ELECTRICAL AND COMPUTER ENGINEERING

DIGITAL SIGNAL PROCESSING

FINAL PROJECT REPORT

REAL TIME BACKGROUND SUBSTRACTION USING KINECT V1'S COLOR AND INFRARED CAMERAS ON WINDOWS WITH C#

TEAM: SOLIS

Ruperto Solis
Net ID: rs2362

Professor
John E. Ball

ABSTRACT

This project guides you through the steps of Installing and configuring the Microsoft Kinect V1 for software implementation of Digital Image Processing, specifically on the simulation of the green screen technique to remove the background based on position of undesired objects relative to the spatial position of the desired ones; this includes, the specification of the Kinect hardware, the selection of the operative system and programming language based on the desired application, description of system requirements for the use of the Kinect, and the steps for the recognition and implementation of libraries to control the Kinect hardware. The document guides you through the process of data acquisition, conversion of data into images, image processing techniques such as scaling, repositioning, aerial perspective of intensity images, and masking for the removal of background regular images.

Table of Contents

I. INTRODUCTION

The world is moving towards a digital world and Image processing specialists are needed more than never before. Virtual reality, augmented reality, phones with higher sensing capabilities and autonomous vehicles are around the corner and this is just the beginning of the new technology that is about to come. This project is the first step of the pyramid of infinite possibilities and infinite worlds, understanding how we can perceive our environment in different ways and see things with different eyes is something that will take us there. This document is going to help you to understand how to determine the location of objects using infrared emissions and how to use that information to remove undesired objects from images for future analysis on different disciplines. Kinect V1 is not but the pioneer on objects detection on civil applications, keep reading if you want to learn how to use it on your own projects and take advantage of the multiple possibilities that come with it.

II. BACKGROUND

2.1 HISTORY OF GREEN SCREENS

Georges Melies was a French Illusionist and film director famous for leading many technical and narrative developments in the earlies days of cinema. In his 1898 film "Four heads are better than one", Melies employed composing technique in which combined different shots and elements into one image; He would use a piece of glass with black paint to black out pieces of the frames, then rewind the film, black out other parts, and expose the previously covered parts. This visual trick is the rudimentary beginning of what is known today as the green screen composing [1].

The use of green screen composing is a very common and essential practice in today filmography, most of the backgrounds we see in today's movies are recorded or artificially generated separately and then composed with the rest of the frames. The technique is called Green Screen Composing because the majority of time, a green screen is used behind the people and forward objects; This screen will be easily identified by a computer in order to remove it and replace it with a different background. Other colors can be used; however green is the most common because it can be easily separated from the reddish tones of the human skin, in fact, blue is another widely-used color for this kind of techniques.

2.2 VISUAL PERCEPTION IN NEW TECHNOLOGY

It is well known that technology is evolving towards completely autonomous systems, virtual reality, and augmented reality, and to successfully implement these new sciences in every day's life it is necessary to develop devices and techniques capable of understanding our environment; See what we see and understand what we understand to take decisions the same way we do.

One of the most important perception systems for the human beings is the visual perception, we take most of our decisions based on what we see and where there is conflict between other sensorial systems and the vision, we tend to trust on our vision of a first instance. While modern computer vision systems have impressive performance in some specific domains, there are no systems able to recognize instances of visual categories or understand the contents of visual scenes with anywhere near the generality and robustness of human perception [2].

Since green screens cannot be use in a daily basis in order to separate what we want on an image from the rest, new techniques in the fields of Image processing and Machine Learning are being developed and continuously subjected to improvement in order to obtain the same results. It is difficult to separate people and forward objects on an image

without a colored background because in order to do that, the algorithms used to perform this action would need to understand what needs to be removed, based on a number of characteristics that are not so easily identified as unique on a 2D picture.

2.3 KINECT HARDWARE AND FUNCTIONALITY

In November of 2010, Microsoft introduced the first-generation Kinect which is a motion sensing input device created for XBOX 360. The hardware is a box with several microphones and digital audio sources, one Color Camera, an Infrared Path Projector and an Infrared camera [3]. It enables users to control and interact with their console/computer without the need for a game controller, through a natural user interface using gestures and spoken commands.

The Kinect is a depth camera. Normal cameras collect the light that bounces off the objects in front of them. They turn this light into an image that resembles what we see with our own eyes. The Kinect in the other hand, records the distance of the objects that are placed in front of it. It uses infrared light to create an Image (a depth image) that captures not what the objects look like, but where they are in space [5].

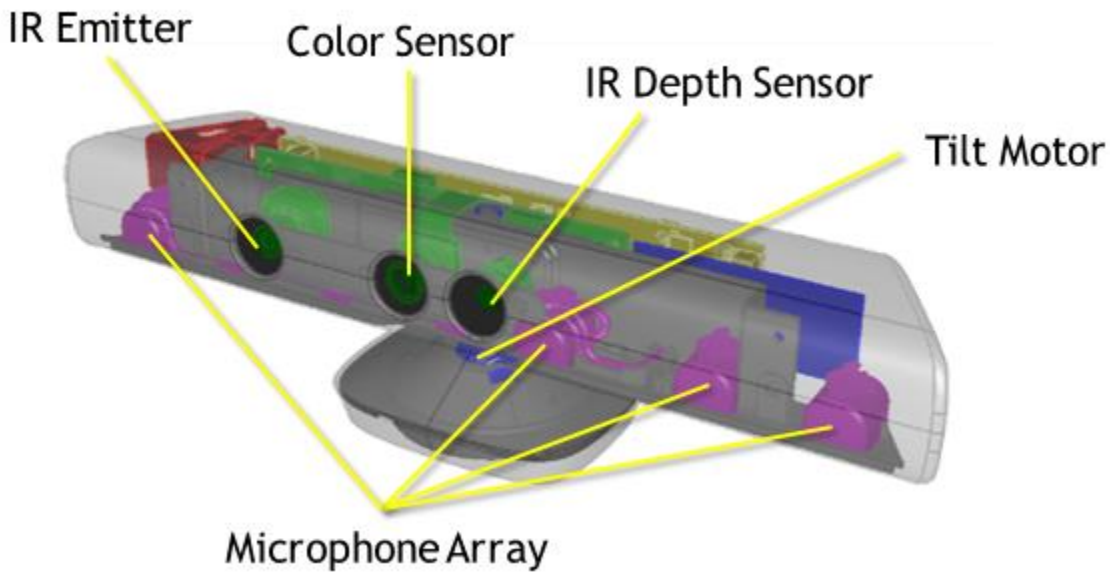

Fig. 1 Kinect Components

2.4 DIGITAL IMAGE PROCESSING

Digital image processing is the use of computer algorithms to perform image processing on digital images. As a subcategory or field of digital signal processing, digital image processing has many advantages over analog image processing. It allows a much wider range of algorithms to be applied to the input data and can avoid problems such as the build-up of noise and signal distortion during processing. Since images are defined over two dimensions (perhaps more) digital image processing may be modeled in the form of multidimensional systems.

Interest in digital image processing methods stems from two principal application areas: improvement of pictorial information for human interpretation; and processing of image data for storage, transmission, and representation for autonomous machine perception.

An image may be defined as a two-dimensional function, (x, y), where x and y are spatial (plane) coordinates, and the amplitude of "f" at any pair of coordinates (x, y) is called the intensity or gray level of the image at that point. When x, y, and the intensity values of "f" are all finite, discrete quantities, we call the image a digital image. The field of digital image processing refers to processing digital images by means of a digital computer. Note that a digital image is composed of a finite number of elements, each of which has a value. These elements are called picture elements, image elements, pels, and pixels. Pixel is the term used most widely to denote the elements of a digital image [6].

2.5 KINECT ARTISTS

Robert Hodgin

Robert Hodgin is an artist and programmer working in San Francisco. He was one of the founders of the Barbarian Group, a digital marketing and design agency in New York. Hodgin has been a prominent member of the creative coding community since before that community had a name, creating groundbreaking work in Flash, Processing, and the C++ framework, Cinder. He is known for creating beautiful visual experiences using simulations of natural forces and environments. His work tends to have a high degree of visual polish that makes sophisticated use of advanced features of graphical programming techniques such as OpenGL and GLSL shaders. He has produced visuals to accompany the live performances of such well-known musicians as Aphex Twin, Peter Gabriel, and Zoe Keating. Soon after the release of the Kinect, Hodgin released Body Dysmorphia (http://roberthodgin. com/body-dysmorphia/), an interactive application that used the depth data from the Kinect to distort the user's body interactively in real time to make it appear fat and bloated or thin and drawn [5].

Fig. 2 Body Dysmorphia by Robert Hodgin.

blablabLAB

blablabLAB is an art and design collective based in Barcelona. They describe themselves as "a structure for transdisciplinary collaboration. It imagines strategies and creates tools to make society face its complex reality (urban, technological, alienated, hyper-consumerist). It works without preset formats nor media and following an extropianist philosophy, approaching the knowledge generation, property and diffusion of it, very close to the DIY principles." Their work explores the impact of technology on public life, from urban space to food. In January 2011, they produced an installation in Barcelona called "Be Your Own Souvenir." The installation offered passersby the opportunity to have their bodies scanned and then receive small plastic figurines 3D printed from the scans on the spot. "Be Your Own Souvenir" won a 2011 Prix Arts at the Ars Electronica festival. blablabLAB expresses a highly pragmatic and hybrid approach to using technology for cultural work, exploring a wide variety of platforms and programming languages [5].

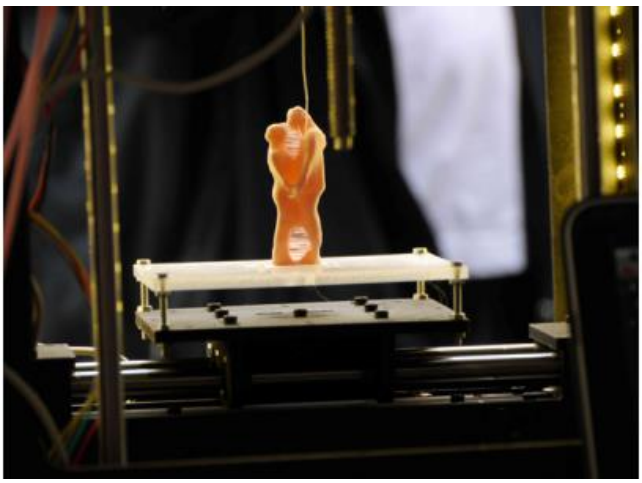

Fig. 3 A print produced by blablabLAB's "Be Your Own Souvenir" project based on a scan of two lovers holding hands.

III. METHODOLOGY

3.1 HARDWARE DEFINITION

Inside the sensor case, a Kinect for Windows sensor contains:

- An RGB camera that stores three channel data in a 1280x960 resolution. This makes capturing a color image possible.
- An infrared (IR) emitter and an IR depth sensor. The emitter emits infrared light beams and the depth sensor reads the IR beams reflected to the sensor. The reflected beams are converted into depth information measuring the distance between an object and the sensor. This makes capturing a depth image possible.
- A multi-array microphone, which contains four microphones for capturing sound. Because there are four microphones, it is possible to record audio as well as find the location of the sound source and the direction of the audio wave.
- A 3-axis accelerometer configured for a 2G range, where G is the acceleration due to gravity. It is possible to use the accelerometer to determine the current orientation of the Kinect.

Kinect	Array Specifications
Viewing angle	43° vertical by 57° horizontal field of view
Vertical tilt range	±27°
Frame rate (depth and color stream)	30 frames per second (FPS)
Audio format	16-kHz, 24-bit mono pulse code modulation (PCM)
Audio input characteristics	A four-microphone array with 24-bit analog-to-digital converter (ADC) and Kinect-resident signal processing including acoustic echo cancellation and noise suppression
Accelerometer characteristics	A 2G/4G/8G accelerometer configured for the 2G range, with a 1° accuracy upper limit.

Table 1 Kinect Specifications

3.2 OPERATIVE SYSTEM AND PROGRAMMING LANGUAGE

There are many open source Applications and of plenty documentation on how to set up the Kinect with different libraries for various programming languages, however, since Microsoft developed it, they already created their own drivers for windows and libraries that can be used with C++ and C#.

Considering the advantages offered by the official API, the existing knowledge regarding the programming languages, and the scope of the current investigation are the reasons for selecting WINDOWS as the operative system for the project. Once the operative system has been selected, it comes to define whether to create to use C++ or C#. Although C++ can be a much more powerful in terms of efficiency, which is always very welcome when dealing with processing of digital signals, for this first step I decided to select C# due to its high level architecture, this with the purpose of spending less time on the configuration and acquisition stages and spend more time on the actual implementation of the logic behind the background removal.

Kinect for Windows drivers support the use of multiple sensors on a single computer. The API includes functions that enumerate the sensors so that you can determine how many Kinects are connected to the computer, get the name of a particular sensor, and set streaming characteristics for each sensor.

3.3 SYSTEM REQUIREMENTS

Supported Operating System
Windows 7, Windows 8, Windows 8.1, Windows Embedded Standard 7

- Hardware Requirements
  Your computer must have the following minimum capabilities:
  - 32-bit (x86) or 64-bit (x64) processor.
  - Dual-core 2.66-GHz or faster processor.
  - Dedicated USB 2.0 bus.
  - 2 GB RAM.
  - A Microsoft Kinect for Windows sensor.

- Software requirements
  - Visual Studio 2010, or Visual Studio 2012. The free Express editions can be downloaded from Microsoft Visual Studio 2010 Express or Microsoft Visual Studio 2012 Express.
  - .NET Framework 4 (installed with Visual Studio 2010), or .NET Framework 4.5 (installed with Visual Studio 2012).

3.4 HARDWARE INSTALLATION AND CONFIGURATION

1. Install KINECT SDK (Starter Development Kit) V1.8 [For instructions, refer to: https://www.microsoft.com/en-us/download/details.aspx?id=40278]:

    a. Make sure the Kinect sensor is not plugged into any of the USB ports on the computer.

    b. Close Visual Studio. You must close Visual Studio before installing the SDK and then restart it after installation to pick up environment variables that the SDK requires.

    c. Once the SDK has completed installing successfully, ensure the Kinect sensor is plugged into an external power source and then plug the Kinect sensor into the PC's USB port. The drivers will load automatically (There is a Kinect V1 for windows, it already comes with the appropriate USB and power connectors, If you are using the Kinect v1 for the XBOX 360, you will need an adapter to get the USB connection and the power source).

2. Download the Kinect for Windows Developer Toolkit, which contains source code samples, tools, and other valuable development resources that simplify developing Kinect for Windows applications [Refer to https://www.microsoft.com/en-us/download/details.aspx?id=40276].

3. Too use the libraries in C#:

    Create a new C# project in Visual studio and link the Microsoft.Kinect.dll dynamic library (Go to Project/Add Reference/Assembly Tab/Framework/Browse Button), the library is typically located at C:\Program Files\Microsoft SDKs\Kinect\v1.8\Assemblies.

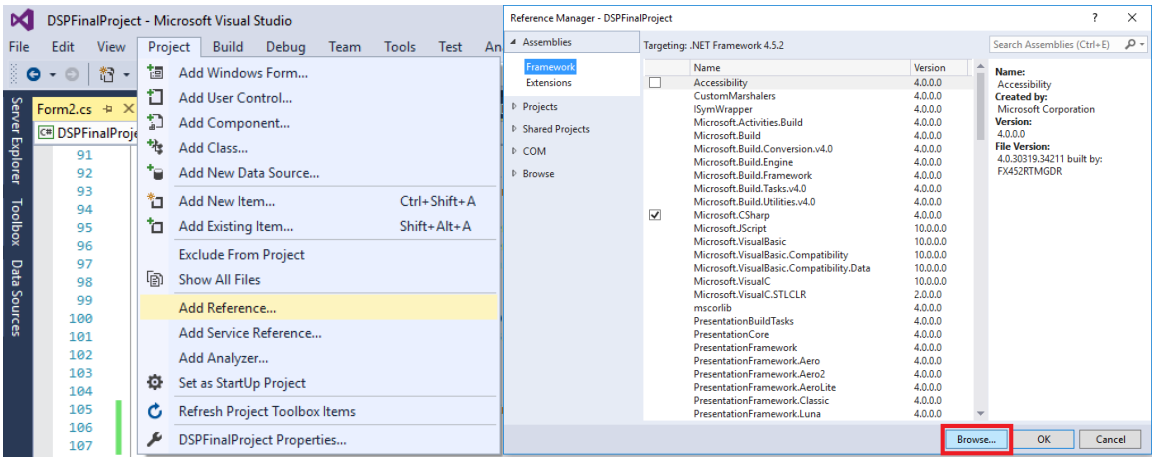

Fig. 4 Linking Kinect Library

3.5 KINECT CAMERAS

The Kinect V 1 has several sensors and operation modes that can be used for various application, however the 2 particular ones we are interested ones are the camera and the infrared cameras:

- The color camera can be configured to operate on RGB with a resolution of 640x480 at 30fps or 1280x960 at 12fps.
- The infrared camera can operate with a resolution of 640x480 at 30fps.

For the purpose of this project we need use a common resolution between the infrared and the color camera, hence 640x480 at 30fps on both.

- By using the C# libraries, the color camera will provide pixel data in RGB; 1 stream for every color per pixel with an intensity between 0 and 255.
- The infrared camera can be set to provide a depth image which contains intensity data for each pixel between minDepth and maxDepth (Then value of this minDepth and maxDepth needs to be read using software, both can be store on integer variables).

3.6 GENERAL METHODOLOGY

The overall process to obtain the final isolated image were:

1. Created a class called colorCamera which reads the color data from the color camera and transforms it into a 640x480 RGB Bitmap object that can be read by the rest of the program.
2. Created a class called depthCamera which reads the depth data from the Infrared camera and transforms it into a 640x480 RGB Bitmap object that can be read by the rest of the program.
3. Using timers, acquire the original image from both cameras and frame by frame display them into a pictureBox object.
4. Both cameras have different depth of view creating different scale and position of the objects, that need to be corrected by using the Scaling and Reposition Digital Image Processing techniques.
5. The scaled and repositioned color and depth images are Bitmap objects as well and need to be displayed in pictureBox objects for visual comparison.
6. A projection algorithm was created to display the intensity of the infrared data from an Aerial perspective, this provides a visualization of the relative position of the objects as the infrared beams of the Infrared Projector hit them. This information is useful for calibration of the dimensional filtering and distinguish between objects that are in the back from the objects that we want to display.
7. Once we have the calibration parameters we can apply the dimensional filter which will remove any undesired object from the depth Image, creating what I call “The Image Mask”.
8. For the final step, we just need to replace the remaining pixels in the Image Mask with Pixels from the color camera.

3.7 DATA ACQUISITION AND CONVERTION INTO BITMAP

1) Color Camera class:

For Code Details refer to APENDIX A

a) Create a sensor object using the KinectSensor class Microsoft.Kinect library.
b) Set the format resolution to 640x480 with 30fps.
c) Create the Bitmap object with the same resolution and format.
d) Initialize the sensor object.
e) Enable and event to notify when an image frame is ready to be read.
f) On the event handler copy the image data which is contained in a ColorImageFrame object into a byte array; The length of the array is 640 x 480 x 4 (4 bytes per pixel, XRGB where the X is not used).
g) Fill the bitmap object with the data from the array (See code it's a bit complex).
h) Return the Bitmap Object.

2) Depth Camera class:

For Code Details refer to APENDIX B

a) Create a sensor object using the KinectSensor class Microsoft.Kinect library.
b) Set the format resolution to 640x480 with 30fps and DepthImage format.
c) Create the Bitmap object with the same resolution but with a color Image format (RGB).
d) Initialize the sensor object.
e) Enable and event to notify when a frame is ready to be read.
f) On the event handler copy the depth data which is contained in a DepthImageFrame object into a DepthImagePixel object array; The length of the array is 640 x 480 (1 per pixel).
g) Read the maximum Intensity and the minimum Intensity of the current frame.
h) Create a depth factor that will be used to scale the intensity value into a value from 0 to 255 (We need the intensity value to be between 0 and 255 because it will be used to create a color Image latter on):

$$h = \frac{255}{maxIntensity - minInstensity}$$

i) With a loop go thru every single pixel in the current frame and read the intensity of every pixel using the conversion factor "h".
j) Create a Byte array to store the color equivalent data from depth; the length of the array is 640x480x4 to create a XRGB structure where all values are 0 except by the green, which will contain the color equivalent value of the intensity.
k) Fill the bitmap object with the data from the array just like we did with the color camera.
l) Return the Bitmap Object.

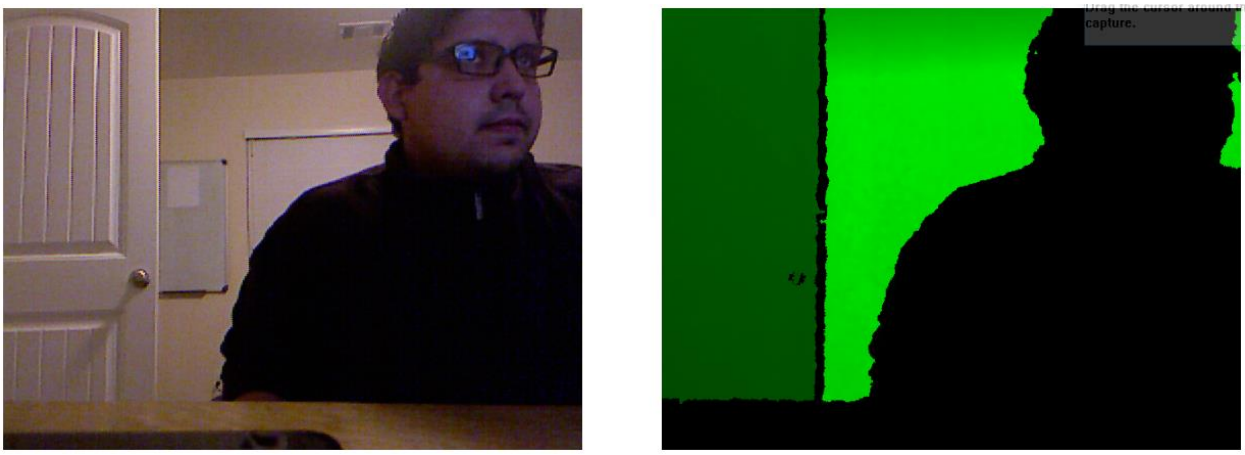

Fig. 5 Original Images from color camera and depth camera respectively

3.8 SCALING AND REPOSITIONING OF THE IMAGES

As discussed before, the field of view of the color and the infrared camera are not the same and this require the application of Digital Image Processing Techniques to make them match.

Fig. 6 Discrepancy on field of view

After some experimentation using GIMP 2 (Image Editor), I got to the following conclusions:

- To compensate the difference in field of view, the depth image need to be scaled from 640x480 to 576x432 (0.9 scaling).
- After scaling, the left most 8 columns of the depth image are not usable (they are pure black) creating a new image with resolution of 568x432.
- The color image has a larger field of view showing information that is not visible in the depth image (the first 38 columns on the left, the las 40 columns on the right, the first 43 rows on the top and the last 5 rows in the bottom) which need to be removed.
- The final resolution of both images is 568x432.

Code:

```
//Specifications of new resolution
      newSize = new Size(576, 432);
      DepthRectangle = new Rectangle(8, 0, 568, 432);
      ColorRectangle = new Rectangle(38,43,568,432);
//Scale Depth Image to 90% and remove first 8 columns
      if (OriginalDepthImage != null)
      {
            temporaryBitmap = new Bitmap(OriginalDepthImage, newSize);
            TrimmedDepthImage = temporaryBitmap.Clone(DepthRectangle,
            OriginalDepthImage.PixelFormat);
            temporaryBitmap.Dispose();
      }
      //TrimColorImage
      TrimmedColorImage = OriginalColorImage.Clone(ColorRectangle,
      OriginalColorImage.PixelFormat);
```

3.9 PROJECTION ALGORITHM

The resulting depth image was created using a green scale from the intensity information provided by the depth camera, hence the intensity of the green color image indicates how far objects are. I created an algorithm that allow us to see how far objects are by converting the depth Image into an aerial view, where the x axis the entire plane of the depth image and the y axis is the intensity of the colors from the depth image plane.

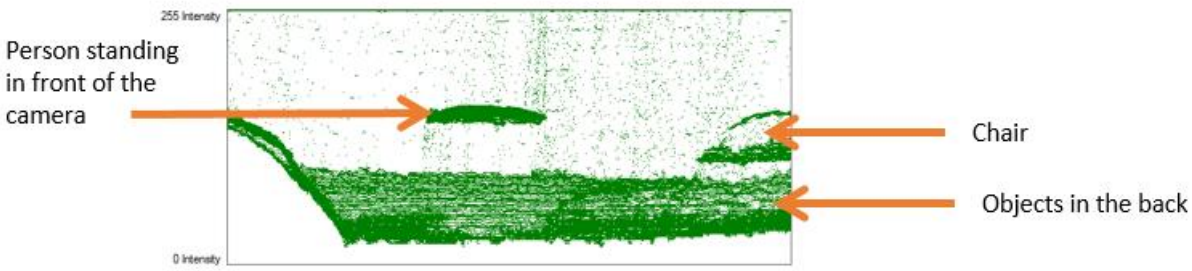

Fig. 7 Aerial view of depth image vs intensity

Code:

```
WhiteProjection = new Bitmap(568, 256);
WhiteMatrix = new byte[568, 256];
WhiteMatrix[i, j] = 255;
WhiteProjection.SetPixel(i, j, Color.White);
DepthContentMatrix = WhiteMatrix;
DepthContentImage = WhiteProjection.Clone(ProjectionRectangle,
WhiteProjection.PixelFormat);
for (int i = 0; i < 432; i++)
{
      for (int j = 0; j < 568; j++)
      {
            Color currentColor = TrimmedDepthImage.GetPixel(j, i);
            byte currentIntensity = currentColor.G;
            DepthContentMatrix[j, currentIntensity] = 0;
            DepthContentImage.SetPixel(j, currentIntensity, Color.Green);
      }
}
      this.depthContentBox.Image = DepthContentImage;
```

Fig. 8 Image used to create the aerial view

3.10 MASK IMAGE AND FILTERING

The next step is to remove from the depth Image all the undesired objects, thanks to the projection algorithm we can visually analyze where the objects are, therefore, define a dimensional range where desired objects need to be. As an example, from figure 7 we can see that the undesired objects have an intensity of approximate 150 or less, we also want to get rid of things that are to close.

Steps to create the mask image:

1) Define the max intensity and the min intensity of the objects you want to visualize, for this example, based on the information from figure 7, we can set a min intensity of 5 and a max intensity of 150.
2) Create an algorithm that goes through every pixel of the depth image and for all those pixels which intensity is less than 5 or greater than 150, set them to 255 on the 3 RGB (white).
3) Set to 0 (black) any pixel which intensity is: 150<= intensity <= 5.
4) Finally, create a new image and replace any black pixel from the depth image with a pixel of the same position from the color image.

IV. RESULTS

Figure 9 shows the resultant mask and final Image where most of the undesired content was removed, keeping only those objects within the intensity interval that we define in the user interface.

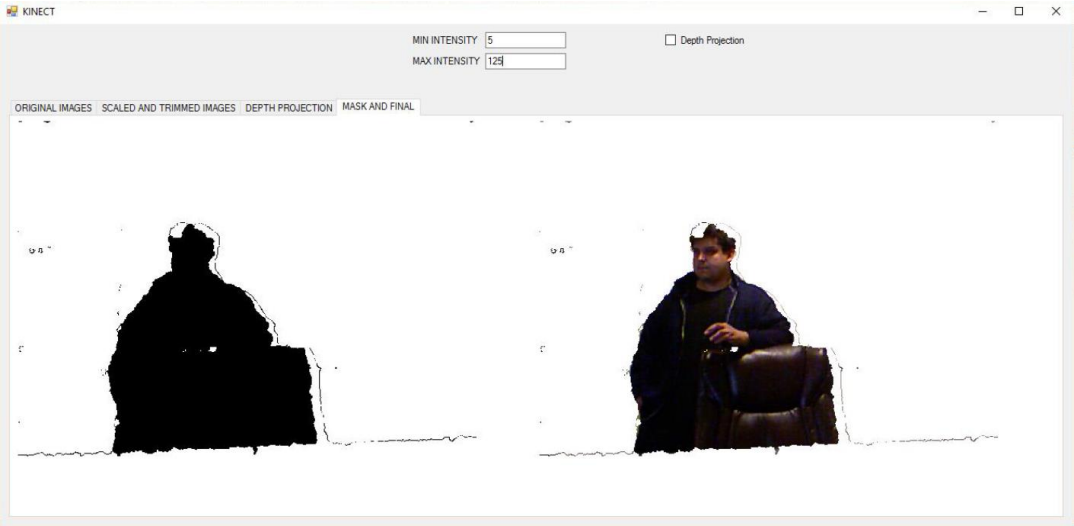

Fig. 9 Mask Image (left) and Final Image (Right)

Qualitative Analysis of the resulting Image:

- The final image is not an exact representation of the desired objects, as we can see from figure 9, some parts are noticeably missing around the body and the edges are not sharply defined. This is a hardware limitation from the Kinect V1, the depth image is created by using a projection of infrared beams that are reflected from the object and are picked up by the infrared camera, however these beams are part of a mesh pattern that have gaps in between the dots which in combination with the size of the dots, create a poor resolution of captured objects. In addition, some parts of the body such as the hair are not solid enough to provide a good reflecting body for the camera.
- We can appreciate some noise on the stop band of the filter, this is a result of the scaling of the depth image, some experiments where performed to the image without scaling and the noise is much less.

Qualitative Analysis of the resulting Image:

- An small algorithm was created to read the frames per second of the image at different points of the processing of the image, although the rate provided by the camera is 30fps on both spectrums, the processing time is such that the resulting image is displayed at approximate 2fps, This is caused by a combination of factors such as the architecture of the programming language and the inefficiency of processing the image and displaying at different stages.

V. CONCLUSIONS AND FUTURE WORK

The main reason for selecting this project was to understand how the Kinect works in order to use it or a similar technology in the development of more complex project in the future.

The results are not ideal but are satisfactory taking into consideration the hardware limitation and the fact that the software was not created to be efficient but to visually show the different stages involved in the implementation of a project like this.

Many things where learned including the acknowledgement of those that need to be learned in order to successfully implement a project with more complex specifications.

I will definitely implement this knowledge in future projects with better hardware (there is already a version 2 of the Kinect for XBOX one with higher specs) and more efficient software, I want to keep learning about image processing to improve the quality of images through software.

VI. REFERENCES

[1] filmmakeriq.com, 'Hollywood's History of faking it | The evolution of green screen compositing'. [Online]. Available: http://filmmakeriq.com/lessons/hollywoods-history-of-faking-it-the-evolution-of-greenscreen-compositing. [Accessed: Sept-25-2016].

[2] Melanie Mitchell, Visual Understanding, Santa Fe Institute Bulletin, Spring 2008.

[3] i-programmer.info, 'All About Kinect'. [Online]. Available: http://www.i-programmer.info/babbages-bag/2003-kinect-the-technology-.html. [Accessed: Sept-25-2016].

[4] Microsoft.com, 'Kinect for Windows Programming Guide'. [Online]. Available: https://msdn.microsoft.com/en-us/library/hh855348.aspx. [Accessed: Sept-22-2016].

[5] Making Things See, 1st ed., O'Reilly Media, Inc., Sebastopol, CA, 2012.

[6] Digital Image Processing, 3d ed., Pearson, Upper Saddle River, NJ, 2007.

[7] Digital Signal Processing Using MATLAB, 4th Edition, CENGAGE Learning, 2012.

APENDIX A – Color Camera Class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Microsoft.Kinect; //For kinect
using System.Diagnostics; //For Debuging
using System.Drawing; //For Bitmap Class
using System.Drawing.Imaging; //For pixel format
using System.IO; //For stream Class
using System.Runtime.InteropServices;
using System.Windows.Forms; //To use PictureBox

namespace DSPFinalProject
{
    class colorCamera
    {
        private KinectSensor Sensor;
        private byte[] cameraFrame;
        private Bitmap colorBitmap;
        public Bitmap CurrentImage;
        PixelFormat formato;
        public colorCamera()
        {
            //Initialize Kinect Sensor
            //Debug.WriteLine(KinectSensor.KinectSensors.Count);
            foreach(var potentialSensor in KinectSensor.KinectSensors){
                this.Sensor = potentialSensor;
                break;
            }
            //Sensor Settings
            if (this.Sensor != null)
            {
                //Color format Settings

this.Sensor.ColorStream.Enable(ColorImageFormat.RgbResolution640x480Fps30);
                //Set the Size of the byte array that is going to store the
frame
                this.cameraFrame = new
byte[this.Sensor.ColorStream.FramePixelDataLength];
                //Bitmap thaty will be used to store the Image
                formato = PixelFormat.Format32bppRgb;
                this.colorBitmap = new Bitmap(640, 480, formato);
            }
            //Start Kinect
            if (this.Sensor != null)
            {
                this.Sensor.Start();
            }
            //Enable Event to notify when frame is ready
            if (this.Sensor != null)
            {
```

```
                this.Sensor.ColorFrameReady += this.SensorColorFrameReady;
            }

        }
        public bool changeAngle(int angle)
        {
            if(this.Sensor != null)
            {
                //Move Camera
                this.Sensor.ElevationAngle = angle;
                return true;
            }
            else
            {
                return false;
            }
        }
        public Bitmap getImage()
        {
            return this.colorBitmap;
        }

        //Event handler trigers when frame is ready
        private void SensorColorFrameReady(object sender,
ColorImageFrameReadyEventArgs e)
        {
            using (ColorImageFrame colorFrame = e.OpenColorImageFrame())
            {
                if (colorFrame != null)
                {
                    //Copy frame into array
                    colorFrame.CopyPixelDataTo(this.cameraFrame); //the
length of the Byte Array is width * height * 4, each pixel has 4 bytes due to
the color content
                    BitmapData bmData = this.colorBitmap.LockBits(new
Rectangle(0,0,colorFrame.Width,colorFrame.Height),ImageLockMode.ReadWrite,
formato);
                    IntPtr pointer = bmData.Scan0;

Marshal.Copy(this.cameraFrame,0,pointer,this.cameraFrame.Length);
                    this.colorBitmap.UnlockBits(bmData);
                    this.CurrentImage = this.colorBitmap;
                }
            }
        }

    }
}
```

APENDIX B – Depth Camera Class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Microsoft.Kinect; //For kinect
using System.Diagnostics; //For Debuging
using System.Drawing; //For Bitmap Class
using System.Drawing.Imaging; //For pixel format
using System.IO; //For stream Class
using System.Runtime.InteropServices;
using System.Windows.Forms; //To use PictureBox

namespace DSPFinalProject
{
    class depthCamera
    {
        private KinectSensor Sensor;
        private DepthImagePixel[] depthPixelsArray;
        private byte[] colorPixels;
        private int bytesPerDepthPixel;
        private Bitmap colorBitmap;
        public Bitmap currentImage;
        PixelFormat formato;
        public depthCamera()
        {
            //Initialize Kinect Sensor
            //Debug.WriteLine(KinectSensor.KinectSensors.Count);
            foreach (var potentialSensor in KinectSensor.KinectSensors)
            {
                this.Sensor = potentialSensor;
                break;
            }
            //Sensor Settings
            if (this.Sensor != null)
            {
                //Color format Settings

this.Sensor.DepthStream.Enable(DepthImageFormat.Resolution640x480Fps30);
                //Size of the Depth pixels array
                    this.depthPixelsArray = new
DepthImagePixel[this.Sensor.DepthStream.FramePixelDataLength];
                //Number of bytes required to represent a single depth pixel
(4)
                    this.bytesPerDepthPixel = 4;
                //Size of colorPixels equivalent of depth pixels (4 times
longer)
                    this.colorPixels = new
Byte[this.Sensor.DepthStream.FramePixelDataLength*bytesPerDepthPixel];
            }
            //Start Kinect
            if (this.Sensor != null)
            {
                this.Sensor.Start();
            }
```

```
            //Enable Event to notify when frame is ready
            if (this.Sensor != null)
            {
                formato = PixelFormat.Format32bppRgb;
                this.colorBitmap = new Bitmap(640, 480, formato);
                this.Sensor.DepthFrameReady += this.SensorDepthFrameReady;
            }
        }

        public Bitmap getImage()
        {
            return this.colorBitmap;
        }

        //Event handler trigers when frame is ready
        private void SensorDepthFrameReady(object sender,
DepthImageFrameReadyEventArgs e)
        {
            using (DepthImageFrame depthFrame = e.OpenDepthImageFrame())
            {
                if (depthFrame != null)
                {
                    //Copy frame into  array of "Depth" Pixels

depthFrame.CopyDepthImagePixelDataTo(this.depthPixelsArray);
                    //Convert the Depth data into RGB and store in Bitmap
                        //Get min and Max Depth for Current Frame
                            int minDepth = depthFrame.MinDepth;
                            int maxDepth = depthFrame.MaxDepth;
                            //I will be using two bytes, the BG byte and the
GR byte to go from blue to red
                            int minColor = 0;
                            int maxColor = 255;
                            double depthColorFactor = ((double)(maxColor -
minColor) / (double)(maxDepth - minDepth));//To scale the depth on color
scale
                            Debug.WriteLine(depthColorFactor);
                            int colorPixelIndex = 0;
                            //Go depthPixel by depthPixel get intensiry of
the depth pixel and translate it into RGB
                            for(int i=0;i<this.depthPixelsArray.Length;i++)
                            {
                                //Get depth for current Depthpixel
                                short depth = this.depthPixelsArray[i].Depth;
                                //If the Depth is between minDepthand
MaxDepth then keep its value, set it black otherwise
                                if (depth > maxDepth)
                                {
                                    depth = (short)maxDepth;
                                }
                                else {
                                    if (depth < minDepth) { depth =
(short)minDepth; }
                                }
                                Byte color=Convert.ToByte((depth-
minDepth)*depthColorFactor);
```

```
                                this.colorPixels[colorPixelIndex++] = 0;
//Red starting at index=1 (0 is unused hence ignored)
                                this.colorPixels[colorPixelIndex++] = color;
//Green
                                this.colorPixels[colorPixelIndex++] = 0;
//Blue
                                ++colorPixelIndex;//Skip the unused pixel (at
0,4,8, etc)
                    }
                    BitmapData bmData = this.colorBitmap.LockBits(new
Rectangle(0, 0, depthFrame.Width, depthFrame.Height),
ImageLockMode.ReadWrite, formato);
                    IntPtr pointer = bmData.Scan0;
                    Marshal.Copy(this.colorPixels, 0, pointer,
this.colorPixels.Length);
                    this.colorBitmap.UnlockBits(bmData);
                    this.currentImage = this.colorBitmap;
                }
            }
        }
    }
}
```