. This image is then converted to a BitmapSource via the BitmapSource.Create() method which takes information about the size and format and creates a new BitmapSource object from an array of pixels. The Pixel Format being used is BGR32, which has a blue channel, green channel, red channel, and an empty channel. Each pixel has 32 bits of information, 8 bits per channel. In order to get the array of pixels from the Kinect, after getting the image from ImageFrame.Image(), image.Bits gets the array of pixels for that image. The stride of an image is the distance in bytes from the beginning of a row of the image to the beginning of the next row, in other words the number of bytes per row of pixels. In this case the stride is the width of the image in pixels times the bytes per pixel. The depth stream event handler also calls the DepthCalculation() method, giving the depth image frame as its argument. The DepthCalculation() method calculates the depth of every pixel in the image and returns

an array of bytes. Figure 2.2.2 shows the initialization of the two Kinect sensors and the event handlers. In addition to the code seen in Figure 2.2.2, the two streams need to be initialized in XAML as well [3].

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    runtime = Runtime.Kinects[0];
    runtime.Initialize(RuntimeOptions.UseColor | RuntimeOptions.UseDepth);
    //kinect information used

    runtime.VideoFrameReady += new System.EventHandler<ImageFrameReadyEventArgs>
        (mainWindow_Video);
    //begins new EventHandler

    runtime.VideoStream.Open(ImageStreamType.Video, 2,
        ImageResolution.Resolution640x480, ImageType.Color);
    //poolSize = 2 is num of frames buffered

    runtime.DepthFrameReady += new System.EventHandler<ImageFrameReadyEventArgs>
        (mainWindow_Depth);
    //similar to videoFrameReady above

    runtime.DepthStream.Open(ImageStreamType.Depth, 2,
        ImageResolution.Resolution640x480, ImageType.Depth);
    //similar to videoFrameReady above
}

void mainWindow_Video(object sender, ImageFrameReadyEventArgs e)//event handler 1
{
    PlanarImage data = e.ImageFrame.Image;
    cameraFeed.Source = BitmapSource.Create(data.Width, data.Height, dpi, dpi,
        PixelFormats.Bgr32, null, data.Bits, data.Width * data.BytesPerPixel);
    colorArray = data.Bits;
}

void mainWindow_Depth(object sender, ImageFrameReadyEventArgs e)//event handler 2
{
    depthCBytes = DepthCalculation(e.ImageFrame);//array that will hold depth data
    PlanarImage data2 = e.ImageFrame.Image;
    depthImage.Source = BitmapSource.Create(data2.Width, data2.Height, dpi, dpi,
        PixelFormats.Bgr32, null, depthCBytes, data2.Width
        * PixelFormats.Bgr32.BitsPerPixel / 8);
}
```

Figure 2.2.2. Initialization of the Kinect Sensors

2.3 Depth Calculation

Depth was calculated in a method called DepthCalculation(). The data that the Kinect depth camera sensor gets is 16 bits (2 bytes) per pixel. A bitshift must be done in order

```
private int GetDistance(byte firstFrame, byte secondFrame)
{
    //return int

    int distance = (int)(firstFrame | secondFrame << 8);
    //Bitshift by 8 for distance

    return distance; //return int dist
}
```

Figure 2.3.1. GetDistance()

to get the distance in a unit of measure we are familiar with, millimeters. This is done in the GetDistance() method seen in Figure 2.3.1. GetDistance takes two bytes and returns an integer. In order to convert the two bytes to an integer, the second byte is shifted left 8 bits and the first and second bytes are combined via a bitwise OR operation. The second byte is shifted, concatenated with the first via the OR operation, then returned as an integer [8].

When the DepthCalculation() function is called via the depth stream event handler, an image frame is taken as input. This image frame consists of the depth data directly from the Kinect depth camera. This image frame data is then put into a byte array called depthData, which consists of 2 bytes per pixel. As seen in Figure 2.3.1, the GetDistance function takes two bytes, so we have a nested loop in order to call that method. In Figure 2.6.1, y and x are incremented until they reach the maximum height and width respectively.

GetDistance is called and returns a variable of type int containing the distance. The distance is then compared with a set distance and set to different grayscale shades on the spectrum from 0 (black) to 255 (white), specifically either 0 or 200 for testing. The closer the pixel is to the camera, the lighter its color. The color is set via an array called depthFrame which consists of four bytes per pixel, where the first byte is the blue channel, second byte is the green channel, third byte is the red channel, and fourth byte is empty. The returned array is depthFrame, and it is set to a global array depthCBytes in the depth

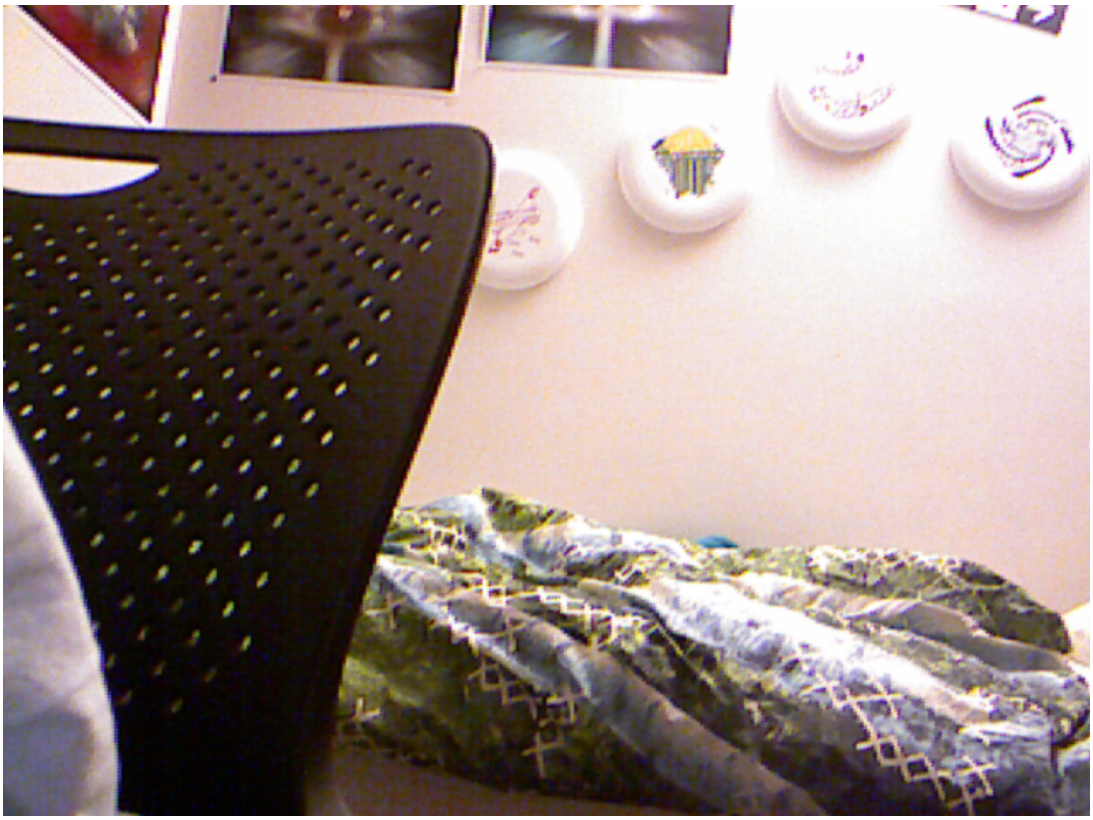

Figure 2.3.2. RGB Image from the Kinect Camera

stream event handler after DepthCalculation completes. Figure 2.3.3 shows the image depthFeed that was defined in Figure 2.1.1 calculated by the method DepthCalculation() for two different depth variations. The corresponding RGB image is seen in Figure 2.3.2 [3].

2.4 Image Manipulation and Augmentation

When a User clicks on buttonTable, the tableClicked() method is called. In the tableClicked method shown in Figure 2.4.1, a BitmapSource object called tableSrc is created via the table2 source, which was defined in XAML via a file path to the image of the table. The camera feed is augmented with images in a separate class called ImageCalculation. Thus an ImageCalculation object is created called imageCalc. Since image2's source was not

Figure 2.3.3. Depth Image calculated via DepthCalculation() from the Kinect camera with 2 depth regions (Close values are light grey and far values are black)

defined in XAML (see Figure 2.1.1,) we must define it now. In tableClicked, image2.Source is set to image imageCalc.writeToFeed(). This method takes a BitmapSource and two byte arrays as inputs. One of these arrays consists of all depth pixels in the image frame and the other consists of all color pixels in the image frame. The depth array is highlighted in Figure 2.4.1. When tableClicked is complete the table will show up not only under the button called buttonTable, but also augmenting the video feed cameraFeed.

Figure 2.4.2 shows the writeToFeed() method of the ImageCalculation class which returns a WriteableBitmap called target. This method is where the image manipulation is being done, so that the camera feed will be augmented with the table. The stride of the BitmapSource tableSrc defined previously is declared similarly to the previous strides: the

```
private void tableClicked(object sender, RoutedEventArgs e)
{

    BitmapSource tableSrc = table2.Source as BitmapSource;
    ImageCalculation imageCalc = new ImageCalculation
        (/*1, colorArray, depthArray*/);
    //bench2.BeginInit();
    image2.Source = imageCalc.writeToFeed(tableSrc, colorArray,
        depthCBytes);
    //bench2.EndInit();
}
```

Figure 2.4.1. tableClicked()

pixel width of the source image times the Bits per pixel divided by 8. A new empty byte array called data is declared with space for four bytes per pixel. The pixels are then copied from tableSrc to data via the CopyPixels method. This method takes three arguments: an array of pixels, a stride, and an offset (which is always zero for this project). A WriteableBitmap called target is created. A WriteableBitmap is a bitmap that can be written to and changed. In order to create a WriteableBitmap, image size and format must be declared. The pixel format used was Bgra32, consisting of four channels, with the fourth channel consisting of a byte representing the alpha channel or transparency. The alpha value is less opaque at lower values with 0 being fully transparent [3].

A nested for loop increments through the pixels in target. A single pixel by single pixel rectangle is created each iteration for use in the WritePixels() call below. An array called subdata is created for each pixel in the WriteableBitmap. A transparent pixel is created to be used to set pixels transparent. The most important part of this method is the WritePixels() call. A depth is specified for the image. WritePixels is only being called on a given pixel if subdata[1] is less then or equal to 250 (if the color at that pixel is not green) and if the depth values are equal to a specified depth (of the image). This can be changed to be greater than or equal to as well as seen in the Appendix code. So if the pixels are not green and the image is farther from the camera at a specific point then show

```
public WriteableBitmap writeToFeed(BitmapSource source, byte[]color, byte[] depth)
{
    int stride = source.PixelWidth * ((source.Format.BitsPerPixel )/ 8);

    byte[] data = new byte[stride * source.PixelHeight*4];
    source.CopyPixels(data, stride, 0);
    WriteableBitmap target = new WriteableBitmap(source.PixelWidth, source.PixelHeight,
        source.DpiX, source.DpiY, PixelFormats.Bgra32, null);
    int w = (int) target.Width;
    int h = (int) target.Height;

    for (int y = 0; y < h; y++)
    {
        for (int x = 0; x < w; x++)
        {
            Int32Rect rect = new Int32Rect(x,y, 1, 1);
            byte[] subdata = { data[x * 4 + stride * y], data[x * 4 + stride * y + 1],
                               data[x * 4 + stride * y + 2], data[x * 4 + stride * y + 3] };
            byte b = 255;
            byte g = 255;
            byte r = 255;
            byte a = 0;
            byte[] transparent = { b, g, r, a };
            byte ItemDepth = 0;

             if ((subdata[1]<= 250&& depth[(y*stride  +  x*4)] == ItemDepth) &&
                (depth[(y *stride +  x*4) + 1] == ItemDepth) && (depth[(y * stride + x*4) + 2]
                == ItemDepth))//try2
                   target.WritePixels(rect, subdata, stride, 0);
        }
    }
    return target;
}
```

Figure 2.4.2. writeToFeed() in ImageCalculation

that pixel. In the end, pixels being shown are those behind a specified depth so that if an object is in front of the table, the table will be partially occluded. The camera feed will be augmented with the table.

2.5 The User Interface

The User Interface can be seen in Figure 2.4.3; it consists of a set of images and buttons on the left hand side, with a view of the Kinect RGB camera on the right. Each time a button is pressed, that image is updated so if the camera is moved, you must press the

Figure 2.4.3. User Interface

button again and the image will update with the correct depth values. The interface is easy to use; you can simply run the program to initialize the window.

2.6 Class Structure

There are three different classes in the project: MainWindow.xaml, MainWindow.xaml.cs, and Class1.cs (Image Calculation) as seen in Figure 2.6.2. MainWindow.xaml consists of the XAML code to load in images, place images, and create the overall user interface of the program. MainWindow.xaml.cs contains the C# code for initializing the Kinect sensors, calculating the depth, and handling events such as button clicks and updating frames. Class1.cs contains the C# code for the ImageCalculation class, and contains the methods for augmenting images with the Kinect video feed.

```
    var depthIndex = 0;//+2 ea loop keeps index in depthData
    for (var y = 0; y < height; y++)
    {
        var yTimesWidth = y * width;

        for (var x = 0; x < width; x++)
        {
            //index in array
            var index = ((width - x - 1) + yTimesWidth) * 4;
            //returns int dist, since we use two vales from depthData,
                //we need to increment
            //depthIndex by 2 each time
            var distance = GetDistance(depthData[depthIndex],
                depthData[depthIndex + 1]);
            double i = 1.5;
            //2 color regions (version 1)
            if (distance <= 100*i)
            {
                //close dist
                depthFrame[index + BlueIndex] = 200;
                depthFrame[index + GreenIndex] = 200;
                depthFrame[index + RedIndex] = 200;
            }

            else if (distance > 100*i)
            {
                //mid dist
                depthFrame[index + BlueIndex] = 0;
                depthFrame[index + GreenIndex] = 0;
                depthFrame[index + RedIndex] = 0;
            }
            depthIndex += 2;
        }
    }
    return depthFrame; //return depths of image
}
```

Figure 2.6.1. A Section of DepthCalculation()

Classes and what's in them

MainWindow.xaml	**MainWindow.xaml.cs**	**Class1.cs (ImageCalculation)**
Contains the following Objects in Grid: depthImage cameraFeed buttonBench bench1 image1 buttonTable table2 image2 buttonLamp lamp3 image3	Contains the following methods: MainWindow() Window_Loaded() mainWindow_Video() mainWIndow_Depth() benchClicked() tableClicked() lampClicked() Window_Closed() DepthCalculation() GetDistance()	Contains the following methods: ImageCalculation() writeToFeed()

Figure 2.6.2. Classes and Methods

3
Results

I took a set of images with the depth data being shown and two depth values being used. This produced the best results since it was a simple case and it is easily to visually distinguish the error. I analyzed the images in order to evaluate how well my program completed the task of being to augment the RGB camera feed from the Kinect with images. These image feed frames were all taken immediately after clicking the button to realign the image being augmented with the feed to insure accuracy. I used the table image as the test image for these image captures. Light gray depth values are closer than the table, and black depth values are farther than the table. Thus if an object is closer than the table, it would show up light grey. These tests can be seen in Figures 3.0.1 through 3.0.8. Through these tests, I found that vertically the augmentation of images with the video feed was accurate, however horizontally it was not.

In Figure 3.0.1, the table is completely obscured. We can see that the table is completely hidden from view correctly since all of the pixels are in front of the table.

In Figure 3.0.2, the table is almost completely shown. It has a few breaks in the image, due to some noise in the bottom left hand side of the image, however it is mostly accurate

as almost all pixels are shown on the table. The noise most likely caused small pixelation in the image due to a stride error, since small pixel lines are taken out of the image horizontally, not vertically.

As seen in Figure 3.0.3, the table is obscured quite accurately for values in front of the table. However, some values are obscured which should not be in the lower part of the image. This test did a good job of getting rid of pixels horizontally that needed to be gotten rid of, however it also got rid of a few extra pixels in the lower part of the image.

In Figure 3.0.4, most of the upper pixels that should be obscured are obscured, however not as accurately as in Figure 3.0.3. The pixels that are not supposed to be obscured however are more accurate in Figure 3.0.4. Fewer extra pixels are incorrectly obscured. The error in both of these Figures is fairly low since most of the pixel separation between depth values is vertical. In one image, too many pixels are being erased, while in the other not enough are.

Looking at Figure 3.0.5, we see the same thing happening as in Figure 3.0.3: the Pixels that should be obscured are obscured correctly however extra pixels are being obscured as well. The difference now is that the bottom pixels should be obscured instead of the top pixels.

Figure 3.0.6 is the most accurate of the partially obscured images. It doesn't fully obscure the pixels that should be obscured, however it has a clear cutoff between the depths and does a very good job of keeping the pixels that should not be obscured.

In Figure 3.0.7, we begin to see the shortcomings of the program. With a horizontal cutoff of the depth data half the image is obscured however, banding occurs and the half that is obscured is not based on position and is uniform throughout the image. This is likely due to an indexing error, or stride error possibly in the ImageCalculation class. We also see errors when there is more noise in the depth data from the Kinect or the depth camera gets bad data as shown in Figure 3.0.8.

From these test images, we can conclude that vertical differentiation of depth data works to an extent, however horizontal differentiation does not work as well. Most likely this problem comes from an indexing error somewhere in the program. Likely this error is in the writeToFeed method seen in 2.4.2. It could have to do with the stride, which is the most likely reason that horizontal depth differentiation is not working entirely correctly. However, I debugged and tried several possible strides and found that the stride calculated in the code gave the most accurate results. Another possible error could be in calculating the index of the byte arrays being used; however the arrays should all be the same length since I am getting the bytes per pixel for each and calculating the length of the arrays the same way. The error seems to stem from the target.WritePixels() method call in writeToFeed since that is where the pixels are being written to the WriteableBitmap.

Figure 3.0.1. Completely Obscured table

Figure 3.0.2. Completely Unobscured table [10]

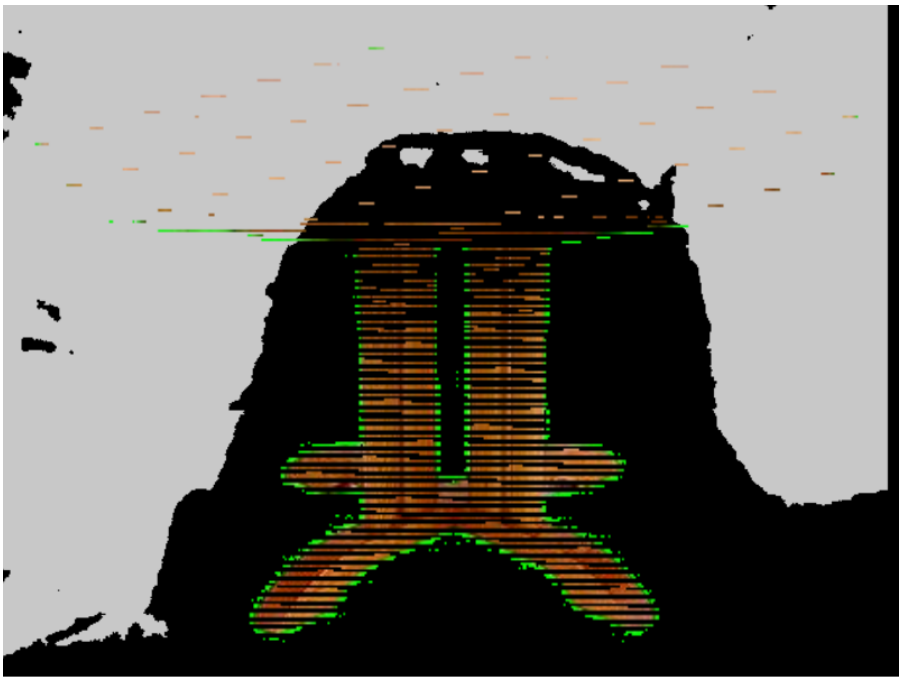

Figure 3.0.3. Partially Obscured upper part of table

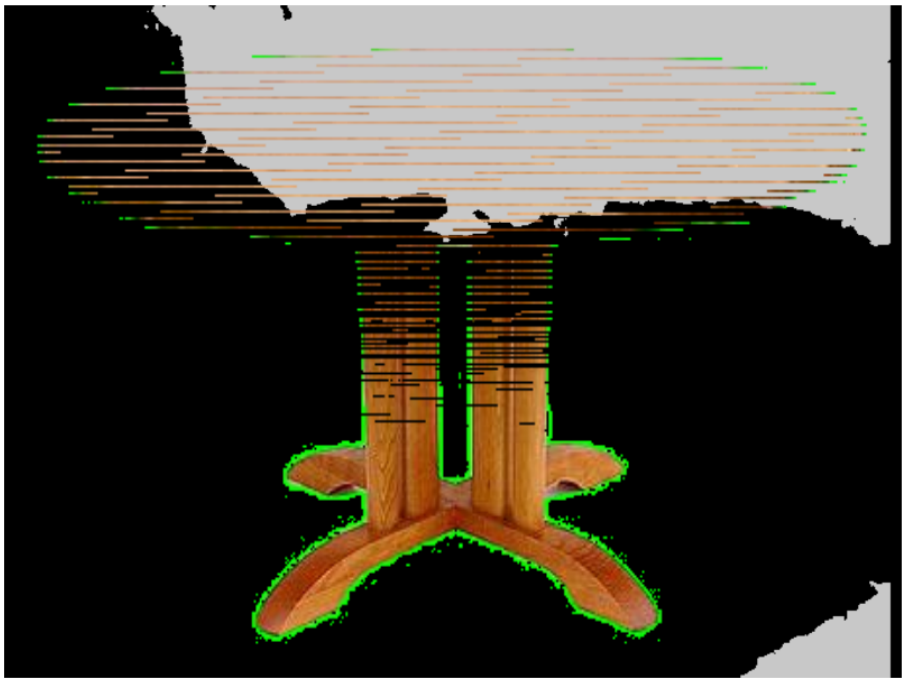

Figure 3.0.4. Partially Obscured upper part of table

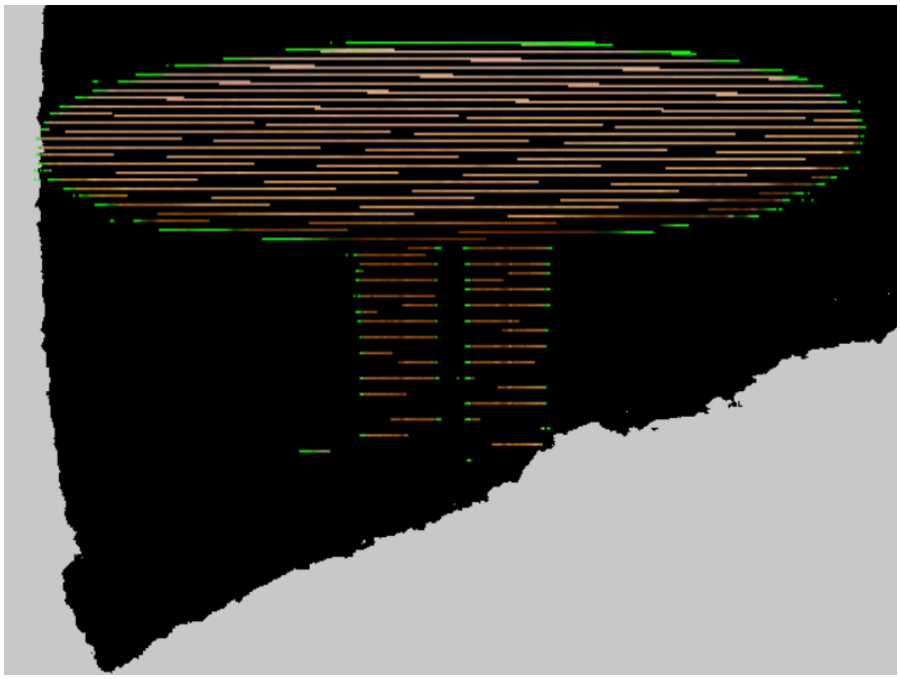

Figure 3.0.5. Partially Obscured lower part of table

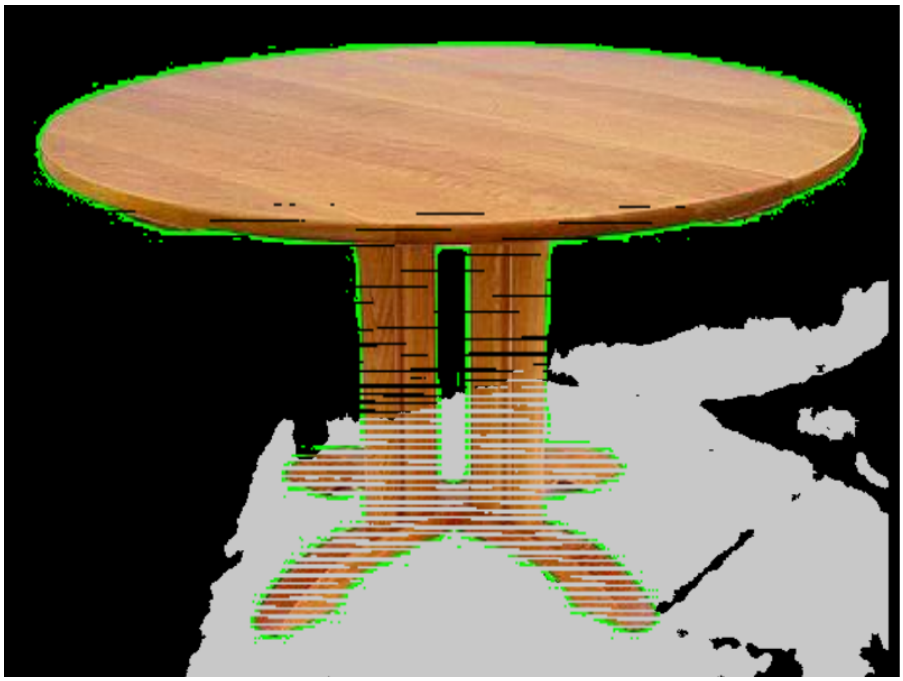

Figure 3.0.6. Partially Obscured lower part of table

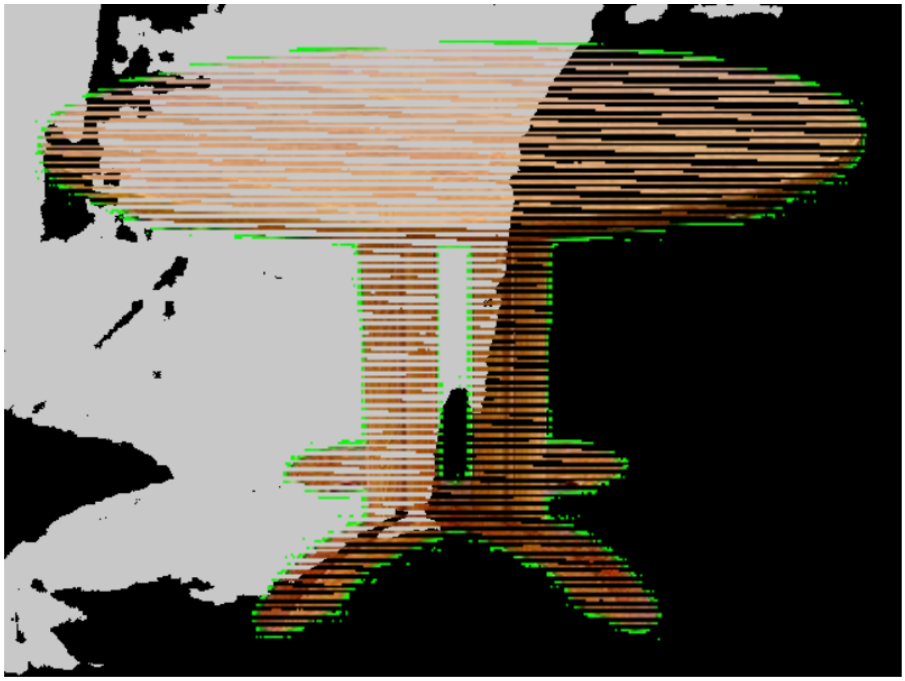

Figure 3.0.7. Partially horizontally Obscured table

Figure 3.0.8. Partially Obscured table with noise

4
Conclusion and Discussion

With more time, I would have liked to implement additional features and get those that were implemented working more accurately. I felt that I was able to get a significant amount of work complete; however, some aspects are not working as intended.

Of the features I have, I would have liked to get the horizontal obscuring of images working correctly or at least well as the vertical obscuring did. When augmenting the images with the video feed, depth differences in the vertical were much more accurately recognized than those in the horizontal direction. Also, getting rid of noise in the images, and using more advanced image manipulation techniques would be good to do in the future. In addition, after this was complete it would have been nice to have variable sizes of images working with the depth data, which would be easy to implement had I gotten both the vertical and horizontal augmentation working entirely correctly.

The features that I would like to add are the following. I would add the ability to tilt and rotate the camera, with live updating of the images augmented with the video feed based on movement. I would also add additional different images. The ability to update images in real time automatically would be useful; if this were implemented, I would

need to lower the frame rate or make the program more efficient due to small delays in image manipulation calculations, since all the calculations are being done in real time. The camera tilt angle is motorized so it would be fairly straightforward to automate, however rotating the camera is not. I would need to create a mechanical rotating system that would be similar to the tilt function build in to the Kinect camera. I would need to save image data in a three dimensional atmosphere of sorts as well if I were to be able to get it working with a rotating Kinect camera. All of these features would be next steps in my work on this project.

In addition, it would be nice to allow the user to see this augmented reality through glasses. This would give the user a first person perspective of the scene, while still allowing the images to be augmented realistically in a real world scene. To do this, I would need to expand the project to use 3D modeled objects, and use matrices to represent the 3D position and orientation of the objects [11].

5
Appendix

The code for the project is shown here, beginning with the MainWindow.cs C# class, followed by the ImageCalculation.cs C# class, and finishing with the XAML code in MainWindow.cs.xaml.

```
//BEGINNING OF CLASS**
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using Microsoft.Research.Kinect.Nui; //Library for kinect

//Main C# class where Kinect is used, initialized and event handlers are
namespace WpfApplication1
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
```

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }

    //create runtime
    Runtime runtime;

    byte[] depthCBytes;
    //array of depth pixels
    //length = [stride * source.PixelHeight * 4]

    byte[] colorArray;
    //array of color pixels
    double dpi = 96;//Resolution

    private void Window_Loaded(object sender, RoutedEventArgs e)
    {
        runtime = Runtime.Kinects[0];
        runtime.Initialize(RuntimeOptions.UseColor |
            RuntimeOptions.UseDepth);
        //kinect information used

        runtime.VideoFrameReady += new System.EventHandler
            <ImageFrameReadyEventArgs>(mainWindow_Video);
        //begins new EventHandler

        runtime.VideoStream.Open(ImageStreamType.Video, 2,
            ImageResolution.Resolution640x480, ImageType.Color);
        //poolSize = 2 is num of frames buffered

        runtime.DepthFrameReady += new System.EventHandler
            <ImageFrameReadyEventArgs>(mainWindow_Depth);
        //similar to videoFrameReady above

        runtime.DepthStream.Open(ImageStreamType.Depth, 2,
            ImageResolution.Resolution640x480, ImageType.Depth);
        //similar to videoFrameReady above
    }

    void mainWindow_Video(object sender, ImageFrameReadyEventArgs e)
        //event handler 1
    {
        PlanarImage data = e.ImageFrame.Image;

        cameraFeed.Source = BitmapSource.Create(data.Width, data.Height,
            dpi, dpi, PixelFormats.Bgr32, null, data.Bits, data.Width
            * data.BytesPerPixel);
        colorArray = data.Bits;
```

```
}

void mainWindow_Depth(object sender, ImageFrameReadyEventArgs e)
    //event handler 2
{
    depthCBytes = DepthCalculation(e.ImageFrame);

    //array that will hold depth data
    PlanarImage data2 = e.ImageFrame.Image;
    depthImage.Source = BitmapSource.Create(data2.Width, data2.Height,
        dpi, dpi, PixelFormats.Bgr32, null, depthCBytes, data2.Width
        * PixelFormats.Bgr32.BitsPerPixel / 8);
}

private void benchClicked(object sender, RoutedEventArgs e)
{

    BitmapSource benchSrc = bench1.Source as BitmapSource;
    ImageCalculation imageCalc = new ImageCalculation();

    //bench2.BeginInit();
    image1.Source = imageCalc.writeToFeed(benchSrc,
        colorArray, depthCBytes);
    //bench2.EndInit();

    //send data about size to depth calc (for variable sizes)
}

private void tableClicked(object sender, RoutedEventArgs e)
{

    BitmapSource tableSrc = table2.Source as BitmapSource;
    ImageCalculation imageCalc = new ImageCalculation();

    //bench2.BeginInit();
    image2.Source = imageCalc.writeToFeed(tableSrc,
        colorArray, depthCBytes);
    //bench2.EndInit();
}

private void lampClicked(object sender, RoutedEventArgs e)
{

    BitmapSource lampSrc = lamp3.Source as BitmapSource;
    ImageCalculation imageCalc = new ImageCalculation();

    //bench2.BeginInit();
    image3.Source = imageCalc.writeToFeed(lampSrc,
        colorArray, depthCBytes);
    //bench2.EndInit();
}
```

```
private void Window_Closed(object sender, System.EventArgs e)
{
    runtime.Uninitialize();
}

//DEPTH CALCULATION-------------------------------------------
//2 bytes per pixel
//Bitshift second byte by 8
//if 0, glass, shadows wont calculate well for depth data
//color information for all pixels in image:
//r,g,b,extra byte needed for bgr32 format so Hxwx4 is needed
//to store each peice of data for color infor. in image

private byte[] DepthCalculation(ImageFrame imageFrame)
    //generates the depth image for coloredDepthImage

{
    int height = imageFrame.Image.Height;
    int width = imageFrame.Image.Width;

    byte[] depthData = imageFrame.Image.Bits;

    byte[] depthFrame = new byte[imageFrame.Image.Height
        * imageFrame.Image.Width * 4];
    const int BlueIndex = 0;
    const int GreenIndex = 1;
    const int RedIndex = 2;


    var depthIndex = 0;//+2 ea loop keeps index in depthData
    for (var y = 0; y < height; y++)
    {
        var yTimesWidth = y * width;

        for (var x = 0; x < width; x++)
        {
            //index in array
            var index = ((width - x - 1) + yTimesWidth) * 4;

            //returns int dist, since we use two vales from
            //depthData

            //we need to increment depthIndex by 2 each time

            var distance = GetDistance(depthData[depthIndex],
                depthData[depthIndex + 1]);

            double i = 1.5;
```

```
                //2 color regions (version 1)
                if (distance <= 100*i)
                {
                    //close dist
                    depthFrame[index + BlueIndex] = 200;
                    depthFrame[index + GreenIndex] = 200;
                    depthFrame[index + RedIndex] = 200;
                }

                else if (distance > 100*i)
                {
                    //mid dist
                    depthFrame[index + BlueIndex] = 0;
                    depthFrame[index + GreenIndex] = 0;
                    depthFrame[index + RedIndex] = 0;
                }

                depthIndex += 2;
            }
        }
        return depthFrame; //return depths of image
    }

    private int GetDistance(byte firstFrame, byte secondFrame)
    {
        //return int
        int distance = (int)(firstFrame | secondFrame << 8);
        //Bitshift by 8 for distance

        return distance; //return int dist
    }

    //nec. methods for image failed (automatically generated)
    private void image1_ImageFailed(object sender,
        ExceptionRoutedEventArgs e)
    {

    }

    private void bench_ImageFailed(object sender,
        ExceptionRoutedEventArgs e)
    {

    }
  }
}
//END OF CLASS**

-------------------------------------------------------------
```

```
//BEGINNING OF CLASS**
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using Microsoft.Research.Kinect.Nui;
using System.Drawing.Imaging;

//Image Calculation class (Class1.cs) that deals with image computation
namespace WpfApplication1
{

    class ImageCalculation
    {
        public ImageCalculation()
        {

        }

        public WriteableBitmap writeToFeed(BitmapSource source, byte[]color,
            byte[] depth)
        {
            int stride = source.PixelWidth * ((source.Format.BitsPerPixel )/ 8);
            byte[] data = new byte[stride * source.PixelHeight*4];

            //copy pixel to array data
            source.CopyPixels(data, stride, 0);

            //creates WriteableBitmap that will be returned
            WriteableBitmap target = new WriteableBitmap(source.PixelWidth,
                source.PixelHeight, source.DpiX, source.DpiY,
                PixelFormats.Bgra32, null);

            //image w/h
            int w = (int) target.Width;
            int h = (int) target.Height;

            //-------------------------------------------------------LOOP(BEG)
            //loops through all pixels
            for (int y = 0; y < h; y++)
            {
                for (int x = 0; x < w; x++)
```

```
                {
                    //creates a 1 by 1 rectangle
                    Int32Rect rect = new Int32Rect(x,y, 1, 1);

                    //subdata is an array of the bgra values at a pixel
                    byte[] subdata = { data[x * 4 + stride * y],
                                       data[x * 4 + stride * y + 1],
                                       data[x * 4 + stride * y + 2],
                                       data[x * 4 + stride * y + 3] };

                    byte b = 255;
                    byte g = 255;
                    byte r = 255;
                    byte a = 0;
                    byte[] transparent = { b, g, r, a };//transparent pixel

                    byte ItemDepth = 0; //can be changed to a higher value,
                    //Any depth lower than this value is hidden from the
                    //image being added to the camera feed (lower values
                    //cooorespond to higher depths)


                    //SECTION 1----------------------------------------
                    if ((subdata[1]<= 250&& depth[(y*stride + x*4)] == ItemDepth)
                        && (depth[(y*stride + x*4) + 1] == ItemDepth)
                        && (depth[(y*stride + x*4) + 2] == ItemDepth))
                            target.WritePixels(rect, subdata, stride, 0);
                    //SECTION 1----------------------------------------

                    //SECTION 2----------------------------------------
                    /* //If ItemDepth is changed from 0, comment section 1 and
                       //uncomment this section, section 2
                    if ((subdata[1] <= 250 && depth[(ystride + x*4)] <= ItemDepth)
                       && (depth[(y*stride + x*4) + 1] <= ItemDepth)
                       && (depth[(y*stride + x*4) + 2]<= ItemDepth))
                           target.WritePixels(rect, subdata, stride, 0);
                     */
                    //SECTION 2----------------------------------------

                    //target.WritePixels(rect, transparent, stride, 0);
                    //writes transparent pixels
                }
            }
            //-------------------------------------------------------LOOP(END)
            return target; //returns a WriteableBitmap object
        }
    }
}
//END OF CLASS**
```

```
<Window x:Class="WpfApplication1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="700" Width="1200" Loaded="Window_Loaded"
        Name ="mainWindow" Closed="Window_Closed">
    <Grid>
        <Image Height="480" HorizontalAlignment="Left" Margin="256,140,0,0"
               Name="depthImage" Stretch="Fill" VerticalAlignment="Top" Width="640" />
        <Image Height="480" HorizontalAlignment="Left" Margin="256,140,0,0"
               Name="cameraFeed" Stretch="Fill" VerticalAlignment="Top" Width="640" />
        <Label Content="Augmented Reality with the Kinect Camera" Height="96"
               HorizontalAlignment="Left" Margin="50,12,0,0" Name="label1"
               VerticalAlignment="Top" Width="1073" FontWeight="Bold" FontSize="64"
               FontFamily="Monotype Corsiva" />
        <Button Content="Click to add Table" Height="28" HorizontalAlignment="Left"
                Margin="50,280,0,0" Name="buttonTable" Click="tableClicked"
                VerticalAlignment="Top" Width="127" />
        <Image Height="120" HorizontalAlignment="Left" Margin="50,314,0,0"
               Name="table2" Stretch="Fill" VerticalAlignment="Top" Width="160"
               Source="file:///C:/Sproj/SeniorProject_Jonathan_DeWolf/SeniorProject
               _Jonathan_DeWolf/table.jpg" />
        <Image Height="480" HorizontalAlignment="Left" Margin="256,140,0,0"
               Name="image2" Stretch="Fill" VerticalAlignment="Top" Width="640" />
        <Button Content="Click to add Bench" Height="28" HorizontalAlignment="Left"
                Margin="50,140,0,0" Name="buttonBench" Click="benchClicked"
                VerticalAlignment="Top" Width="127" />
        <Image Height="72" HorizontalAlignment="Left" Margin="50,174,0,0"
               Name="bench1" Stretch="Fill" VerticalAlignment="Top" Width="127"
               Source="/SeniorProject_Jonathan_DeWolf;component/bench1.jpg" />
        <Image Height="156" HorizontalAlignment="Left" Margin="464,379,0,0"
               Name="image1" Stretch="Fill" VerticalAlignment="Top" Width="230" />
        <Button Content="Click to add Lamp" Height="28" HorizontalAlignment="Left"
                Margin="50,441,0,0" Name="buttonLamp" Click="lampClicked"
                VerticalAlignment="Top" Width="127" />
        <Image Height="60" HorizontalAlignment="Left" Margin="74,475,0,0"
               Name="lamp3" Stretch="Fill" VerticalAlignment="Top" Width="75"
               Source="desklamp.jpg"/>
        <Image Height="80" HorizontalAlignment="Left" Margin="700,363,0,0"
               Name="image3" Stretch="Fill" VerticalAlignment="Top" Width="96" />
        <Image Height="480" HorizontalAlignment="Left" Margin="256,140,0,0"
        Name="imageTest" Stretch="Fill" VerticalAlignment="Top" Width="640" />
    </Grid>
</Window>
```

Bibliography

[1] *Welcome to a world through Glass*, http://www.google.com/glass/start/what-it-does/.

[2] James Fahey, *Augmented Reality: Towards a Better Working Definition*, http://www.jamesfahey.com/2013/01/16/augmented-reality-towards-a-better-definition.

[3] *MSDN Library*, http://msdn.microsoft.com/en-US/library.

[4] Jason Tanz, *Kinect Hackers are Changing the Future of Robotics*, http://www.wired.com/magazine/2011/06/mf_kinect/all/1.

[5] Helen C. Miles, Serban R. Pop, Simon J. Watt, Gavin P. Lawrence, and Nigel W. John, *A review of virtual environments for training in ball sports*, http://www.sciencedirect.com/science/article/pii/S0097849312000957.

[6] Robert T. Held, Ankit Gupta, Brian Curless, and Maneesh Agrawala, *3D Puppetry: A Kinect-based Interface for 3D Animation*, http://vis.berkeley.edu/papers/3dpuppet/3DPuppet-small.pdf.

[7] Ray Walters, *Kinect-powered augmented reality sandbow is better than what you grew up with*, http://www.geek.com/news/kinect-powered-augmented-reality-sandbox-1488215/.

[8] *Bitshift Operators*, http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/BitOp/bitshift.html.

[9] http://answers.ros.org/question/9231/converting-from-rgb-to-depth/.

[10] http://www.garyweeks.com/pedestal_tables.htm/.

[11] Andrew J. Davison, Walterio W. Mayol, and David W. Murray, *Real-Time Localisation and Mapping With Wearable Active Vision*, http://www.robots.ox.ac.uk/ActiveVision/.