A MOTION CAPTURE FRAMEWORK TO ENCOURAGE CORRECT EXECUTION OF SPORT EXERCISES

by Suraj Ho - 1504223
s.k.ho@student.vu.nl

Master thesis presented to the Faculty of Sciences
at the VU University Amsterdam

Supervisor
Anton Eliens

Second reader
Natalia Silvis - Cividjian

Abstract

This thesis will describe the design and development of an exercise monitoring system framework using the Microsoft Kinect v1 and v2. The purpose of this framework was to make it easier to create reference models, especially for domain experts such as trainers and coaches. These models could then be used to monitor users whether they use the proper technique during exercising and help them improve those techniques. This in contrast to similar systems that focus on increasing adherence to exercising. To be able to monitor users, the system should be able to determine when the user does something wrong and what he or she did wrong. During the design and development process I have also compared the accuracy between the Kinect v1 and the v2. Based on those results I have described whether the accuracy is sufficient to determine the technique used during an exercise.

Table of Contents

Preface iii
1 Introduction 1
1.1 Problem statement 1
1.2 Motivation 1
1.3 Agenda 1
2 Motion tracking 3
2.1 Systems 3
2.1.1 Optical systems 3
2.1.2 Non-optical systems 4
3 Related work 7
3.1 Healthcare 7
3.1.1 Jintronix 7
3.1.2 Doctor Kinetic 8
3.1.3 VirtualRehab 8
3.1.4 Reflexion Health 8
3.2 Sports 8
3.2.1 Your Shape: Fitness Evolved 8
3.2.2 Nike+ Kinect Training 9
3.3 Summary 9
4 An overview of the Kinect 10
4.1 The Kinect v1 10
4.1.1 The hardware 11
4.1.2 The software – The Kinect for Windows SDK 15
4.2 The Kinect v2 30
4.2.1 The hardware 30
4.2.2 The software 32
5 Developing the prototype 33
5.1 CrossFit 33
5.1.1 Why CrossFit 34
5.2 Design 34
5.2.1 The hardware setup 35
5.2.2 The application 36

5.3 Analyzing movements 40
5.3.1 Gestures 40
5.3.2 The three movements 43
5.3.3 Recording the movements 44
5.3.4 Breaking down the movements 45
5.4 Implementing the movements 47
5.4.1 Kinect v1 47
5.4.2 Kinect v2 51
5.5 Results 55
6 Evaluation and discussion 57
6.1 Inaccurate data 57
6.2 The difficulty of creating gestures 58
6.3 Future work 59
7 Conclusion 61
7.1 Research questions answered 61
7.2 Reflection 62
8 Acknowledgement 64
Bibliography 65
Appendix A - SkeletonRecorder2 69
MainWindow.xaml 69
MainWindow.xaml.cs 72
Appendix B - BodyBasics-WPF 81
MainWindow.xaml 81
MainWindow.xaml.cs 83
Appendix C - MyOwnVGBTest2 88
MainWindow.xaml 88
MainWindow.xaml.cs 90
KinectBodyView.cs 95
GestureDectector.cs 103
GestureResultView.cs 108

Preface

Ever since I was five years old I have always practiced some sport several times a week. There was this short stint with karate and when that did not work out after a few months I started doing gymnastics and korfball for the rest of my elementary school period. When I started high school, being already 180 cm at that time, I started playing basketball. A sport I practiced and played competitively till the age of 30. And a sport I, if I say so myself, quite excelled in. By that time I was unable to find new challenges in basketball and my cousin introduced me to CrossFit. A sport consisting of (group) workouts, or WOD (Workout Of the Day) as they call it, consisting of a random selection of basic physical movements ranging from gymnastics to Olympic weightlifting. It also contains a competitive element because every workout has some goal that all the practitioners can try to achieve. Setting the fastest time, the heaviest weight, the largest amount of repetitions, etc. I approached the first few workouts with the same mentality as with basketball. Just keep pushing as hard as you can and ignore fatigue.

What I did not take into account was that not every part of my body was used to doing the movements, with or without the weights, which we had to do during CrossFit WODs. After the third week my arm muscles, upper and lower arms, were swollen and I could not straighten my arms completely anymore because the muscles were too tight. What normally happens during any strength workout, the muscles break down after being strained more than they can handle. This is a normal process. After the breakdown, the muscles will heal again during a period of rest and by having access to enough proteins, and they will get stronger to be prepared for such a heavy strain again. What happened in my case was that a larger part than normal of my muscles broke down. The amount of “trash” that was generated by this process was too much for my body to carry off efficiently which caused it to accumulate in my arms. Luckily this sounds a lot worse than it actually was. All I needed to do to make it better was a week’s rest and keep the blood flowing. What I learned from this experience was that my body was not used to the movements used in CrossFit and by just using strength, or in my case the lack thereof, can hurt you. By learning how to move properly, a lot of the exercises can be done more efficiently, will make you rely less on strength and helps preventing injuries.

When I saw the Microsoft Kinect sensor for the first time I was pretty thrilled by the possibilities. But it was not until I saw how Nike used the Kinect sensor with Nike+ Kinect Training[1] that I got really excited by it and thought that such technology could possibly benefit me to improve my technique for some of the exercises during CrossFit. Not having to ask the coach every time to look whether I am doing it right or not and what I am doing wrong and how I should correct it, but to be able to practice it at home as well. This concept could also be extended to healthcare for physiotherapy. And thus was my motivation to start this technical and creative exploration to see whether this concept could be made reality ignited.

[1] http://www.nike.com/us/en_us/c/training/nike-plus-kinect-training

1 Introduction

In this first chapter I will give a brief description of my problem statement and my motivation to use the Kinect sensor to take on this problem. I will conclude this chapter by stating three research questions around which I will conduct this project and a short overview of the rest of this thesis.

1.1 Problem statement

We all know that exercising is good for your health. Unfortunately we do not get enough exercise in this day and age. And those of us that do exercise often do not pay enough attention to applying the proper technique when exercising. This often happens because we never learned the proper technique and even if we are aware of this lack of proper technique, we do not always go to a trainer or coach and ask them to teach us. The reasons for this can range from the costs of gaining access to a proper trainer or coach, to the distance we might need to travel to get there and back home again. Furthermore, the time a trainer or coach would spend with you means he or she cannot spend that time with someone else. Meaning that a trainer or coach cannot always be there to correct your mistakes. This while using proper technique while exercising is very important because not doing so might lead to injuries, which in turn leaves us unable to exercise, which in turn is not a very healthy lifestyle.

1.2 Motivation

A solution to this would be a technology that would take the role of a trainer or coach and help you during your exercises. Motion tracking technology makes this possible where it is not only used in entertainment but in professional sports and healthcare as well. Unfortunately this technology is often too expensive for private use and too large to use at home. With the introduction of the Microsoft Kinect we have access to a single, inexpensive sensor that is able to give real-time feedback on what someone is doing in front of it.

1.3 Agenda

In this thesis I will describe the development of an unfinished prototype using the Microsoft Kinect. This prototype would be able to track users while they perform exercises and notify them when they made a mistake, what the mistake was and how to correct it in real-time. During this project I will answer three research questions:

- How can motion tracking be applied as a cost effective solution to sports and healthcare?
- How should motion tracking data be analyzed to provide recommendations for posture correction in real-time?
- What are the essential elements for a framework of motion and posture correction for CrossFit exercises?

In chapter 2 I will start with discussing the different types of systems that are used for motion tracking, how they differ from one another, their pros and cons and how they are currently applied to healthcare and sports. I will give an overview in chapter 3 of other applications and systems that employ the Kinect sensor to help people during exercises in both healthcare and sports. Chapter 4 gives an in-depth description of the Kinect v1 of both the hardware and the software. This chapter also contains a short

overview of the recently released Kinect v2 of both its hardware and software as well and how they are improved upon compared to the Kinect v1. I will discuss the development process of the prototype and why I chose CrossFit for testing in chapter 5. The chapter concludes with the results of that process. These results and the problems I encountered during this project will be discussed in chapter 6. In this chapter I summarize all the problems I encountered during this project and how they could be solved by discussing possible improvements to the system. The thesis will conclude with chapter 7 where I will come back to the research questions and discuss the extent to which they are answered. During this project I have used and adapted three software applications. The code of these applications can be found in the appendices.

2 Motion tracking

Motion capture is the process or technique of recording patterns of movement digitally. Having started as a photogrammetric analysis tool in biomechanics research in the 1970s and 80s, this technique is now also being used in the military, entertainment, sports, medical applications, computer vision validation and robotics[2]. Motion capture is best known for its use in the film and video games industry where the performance of human actors is captured and used to animate digital character models. Although the technique is commonly called motion capture, it is also known as motion tracking, which is the term I will use in the rest of this thesis since the main focus of my application will literally be the tracking of human movement as opposed to the recording of human movement.

2.1 Systems

Most motion tracking systems use sensors attached to the performer in order to identify the position or angles of the joints they are attached to. Motion tracking systems can be divided in two general categories, optical systems and non-optical systems. The optical systems use one or more cameras to track the movements performed by the actor, patient or athlete. These optical systems can be further divided in two sub-categories, those using markers and the markerless systems.

2.1.1 Optical systems

The optical systems are probably the most well-known of the motion tracking systems, especially those that use markers to determine the position of the person in 3D space.

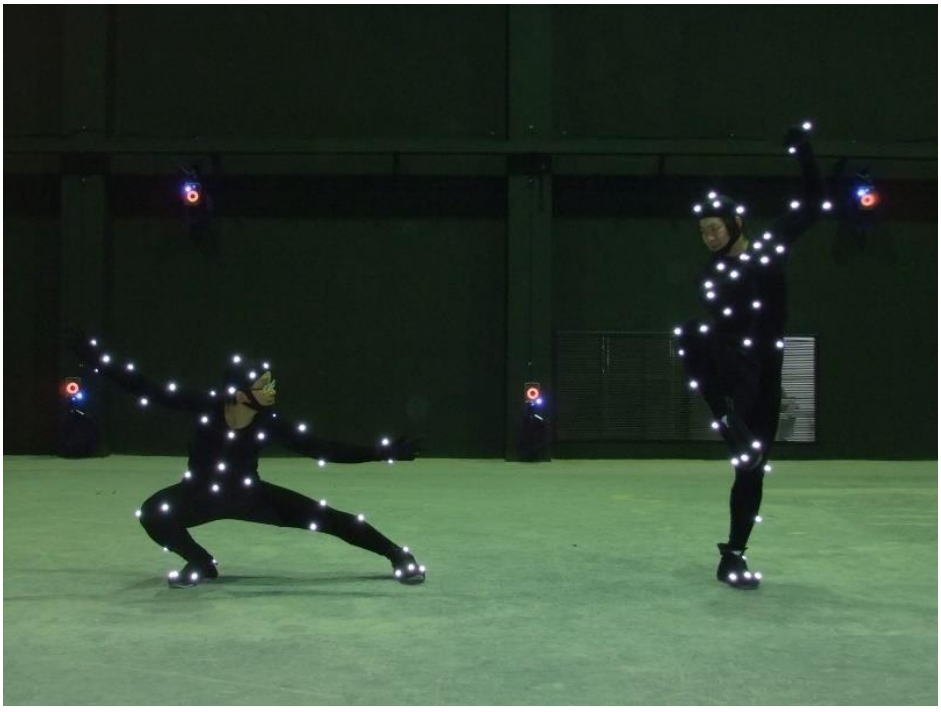

Figure 1 - Two men wearing suits with markers[3]

[2] http://en.wikipedia.org/wiki/Motion_capture
[3] http://kanewinchester.blogspot.nl/2013/01/ba5-contextual-studies-how-has-motion.html

What is probably less well-known is that there are different types of markers, passive, active (2 variants) and photosensitive. You have the passive markers, which are little balls that are slightly larger than a Ping-Pong ball made up of a retro reflective material. Thanks to this material the cameras can be adjusted in such a way that they are only able to see these bright reflective markers and nothing else. Because all the balls look initially the same to the cameras, they need to be calibrated first using an object with markers attached at known positions.

This is less of an issue with active markers. Active markers are a little more sophisticated and come in two variants. Both variants use markers that emit light of which the power in each marker can be controlled. With one variant that control is used to influence the brightness of the markers and with the other variant the amplitude of each marker, which is called time modulation. By either controlling the brightness or the amplitude of the markers, they can each be assigned a unique ID. Both the passive as the active marker techniques require a setup with a large array of high speed cameras and a low light setting. The reason for this is that every marker has to be in view of at least two cameras in order to determine its position in 3D space and is has to be able to track fast movements in order to prevent motion blur, which can prevent the correct tracking of markers. The low light setting is required in order to make it easier for the cameras to identify the reflective or light emitting markers.

The last of the optical systems using markers do not actually use physical markers, but rather photosensitive marker tags to decode the optical signals. A good example of this is the Microsoft Kinect which projects an invisible infrared pattern to recover the depth information of the scene for motion acquisition. In contrast to the other marker based system, this one does not need a low light setting in order for the markers to be the brightest objects in the scene, nor does it need high speed cameras to nullify motion blur.

Then there are also the markerless systems, which differ from their marker using counterparts in that they use algorithms to identify the objects in the scene, often from a video feed, which need to be tracked. This is done by analyzing high resolution images captured by the cameras. Such systems are often used to track objects such as aircraft and missiles and often consist of three subsystems; the optical imaging system, the mechanical tracking platform and the tracking computer. All three of these subsystems are often customized for their intended purpose, either for tracking persons or tracking satellites, but not both. This makes the market of each of such tracking systems very small and hard to commercialize.

2.1.2 Non-optical systems

One obvious disadvantage with optical motion tracking systems is that the motions that are being tracked have to be performed within the view of the camera(s), limiting the size of the area in which the performance can take place. This drawback is less of an issue with non-optical systems, which do not rely on sight but are able to receive their data wirelessly. Within the non-optical systems, three different technologies are used, inertial systems, mechanical motion and magnetic systems.

Inertial systems use miniature inertial sensors to track motion data. These sensors use gyroscopes and/or accelerometers to track the full six degrees of freedom (forward/backward, up/down, left/right,

pitch, yaw and roll) of the joints they are attached to. The Wii controller is in fact a simple inertial sensor, but for professional use, a range of suits which can be equipped with inertial sensors are available.

Figure 2 - A "suit" met inertial sensors[4]

Systems using mechanical motion tracking consist of a skeletal-like structure made of metal or plastic, also referred to as exo-skeleton motion tracking. When a performer wears this structure, its movements are measured by the skeleton in real-time and sent to the receiving system.

Figure 3 - Mechanical motion tracking suit[5]

[4] http://zeroc7.jp/products/products/mvn-biomech-4.html
[5] http://realidadevirtualemocap.wikispaces.com/

Finally, magnetic systems have about the same setup as inertial motion systems, except that instead of inertial sensors, magnetic coils are used to measure the six degrees of freedom. This is done by measuring the relative magnetic flux of three orthogonal coils. The relative intensity of the voltage or current of the three coils allows these systems to calculate both range and orientation.

3 Related work

Although motion capture technology is used in a variety of fields such as military, entertainment and robotics, I will limit my scope to healthcare and sports since these are the fields where my application will ultimately be used. In this section I will discuss several applications and research projects that also use the Kinect for either in healthcare or for sports exercises.

The main reason to use motion tracking technology in health care and sports is *"to create efficient, coordinated, and healthy movement"* (Franklin, 2012). In other words, it is to help the user using the correct posture during exercises and decrease the chance of getting injured (Fitzgerald et al., 2007; Zhou & Hu, 2008). Joints in particular are susceptible to injuries when they are being used to and beyond their limit of possible movements. For example, when you execute an exercise off-balance it puts the stress on parts of the body where it should not. Do this often enough and injuries will follow. Hence the importance of knowing the correct postures during exercises (Tseng et al., 2009).

Another reason to use motion tracking technology in health care and sports is to lighten the load off trainers and offer the possibility for personalized workouts to a larger audience. Teaching yourself the correct posture for any exercise or movement is not an easy task, this is normally taught by a trainer, coach, physiotherapist, etc. By a specially programmed application that uses motion tracking technology you would be able to have access to your own personal trainer. In turn, a trainer would not have to be able to be present with every one of his trainees during every one of their sessions (Aminian & Najafi, 2004; Tseng et al., 2009).

Until recently it was fairly expensive to work with motion tracking technology. With optical systems there were not only the costs of several high speed cameras, but also the availability of a controlled environment to record in. And even with the more 'affordable' non-optical systems such as those using inertial sensors, a complete suit would start around the five thousand euros. Since the release of the Microsoft Kinect for Windows, developers have access to a very affordable, 200 euros, motion tracking camera of which only one is needed.

3.1 Healthcare

3.1.1 Jintronix

Jintronix is a virtual rehabilitation platform designed for physical and occupational therapy. It combines common rehab movements, virtual games and the Kinect sensor which is used as a tool for physical rehabilitation.[6] Currently the platform is used to specifically help patients who suffered a stroke with their rehabilitation to improve their motor functions. Jintronix was created to make the rehabilitation of stroke survivors more accessible and engaging while keeping the costs low. An initial study (Archambault, Norouzi, Kairy, Solomon, & Levin, 2014) shows that a small test group found the platform fun and easy to use. Further study is required to measure to improvements of the patients when they use the platform.

[6] http://www.jintronix.com/

On the other hand, the Jintronix is also designed in such a way to make it easier for clinicians to help their patients. The platform records the data gathered from every session of every patient. This data provides performance metrics and multiple means of displaying improvement. Based on this information the clinician can adjust the sessions of the patient accordingly. All of the information is automatically documented.

3.1.2 Doctor Kinetic

Doctor Kinetic is a Dutch company that creates applications for use at the doctors' practice, at home or in the gym. Their application FysioGaming uses the Kinect sensor to track the users during their exercises. The main goal of the application is to make exercising fun and easy for all ages to encourage adherence to the exercises. FysioGaming is an extensive application that contains a wide variety of exercises.[7]

3.1.3 VirtualRehab

VirtualRehab[8] is an application developed by the Virtualware Group[9] and uses the Kinect sensor. With VirtualRehab, the Virtualware Group tries to increase the adherence of patients to their exercises and lower the costs of rehabilitation. The application contains a set of exercises that are to be used for rehabilitation treatments. It also offers the doctors a tool to manage their patients where they can track their progress.

3.1.4 Reflexion Health

The Rehabilitation Measurement Tool from Reflexion Health[10] (Reflexion Health, 2013) is an application that utilizes the Microsoft Kinect for Windows. Although not on the market yet it is presented as prescription software. Which means that instead of selling its software as a service directly to consumers, Reflexion plans to enroll doctors and physical therapists as resellers who would prescribe the program in much the same way caregivers prescribe other medical products.

The software will be able to track the patients during their prescribed exercise plan, which can be customized and reviewed by the therapist. Because the software is able to track the patient with the Kinect for Windows, it can check the quality of the exercise done by the patient and correct the patient when necessary. It will be able to store the performance or the patient during each session with which the therapist and the patient can review the progress.

3.2 Sports

3.2.1 Your Shape: Fitness Evolved

Your Shape: Fitness Evolved is a video game, or rather, exergame[11] developed by Ubisoft. The game was released on November 4, 2010 as a launch title for the Xbox 360's Kinect sensor as a sequel to Your

[7] http://doctorkinetic.nl/
[8] http://www.virtualrehab.info/
[9] http://www.virtualwaregroup.com/
[10] http://www.xconomy.com/san-diego/2012/10/30/reflexion-health-raises-4-5m-to-advance-idea-for-prescribed-software/
[11] http://en.wikipedia.org/wiki/Exergaming

Shape which used a proprietary motion tracking camera. The game uses "player projection" technology to capture the player's shape and form, dynamically incorporate them into the in-game environment, and tailor routines for the player. Compared to its predecessor, this game also featured more emphasis on the use of resistance training, along with exercise programs developed in partnership with the magazines Men's Health and Women's Health. The sequel Your Shape: Fitness Evolved 2012 was released on November 8, 2011 and was essentially an upgraded version of the previous one. Your Shape essentially provides the user a personal in his or her living room. The user is able to put together his or her own workout program and Your Shape will keep track of your progress. The setting of Your Shape leans to the fun side of exergaming where its primary objective is to motivate the user into a more active lifestyle.

3.2.2 Nike+ Kinect Training

Nike+ Kinect Training[12] was released on October 30, 2012 and provides, just like Your Shape, a personal trainer in the living room. The presentation and the way it does this however are completely different. The setting is more serious and gives the impression that Nike+ Kinect Training is for the more serious athlete. The trainers the user can choose from are well known (in the United States), Nike sponsored personal trainers and other Nike sponsored athletes occasionally encourage you as well after the workout. Instead of choosing your own workout, Nike+ Kinect Training requires the user to first do a test workout which consists of a few basic exercises that measures the user' strength, condition and flexibility. Based on how the user performs during this workout and whether the user wants to get stronger, fitter or more toned, a 4 week program is generated by the application that consists of the days of the user' choosing. After these 4 weeks the user does the same test workout to measure his or her progress. Although the presentation is more serious, progress is still measured in points earned during exercises and is calculated into, what Nike calls, Nike Fuel Points[13]. These can synchronized with the user' account on Nike.com and can be compared to the results of the user' friends.

3.3 Summary

As you can see, the Kinect is applied in in healthcare more often than in sports. And the few sports applications I did find were presented more as games than as a serious alternative of having to go to the gym and neither of them measure actual useful data. All of the applications are also more geared towards helping patients and users to make exercising more accessible and more fun in order to motivate them to adhere to the exercises. With this project I tried to find out whether it is also possible to use the Kinect to validate the form of the user during exercising.

[12] http://www.nike.com/us/en_us/c/training/nike-plus-kinect-training
[13] http://www.nike.com/us/en_us/c/nikeplus/nikefuel

4 An overview of the Kinect

In this chapter I will give an in-depth overview of the hardware and software of the Kinect v1 sensor and a short overview of the Kinect v2. With the Kinect v2 overview I will mainly discuss its improvements upon the Kinect v1.

4.1 The Kinect v1

The first time people were shown the Kinect, it was presented under the codename 'Project Natal'[14]. During this presentation game designer Peter Molyneux showed a demo at the Electronic Entertainment Expo (E3) in 2009, where a woman interacted with a virtual character, a boy, through a Natural User Interface (NUI). This meant that without having to use a traditional controller which you would hold in your hands, the woman could interact with the boy as she would do with someone in real life. She could talk and use gestures and the boy would respond accordingly.

This NUI is made possible by the combination of components that are present in the Kinect. These are Infrared (IR) emitter, IR depth sensor, color camera, LED and microphone array.

Because of the advanced hardware offered by the Kinect for only one hundred euros, the Kinect has been used by many developers to research and develop applications that have nothing to do with video games since its launch in November 2010 for the Xbox 360. From a motion controller for user interfaces such as in the movie Minority Report[15], to virtual dressing rooms where the customer can view different clothes on themselves on a dressing mirror sized screen[16], to the detection of conditions such as autism, attention-deficit disorder and obsessive-compulsive disorder[17]. Microsoft never anticipated that its device would be used in so many different ways other than what it was intended for and was reluctant at first to support this. When the Kinect was released Adafruit Industries offered a 3000 dollar reward to the person who could develop an open source driver for the Kinect. Within a week, on November 10th, Héctor Martín released a Linux driver allowed the user to access the color camera and the depth functions[18].

Seeing how popular the Kinect became outside the gaming community, Microsoft released an official non-commercial Software Development Kit (SDK) for Windows on June 16th 2011 and a commercial update in February 2012 together with the Kinect for Windows costing 200 euros. This SDK enabled developers to build applications with C++, C#, or Visual Basic by using Microsoft Visual Studio. Besides the Kinect SDK from Microsoft it is also possible to use either OpenNI or OpenKinect to develop applications for the Kinect. For these frameworks there are also wrappers available to make it possible to program in other programming languages such as Java and ActionScript.

[14] http://www.youtube.com/watch?v=CPIbGnBQcJY
[15] http://www.youtube.com/watch?v=tlLschoMhuE
[16] http://www.youtube.com/watch?v=L_cYKFdP1_0
[17] http://www.mndaily.com/2011/03/10/researchers-%E2%80%98kinect%E2%80%99-data-make-faster-diagnoses
[18] http://www.adafruit.com/blog/2010/11/10/we-have-a-winner-open-kinect-drivers-released-winner-will-use-3k-for-more-hacking-plus-an-additional-2k-goes-to-the-eff/

4.1.1 The hardware

The Kinect consists of several different optical and audio parts that drives the NUI, namely the color camera, the infra-red (IR) emitter and IR depth sensor and the microphone array. The placement of these parts on the Kinect itself can be seen in figure 4.

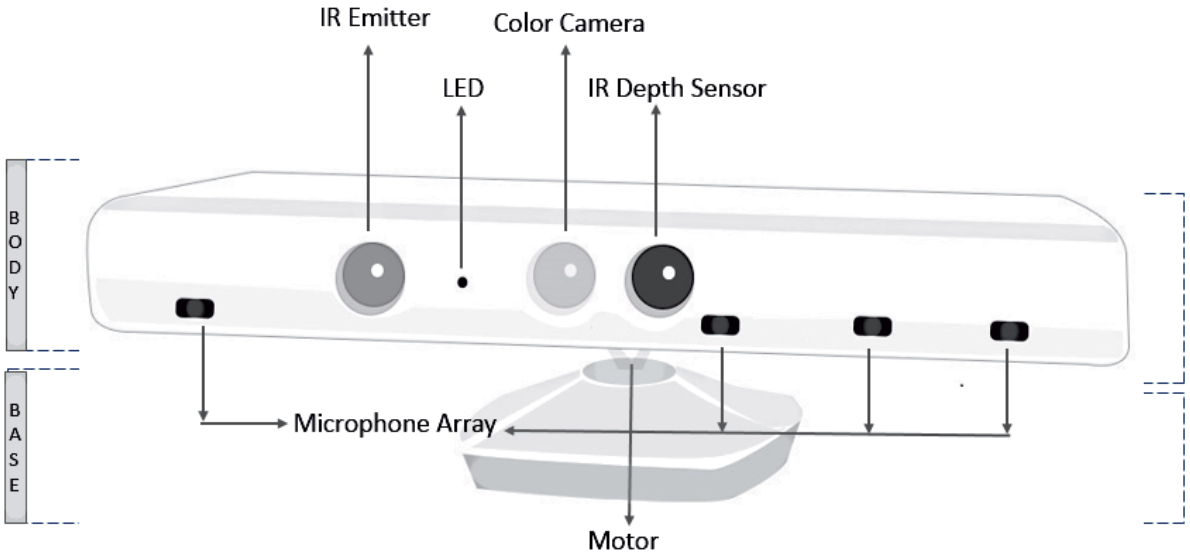

Figure 4 – The placement of all the important parts on the device (Jana, 2012)

The color camera of the Kinect is almost similar to an average webcam. Its main purpose is to detect the red, blue and green (RGB) colors in the frames it captures. The specifications of this camera state that it supports 30 frames per second (FPS) at a resolution of 640 x 480 pixels. It also supports a resolution of 1280 x 960 pixels, but at this resolution it can only capture images at a speed of 12 FPS. The view range of color camera is 43 degrees vertical by 57 degrees horizontal, as illustrated in figure 5.

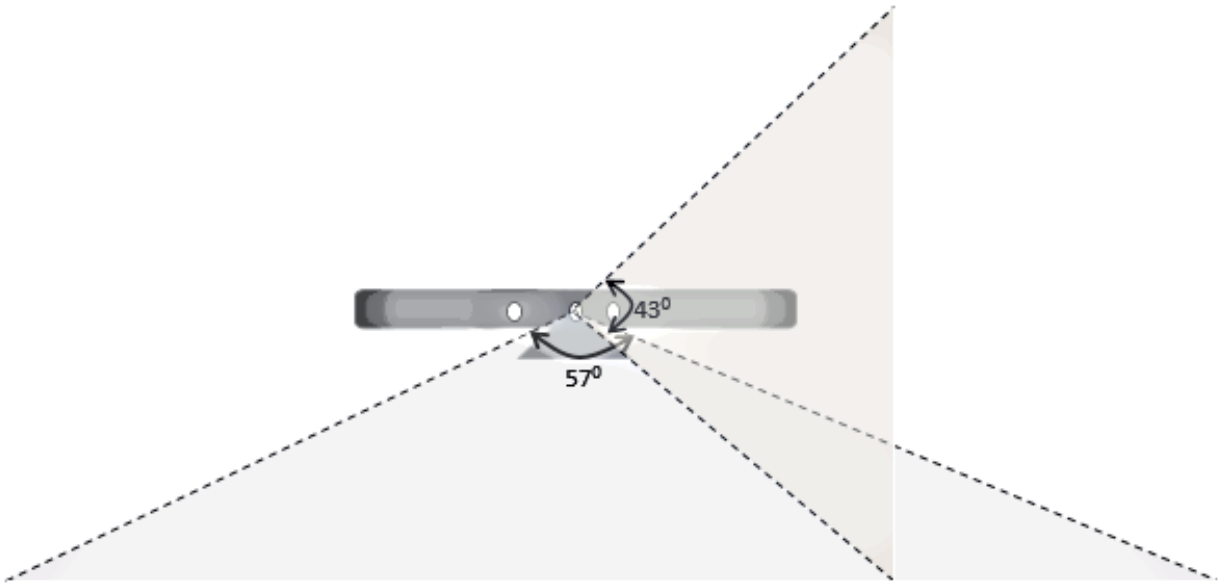

Figure 5 – The angles of the field of view of the Kinect sensor (Jana, 2012)

The IR emitter and IR depth sensor are the parts responsible for capturing the depth data by working in tandem. This is accomplished by letting the IR emitter constantly emit infrared light in a so called 'pseudo-random dot' pattern over everything in its view first, which are invisible to the naked eye. This is because the wavelength of the infrared light is longer than that of the visible light. In order to minimize the noise within the captured data, the wavelength needs to be as consistent as possible. This is achieved by the small fan built into the Kinect, which keeps the temperature of the laser diode at a constant level.

To the naked eye these dots are invisible but the IR depth sensor, which is a monochrome Complimentary Metal-Oxide-Semiconductor (CMOS) sensor, is able to capture them. The depth sensor does this by measuring the distance between itself and the point from which an IR dot is reflected. By doing this for all dots it is possible to construct a 3D space seen from one viewpoint. The resolution of this depth image can be set to 640 x 480 pixels, 320 x 240 pixels or 80 x60 pixels. This technology was not developed by Microsoft itself but by the company PrimeSense. Figure 6 shows a simple diagram of how it works.

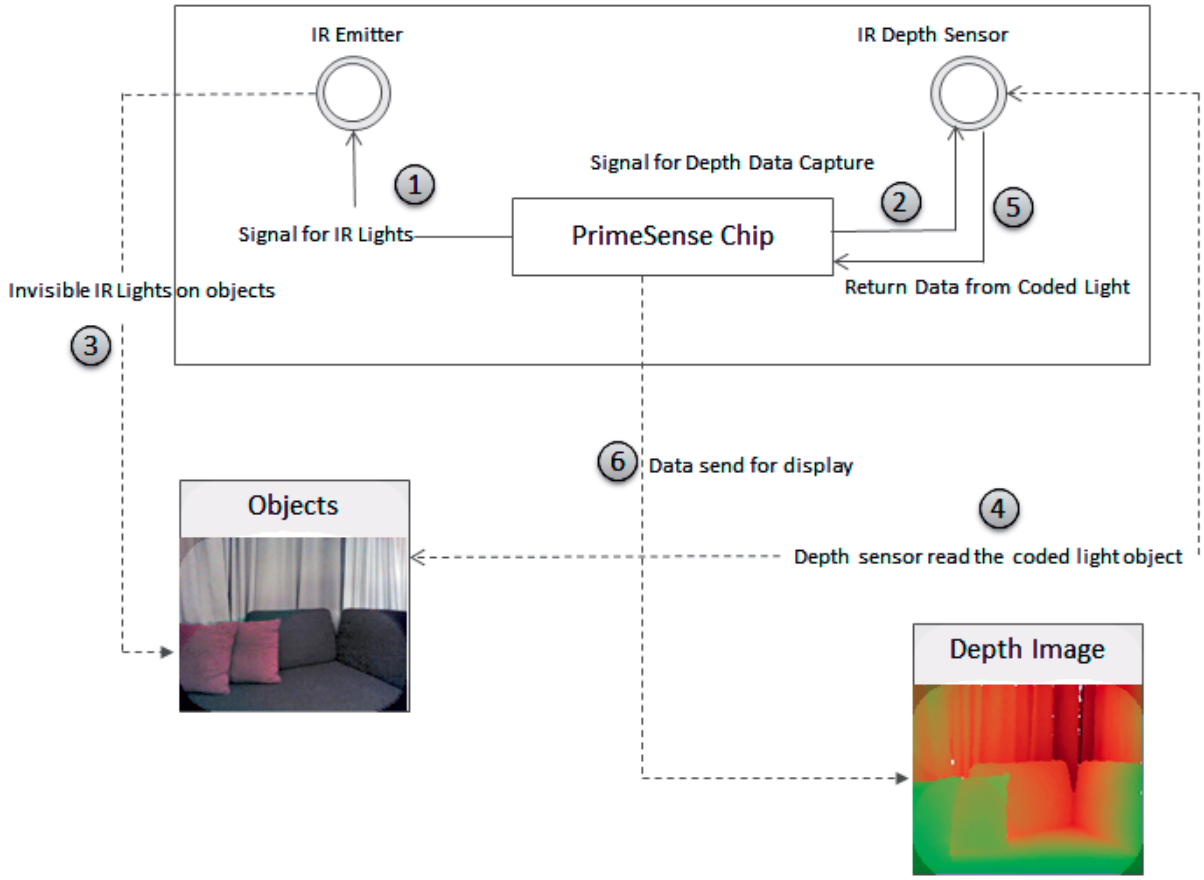

Figure 6 – A diagram of the process of generating a depth image by the PrimeSense chip (Jana, 2012)

As you can see in figure 4 and figure 7, the Kinect uses not one microphone but four, three of which are placed on the right side. The reason for using an array of microphones is not just to capture audio, but to enable the device to also locate the direction of the audio source. Furthermore, the microphone array also enables features such as noise suppression and echo cancellation to capture and recognize voice more effectively.

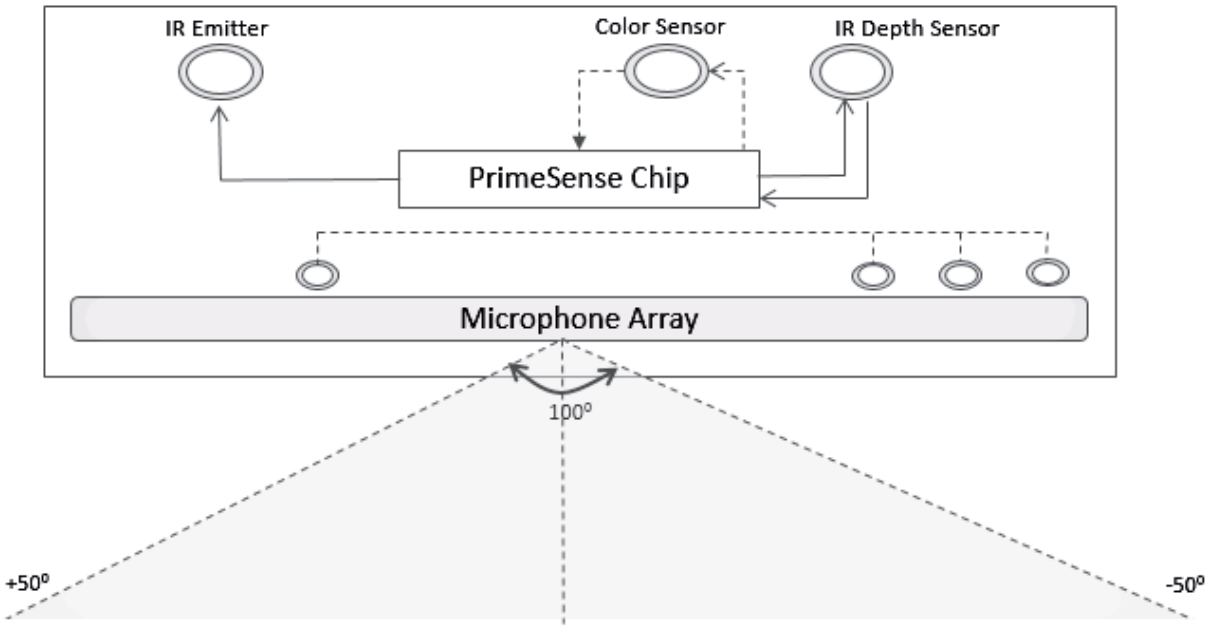

Figure 7 – The area covered by the microphone array for catching sound (Jana, 2012)

Xbox vs. Windows

You might have already wondered what the difference is between the Kinect for Xbox and the Kinect for Windows after having read the beginning of section 3.1, especially since there is a price difference of 100 euros between the two. Both devices are largely similar and the differences are mostly noticeable from a developer's point of view. This is mostly because the Kinect for Xbox was initially developed to accompany the Xbox 360 and add to the gaming experience. It was never meant to be used as a development tool, for which it was also largely used for after it came out. To meet this need of developing applications with the Kinect, Microsoft released the Kinect for Windows and the accompanying SDK.

The first difference between the Xbox and Windows version is the range in which they can track people. Both sensors can track up to 4 meters (or 12 feet) away, but the Xbox sensor cannot track when people are standing closer than 80 cm from it. The Windows sensor has a feature called 'Near Mode' that enables it to track as close as 40 cm.

One minor difference between the Xbox and Windows sensors is that form of the USB connector of the Xbox version does not comply with the USB standard, which can result in the Xbox sensor not being able to fit in every USB port.

But the biggest difference is the fact that the Windows sensor can be used for commercial applications. This basically means that although you can test and develop with both sensors, the finished application will, and may, only work with a Windows sensor.

4.1.2 The software – The Kinect for Windows SDK

4.1.2.1 History

With the SDK for the Kinect for Windows, Microsoft has provided an easy manner for any developer to create their own applications using the Kinect. After the release of the Kinect for Xbox in 2010, the device became very popular with application developers and educational institutions to experiment with. After this became apparent to Microsoft, they released a first beta version of their SDK to support these people in June 2011 and a second beta version in November 2011. Both versions were for non-commercial use only, but in February 2012 a commercial version (v1.0) was released. Starting with v1.0 it was now possible for developers to create applications using the Kinect for Windows that could be deployed for commercial use. Microsoft has eventually released four major updates for their SDK.

1. **V1.5, May 2012** – Introduced Kinect Studio, Near Mode, Seated Tracking and Face Tracking.
2. **V1.6, October 2012** – Introduced a range of minor improvements and support for Windows 8 and Visual Studio 2012.
3. **V1.7, March 2013** – Introduced Kinect Fusion. This essentially turned the Kinect into a 3D scanner and made it possible to create 3D models.
4. **V1.8, September 2013** – Introduced the features to easily remove the background from scenes and access to the Kinect data through the web.

4.1.2.2 Features of the Kinect for Windows SDK

The Kinect for Windows SDK comes with a wide array of features and tools to be used for the development of Kinect enabled applications. A list and short summary of the most notable features and tools are described in this section.

Near Mode

The Near Mode feature is a Kinect for Windows specific feature, which sets it apart from the Kinect for Xbox since it is built into the firmware of the sensor. This features enables the Kinect for Windows to track the human body from a very close range of about 40 centimeters. Near Mode is a setting that can be turned on or off within the code of the application.

Capturing the infrared stream

Capturing images from the infrared stream opens up the possibility to use images in low light conditions. The images captured from this stream are 16-bits per pixel infrared data at a resolution of 640x480. The frames come in with a speed of 30 FPS. Since the infrared image is captured as part of the color image, it is not possible to capture both at the same time.

Tracking human skeleton and joint movements

The tracking of the human body is one of the most interesting parts of the Kinect. Up to 20 joints can be tracked on a single skeleton and up to six persons can be tracked at the same time. Due to resource limitations however, only the joints of 2 skeletons can be returned. If the application only requires the joints of the upper body, it is also possible to track a person in seated mode, which only returns 10

joints. This method saves resources which can either be used elsewhere or left alone to make the application run better on lighter machines.

The audio stream and speech recognition

With the four microphones in the sensor, the Kinect is capable of capturing raw audio, but the SDK also makes it possible to clean up the audio signal by applying features such as noise suppression and echo cancellation. It is also capable of speech recognition for use in voice commands and it can detect the direction and distance of the source of the sound.

Human gesture recognition

Although the SDK has no direct support for an API for human gestures, the availability of skeleton tracking and depth data opens of the option of creating gestures which can be used as input method for any Kinect application.

Tilting the Kinect sensor and using the accelerometer

The SDK gives direct access to the motor of the sensor. With this the developer can change the elevation angle of the sensor. The elevation angle range lies between +27 degrees and -27 degrees. The motor only tilts the sensor vertically. The elevation of the sensor is also used to determine the orientation of the sensor by the accelerometer. The data from the accelerometer can be accessed as well.

Controlling the infrared emitter

The infrared emitter can be turned on or off. This might seem like an insignificant feature, but it is useful when using multiple sensors since the infrared emitters from the different sensors interfere with each other. This feature is also only available when using the Kinect for Windows and not the Kinect for Xbox.

The Kinect for Windows Developer Toolkit

The Kinect for Windows Developer Toolkit is a collection of sample applications and documentation to help the developer to create sophisticated applications quickly.

The Face Tracking SDK

By using the Face Tracking SDK applications can be created to track the positions and orientations of faces. It is also possible to animate brow positions and the shape of the mouth in real time which can be useful for detecting facial expressions for example.

Kinect Studio

Kinect studio is able to record and playback the data stream of the sensor. This is very useful during the testing of applications. With this tool the developer does not have to perform in front of the camera him or herself every time the application needs to be tested. Instead, a recording can be used to playback with the application. The recorded file contains all the data that can be captured by the sensor.

4.1.2.3 How it works in a nutshell

Using the Kinect SDK it is possible to program in the languages Visual Basic (VB) .NET, C# and C++. With VB .NET and C# programmers can use the managed code provided by the SDK which comes in the form of a Dynamic Link Library (DLL) called Microsoft.Kinect.dll. By adding this library to their application the programmers get access to all the features of the Kinect sensor. Programming in C++ means that the programmer can only use the unmanaged code provided by the SDK but it gets the programmer closer to the source code since the C++ accesses the Native Kinect Runtime APIs directly, whereas VB .NET and C# code first needs to be handled by the .NET Kinect Runtime APIs. Figure 8 shows a diagram of how an application interacts with the different layers.

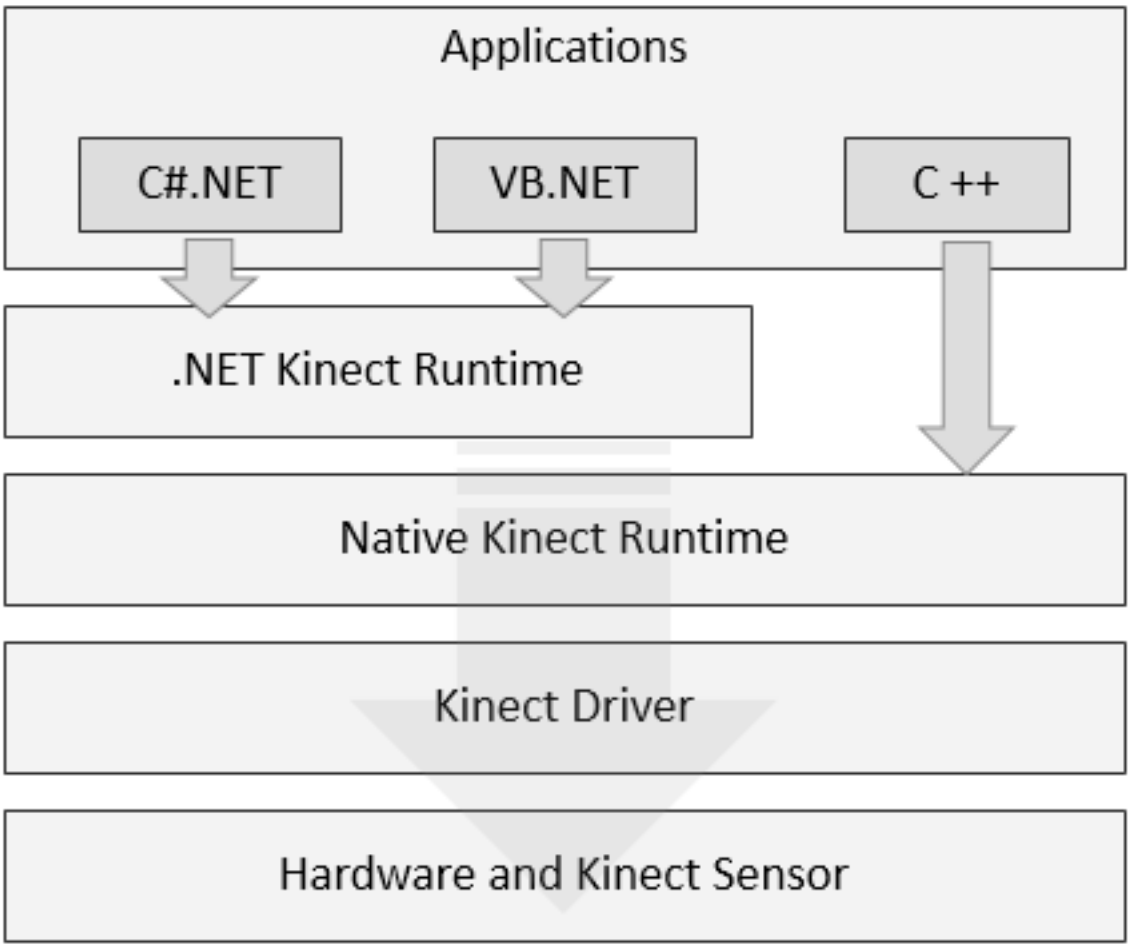

Figure 8 – A diagram showing the interaction between the different layers depending on the programming langauge (Jana, 2012)

When you have built an application for the Kinect sensor it is able to communicate with the sensor using the Kinect for Windows SDK is an interface. Whenever the application needs to access the sensor, it sends an API call to the Kinect driver which controls this access. In figure 9 you can see the layers that lie between the application and the sensor.

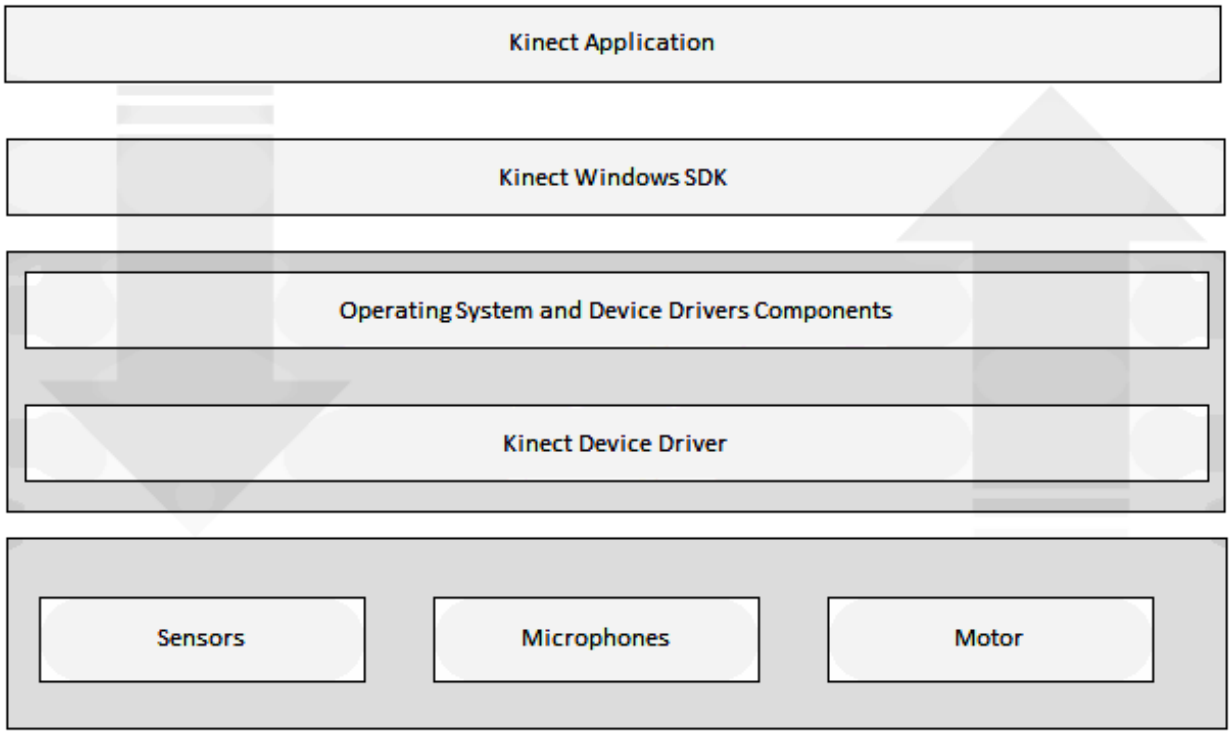

Figure 9 – A diagram showing the layers between the application and the Kinect sensor (Jana, 2012)

Access to the sensor is usually required to obtain sensor data. There are three types of data that are streamed from the sensor to the application. These are the color data stream, the depth data stream and the audio data stream. These data streams are directly available from the sensor through its drivers. The Kinect for Windows SDK makes it possible to access two more data streams, which are the infrared data stream, which gets subtracted from the color data stream, and the accelerometer data from the sensor.

Looking back at figure 7, you might have already guessed that the sensors related to sight are separate from those related to sound. This also holds true for how the libraries in the Kinect for Windows SDK are categorized. These libraries are classified into two categories; libraries concerning the Kinect sensors and those concerning the microphones. Thanks to these libraries the Kinect for Windows SDK gives programmers easy access to the following features: capturing and processing the color image data stream, processing the depth image data stream, capturing the infrared stream, tracking human skeleton and joint movements, human gesture recognition, capturing the audio stream, enabling speech recognition, adjusting the Kinect sensor angle, getting data from the accelerometer and controlling the infrared emitter. I will discuss each of these features separately in the sections below.

4.1.2.3.1 Capturing and processing the color image data stream

The color image date stream is one of the two types of image stream formats that are supported by the SDK, the other one being the depth image stream. Both streams are transferred through different pipelines. All image frames from the image data stream are stored in a buffer in a first in last out fashion. The application makes use of one or more image frames by retrieving them from this buffer.

The FPS speed depends on how fast the application can retrieve image frames from the buffer. This buffer is constantly refreshed with new images frames however. So if the application is not able to retrieve image frames in time, the user will see a stuttering video.

You have already seen that the Kinect camera is capable of capturing 32-bit RGB images at the resolutions 640 x 480 and 1280x960, where it supports 30 FPS at 640 x 480 and 12 FPS at 1280 x 960. Besides the RGB images the SDK also makes it possible to retrieve 16-bit YUV images, raw Bayer images and 16-bit infrared data over the same pipeline. The YUV images are only available with a resolution of 640 x 480 at 15 FPS while the Bayer images offer the same two formats as the RGB images. The infrared data is only available with a resolution of 640 x 480 at 30 FPS. An important note to remember is that only one image type and format can be requested at a time because all of these image frames are transported over the same channel.

Actually retrieving images within an application can be done in two ways, either by using the event model or the polling model. When using the event model you need to subscribe to the specific event handler where you process the incoming frames within your code. When subscribing you must specify the image type and format you would like to receive. Once this setup is complete the Kinect sensor will keep streaming image frames to the application until you stop it.

When you use the polling model you send a request to the Kinect SDK every time you require an image frame. This is handled by the SDK by first looking whether or not there is an open color channel and if no one has subscribed to that channel. If that is the case, it will automatically pick an image frame from that channel. If there are no open channels, a new channel will be created to retrieve the image frame.

4.1.2.3.2 Processing the depth image data stream

The depth data the Kinect sensor returns consists of a 16-bit gray scale image. The Kinect sensor throws a series of algorithms at this image which will return the distance of each pixel in millimeters. The depth data stream can return images with 30 FPS at the resolutions 640 x 480, 320 x 240 and 80 x 60. The image with the depth data is created by the IR emitter and IR depth sensor of the Kinect. The operation of these components can be read in detail in the hardware section but in short, the IR emitter constantly emits infrared light in a pseudorandom dot pattern. The IR sensor reads these dots and with the data being received a depth image is constructed.

The method used to determine the distances in a depth data image is called stereo triangulation. This is an analysis algorithm for computing the 3D position of points in an image. With stereo triangulation, two images from two different views of the same scene are used. The relative depth information is then calculated by comparing these two images. This is about the same way humans are able to perceive depth and why this is difficult for people with only one eye. With the Kinect this method is applied to the image generated by the IR sensor, which is the image that is actually captured, and an ‘invisible’ second image. This second image is actually a pattern of the IR emitter. And since the emitter and the sensor have some distance between them, their viewpoint of the scene is slightly different, making it possible to apply the stereo triangulation algorithm. One limitation to keep in mind concerning this feature is that it only works optimally between 80 centimeter and 400 centimeter, or 2,6 feet and 13,1

feet, not taking near mode into consideration. The Kinect is able to capture beyond 400 centimeter but the quality of the resulting depth image frame suffers dramatically.

After a depth image frame has been created a method called bit shifting is used to calculate the distance from each pixel in the image frame. Since each depth image frame is a 16-bit gray scale image, each pixel contains 16-bit worth of data. Of these 16-bits, the first three are used to represent the player(s) identified by the Kinect. The other 13 bits represent the distance in millimeters.

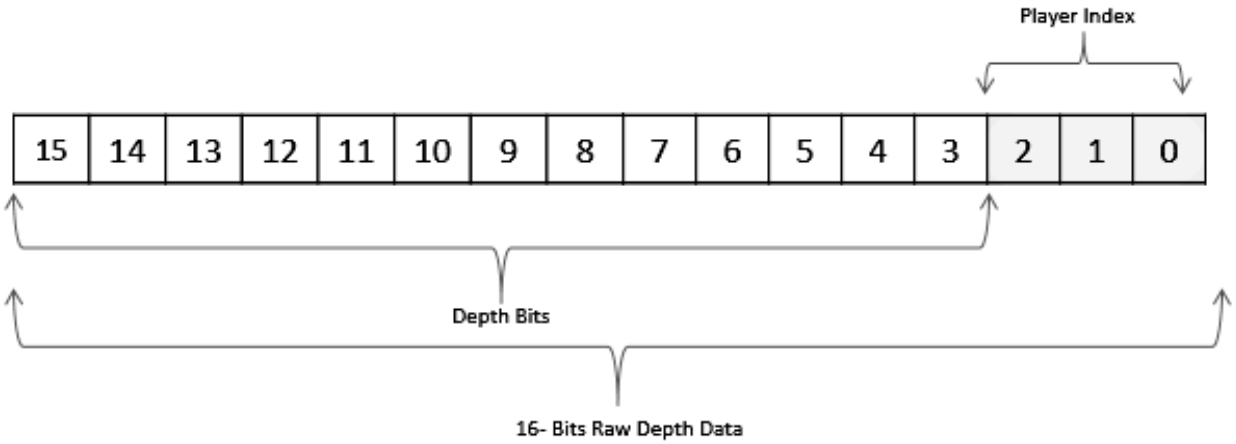

Figure 10 – The first three bits contains a specific player, the player index, in that depth image pixel. The other 13 bits contain the distance between the object in that pixel and the Kinect sensor (Jana, 2012)

By performing a right shift of three bits the distance of that particular pixel can be calculated. Figure 11 shows an example with a pixel that has the value 20952.

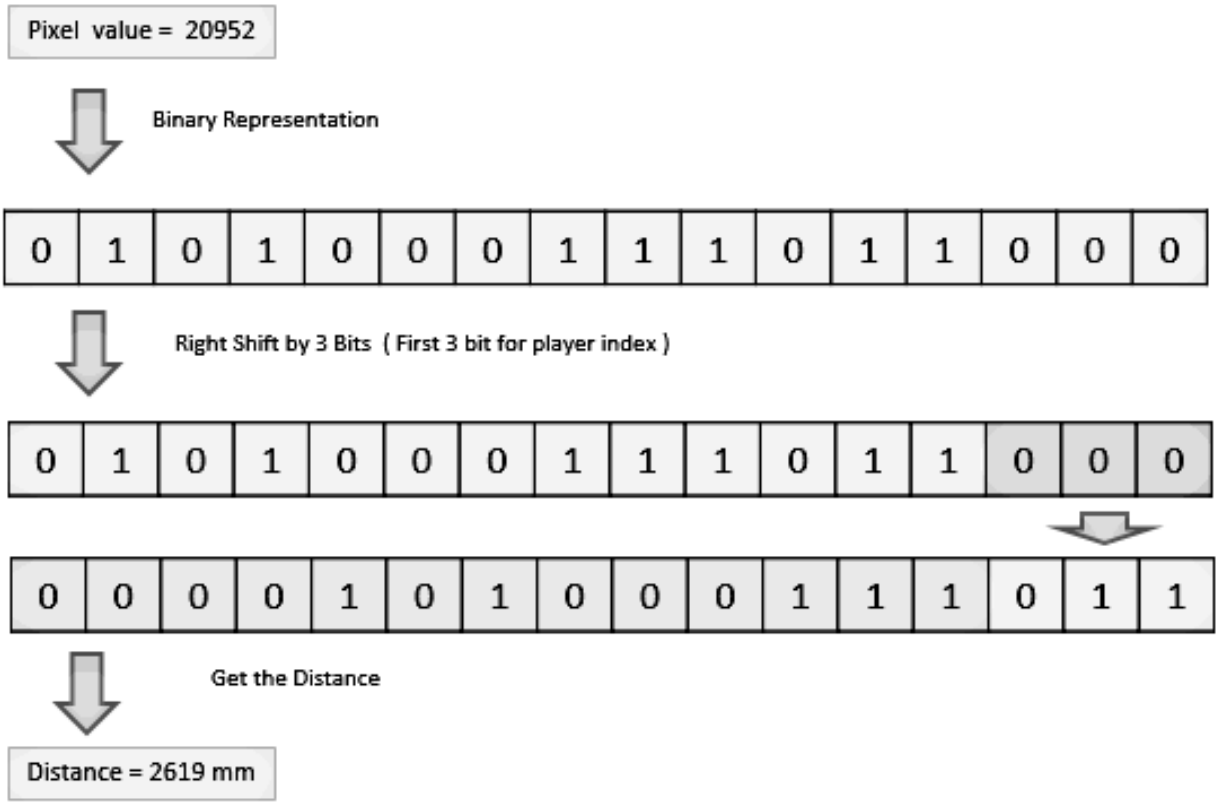

Figure 11 – The process of right shifting three bits (the player index) in order to calculate the distance from a binary representation to millimeters (Jana, 2012)

Besides the distance, the player index can also be calculated with the first three bits. In order for this to work, the skeleton stream needs to be enabled since it contains the player information, which will be discussed further on. The player index can have a value from 0 to 6 where the value 0 states that the sensor does not recognize any player. The player index value is calculated by performing a logical AND operation with the value of the pixel and a constant called the PlayerIndexBitmask. This constant is defined in the class DepthImageStream and always has the value 7. Figure 12 shows an example of such a calculation where the pixel has a value of 8862.

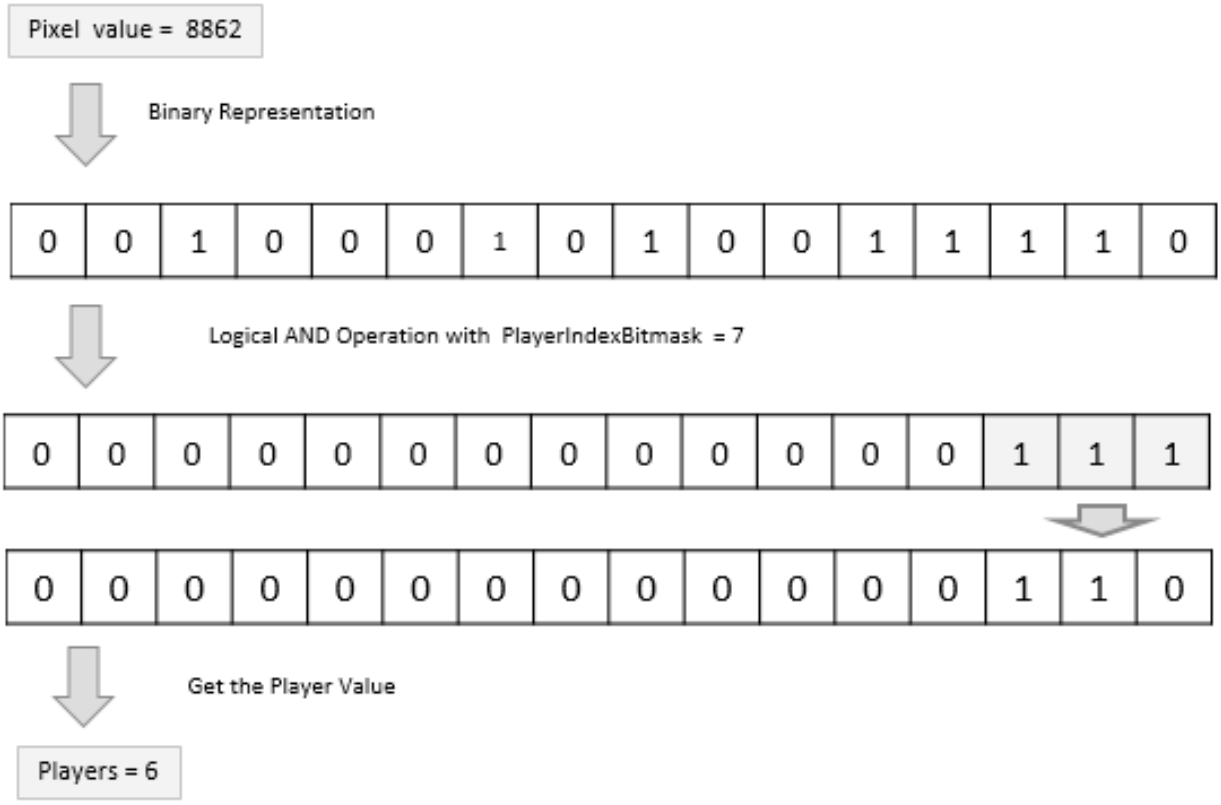

Figure 12 – The steps taken during a logical AND operation with the PlayerIndexBitmask constant to retrieve the player index (Jana, 2012)

4.1.2.3.3 Tracking human skeleton and joint movements

With the Kinect sensor, tracking the person in front of it is achieved by using the raw depth data. Initially the person that is standing in front of the sensor is just another object on the depth image frame. By using the human pose recognition algorithm an attempt is made to recognize the person from the other objects in the depth image frame. The human pose recognition algorithm uses several base character models that vary from one another on several factors such as height, size, clothes, etc. From these base characters machine learned data is collected where each individual body part is identified and labeled. These are then matched with the depth image frame based on probabilities.

Initially the human body is just one of the many objects in the space the sensor sees. As a result the human body looks like the first body in figure 13.

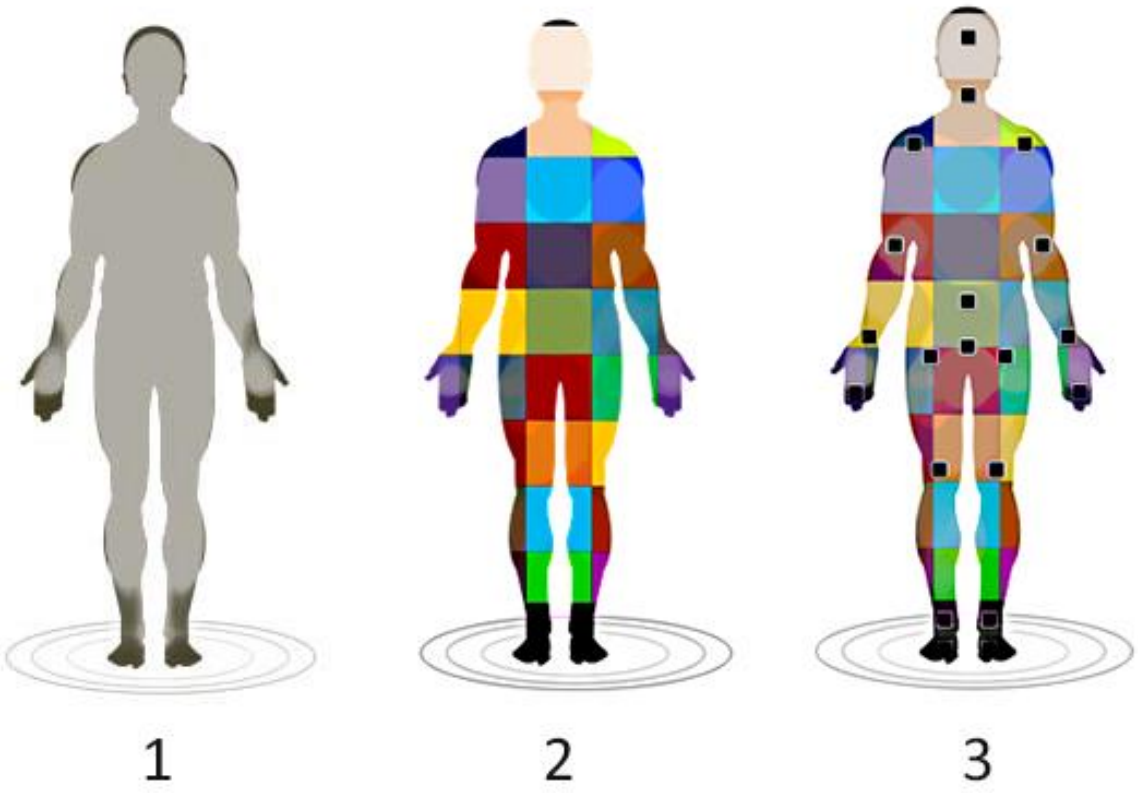

Figure 13
1 – The pixels of the person in the depth image is being recognized as a human body
2 – The human body is segmented to label the body parts
3 – The joints are placed on the body based on reference data
(Jana, 2012)

In order to start identify this object as a human body the sensor matches each individual pixel of the depth data with the data it has learned. Of course the match is seldom perfect thus a successful match is based on probability. When a match had been found the process of identifying the human body immediately goes to the next stop by creating segments to label to the body parts. This segmentation is done by using a Decision Forrest, which consists of a collection of independently trained decision trees[19]. With this method the data is match to specific body types. Every single pixel is passed through this forest where all the nodes represent different model character data labeled with body part names. When a match has been found, the sensor marks the pixel to eventually divide the body into separate segments as seen in the second body of figure 13. When this process is finished the sensor positions the joints on the places that have the highest probability of matching with the referenced data. The third body in Figure 13 shows the result.

[19] http://en.wikipedia.org/wiki/Decision_tree

The coordinates (X, Y and Z) of each of the joints are calculated by three views of the same image namely, the front view, the left view and the top view. This is how the sensor defines the 3D body proposal.

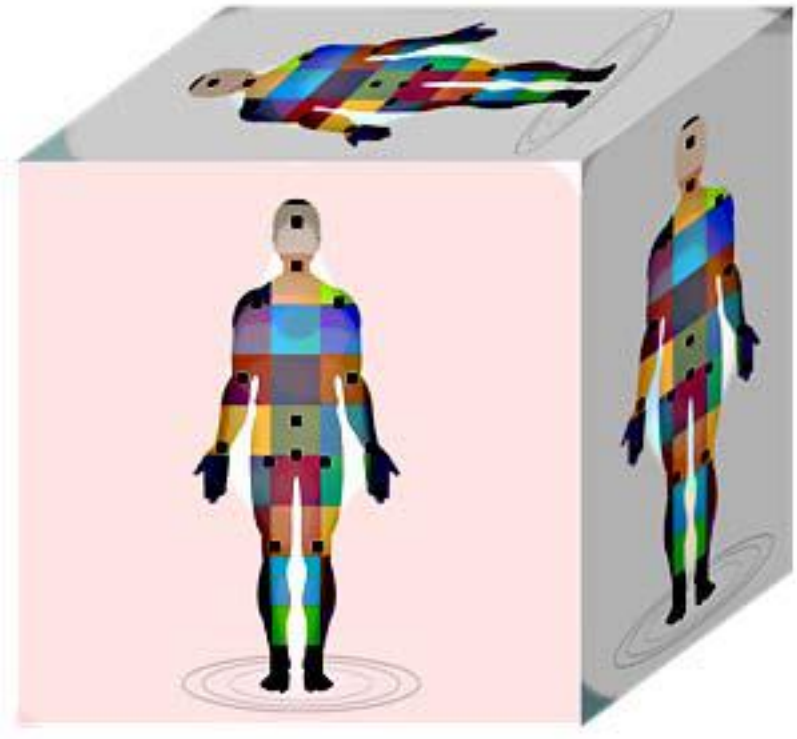

Figure 14 – The views by which the X, Y and Z coordinates of the joints are calculated (Jana, 2012)

The Kinect SDK gives the developer access to all 20 skeleton joints. Each joints is identified by its name (head, shoulders, elbows, etc.) and during tracking are each in one of their three states; Tracked, Not Tracked or Inferred. The skeleton as a whole is in one of its three states as well during tracking namely; Not Tracked, Position Only or Tracked. Figure 15 shows a complete skeleton with all 20 joints visible.

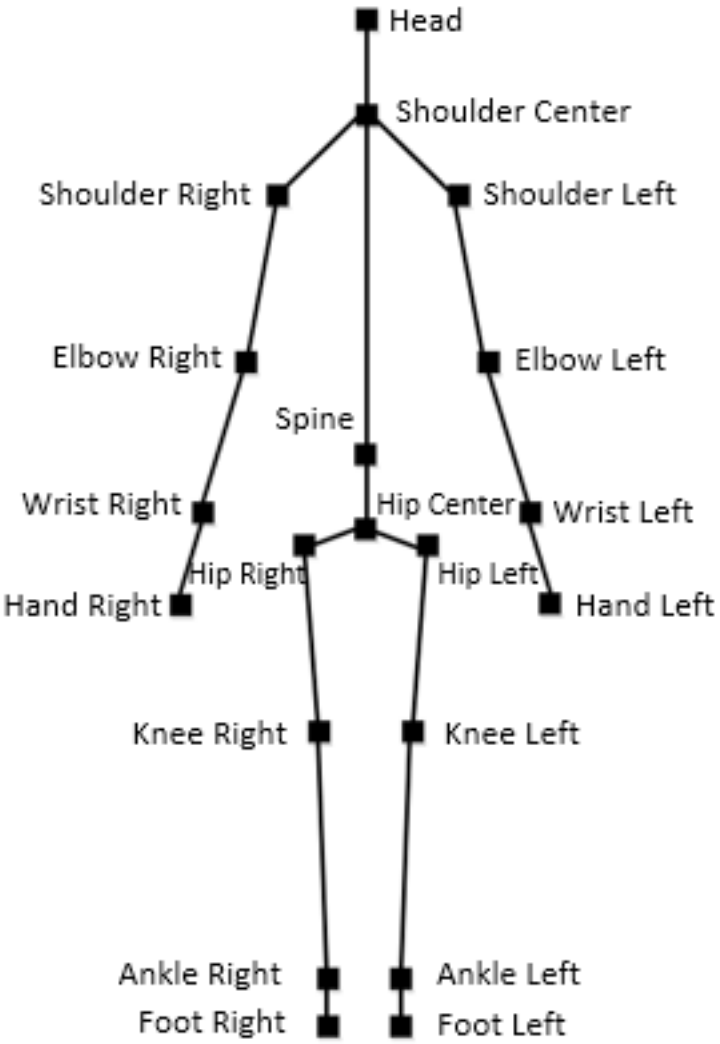

Figure 15 – A complete skeleton with all 20 joints as generated by the Kinect SDK (Jana, 2012)

The sensor is able to detect a maximum of six users but is only able to fully track two of them. The other four will only return the information of their hip center joint. But even of the two fully tracked users, only one will be actively tracked whereas the other skeleton is passively tracked. The difference is that the actively tracked skeleton returns all the available skeleton data but the passive skeleton only returns the proposed positions of the joints. Other data such as joint rotations are not available.

It is also possible to track a seated skeleton instead of the default full skeleton. This mode only tracks and returns 10 joints points, as seen in figure 16. If the developer only needs to track these joints, using the seated skeleton mode can save a lot of unnecessary data being sent from the sensor to the application.

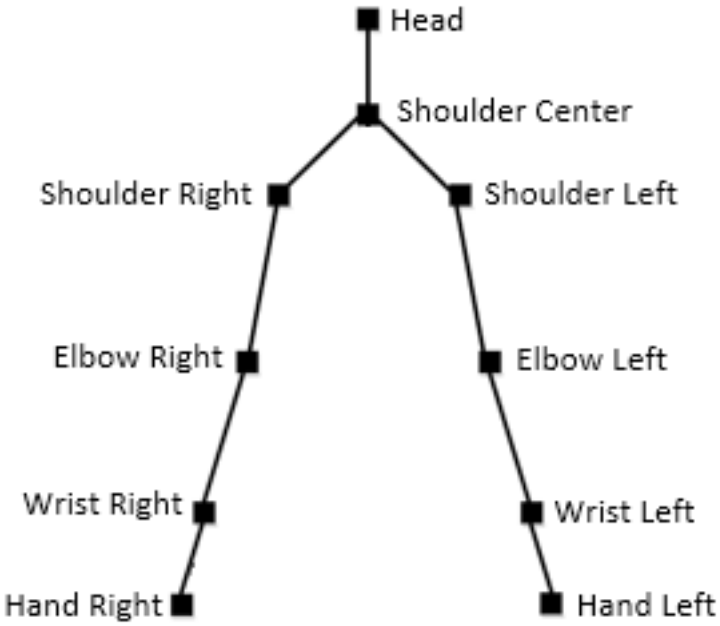

Figure 16 – The skeleton generated when the Seated Mode is active (Jana, 2012)

When tracking a user, the data of the skeleton is returned to the application in a SkeletonStream object. This object can be set to either seated or default mode to determine whether you would want all the joints or just the ten from figure 18. Through the SkeletonStream, frames are passed containing a collection of the skeleton joints. If it is the active skeleton, these joints contain all available data besides their position.

The joints in the skeleton are organized hierarchically. This means that every joint can be a parent and a child unless the joint is a leaf such as the head and hands. The highest joint in the hierarchy is called the root joint and every skeleton only has one of these. With the full, default skeleton the root joint is de Hip Center joint. For the seated skeleton this is the Shoulder Center joint. Figure 17 and 18 show the hierarchy of both tracking modes.

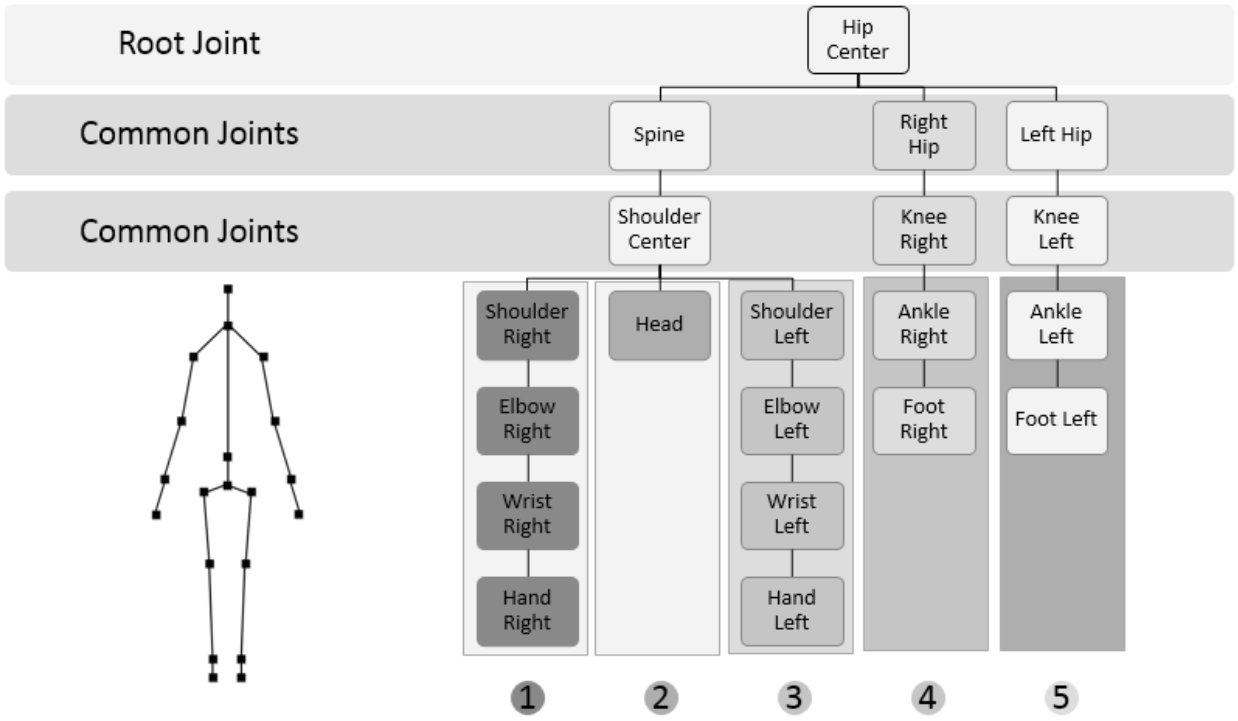

Figure 17 – The joint hierarchy with the default skeleton (Jana, 2012)

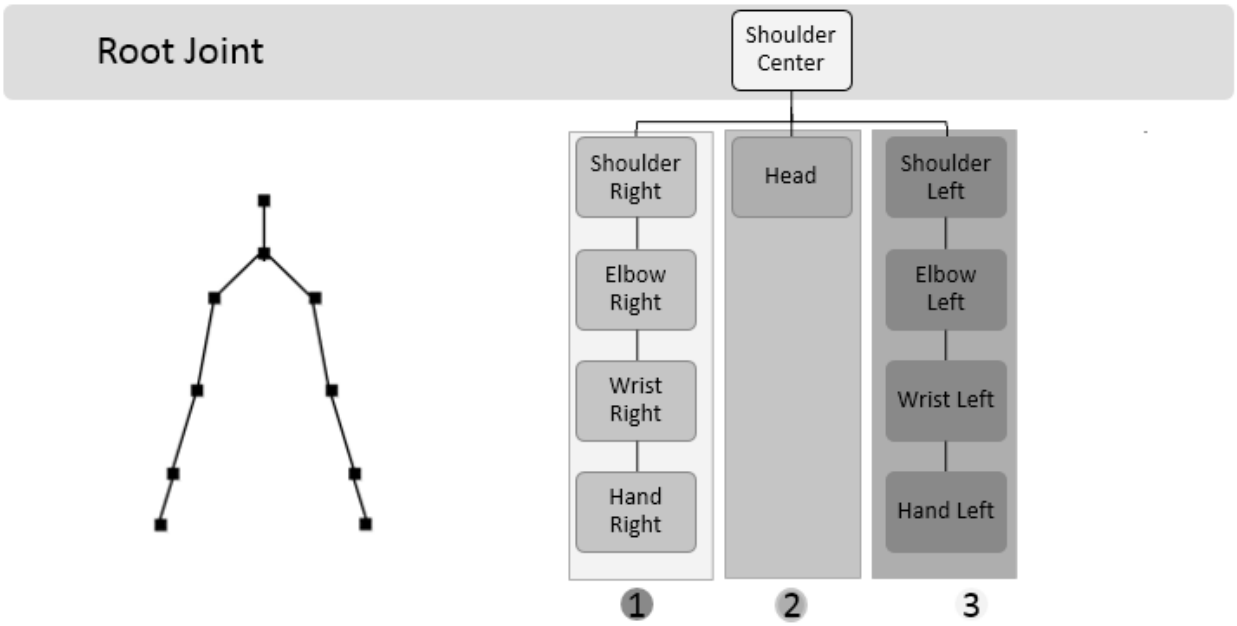

Figure 18 – The joint hierarchy with the seated skeleton (Jana, 2012)

Being hierarchically organized also means that transformations applied to the parent joints also drive the transformations of their children. Both figure 19 and 20 show which underlying joints are influenced when applying some type of transformation to one of the upper joints.

4.1.2.3.4 The Kinect SDK architecture for Audio

The Kinect audio components are built upon existing Microsoft libraries namely the DirectX Media Object (DMO) and the Windows Speech Recognition API (SAPI). The Kinect SDK just exposes the ability to features such as Noise Suppression (NS), Acoustics Echo Cancellation (AEC) and Automatic Gain Control (AGC) which are provided by the DMO which can be controlled through a Kinect application. The Kinect Audio Speech recognition however is completely based on the Windows Speech API.

In order to capture audio, the Kinect has a microphone array that consists of four microphones. The rationale behind using an array was because the two most important focus areas for the Kinect concerning audio was:

1. Human speech recognition
2. Recognizing, or rather locating, the voice of the players within a 3D environment.

Figure 20 shows the placement of the four microphones in the Kinect sensor.

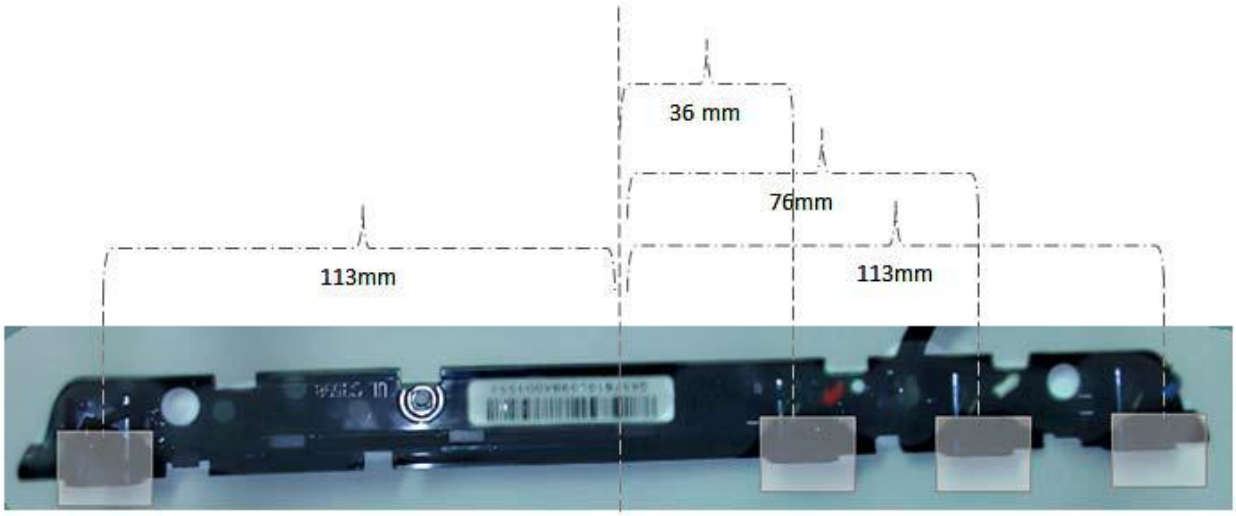

Figure 19 – A photo of the microphone array inside the Kinect sensor (Jana, 2012)

The sound can be identified within a range between +50 and -50 radians. The positioning of the microphones causes the incoming sound to arrive at different time intervals at each of the microphones. By using this difference the Kinect is able to detect the direction and the distance of the sound. These calculations are done within the hardware of the Kinect. After this is done, the audio-processing pipeline merges the sound of all four microphones, also known as beamforming, to produce a signal with high-quality sound. In light of voice recognition, the pipeline suppresses all frequencies that fall outside that of the human voice (80Hz – 1000Hz). Additionally, the pipeline filters out noise with Noise Suppression (NS) and echo with Acoustic Echo Cancellation (AEC) and amplifies the voice by applying the Automatic Gain Controller (AGC).

When using speech recognition, the audio-processing pipeline sends the cleaned up signal to the speech recognition engine. Here, the acoustic model component analyzes the signal and breaks it apart till there

are only basic speech elements left, which are called phonemes. *A phoneme is a basic unit of a language's phonology, which is combined with other phonemes to form meaningful units such as words or morphemes*[20]. In this context, phonemes are used to match with the voice. The phonemes are put through the language model component where the audio signal is analyzed and the phonemes are combined into a word which is than matched within a built-in digital dictionary to see whether it combined the correct word. If there is a match, the speech recognition engine will know what you said. Additionally, the built-in dictionary also provides information on the pronunciation of the words to help with the recognition. The speech recognition is also able to recognize words by the context in which they are used. For example, the words 'their' and 'there' are hard to differentiate from each other just by hearing them. But when used in a sentence such as 'keep it there', the recognition engine knows the word 'there' is being used and not the word 'their'. Figure 20 shows a basic overview of the speech recognition process.

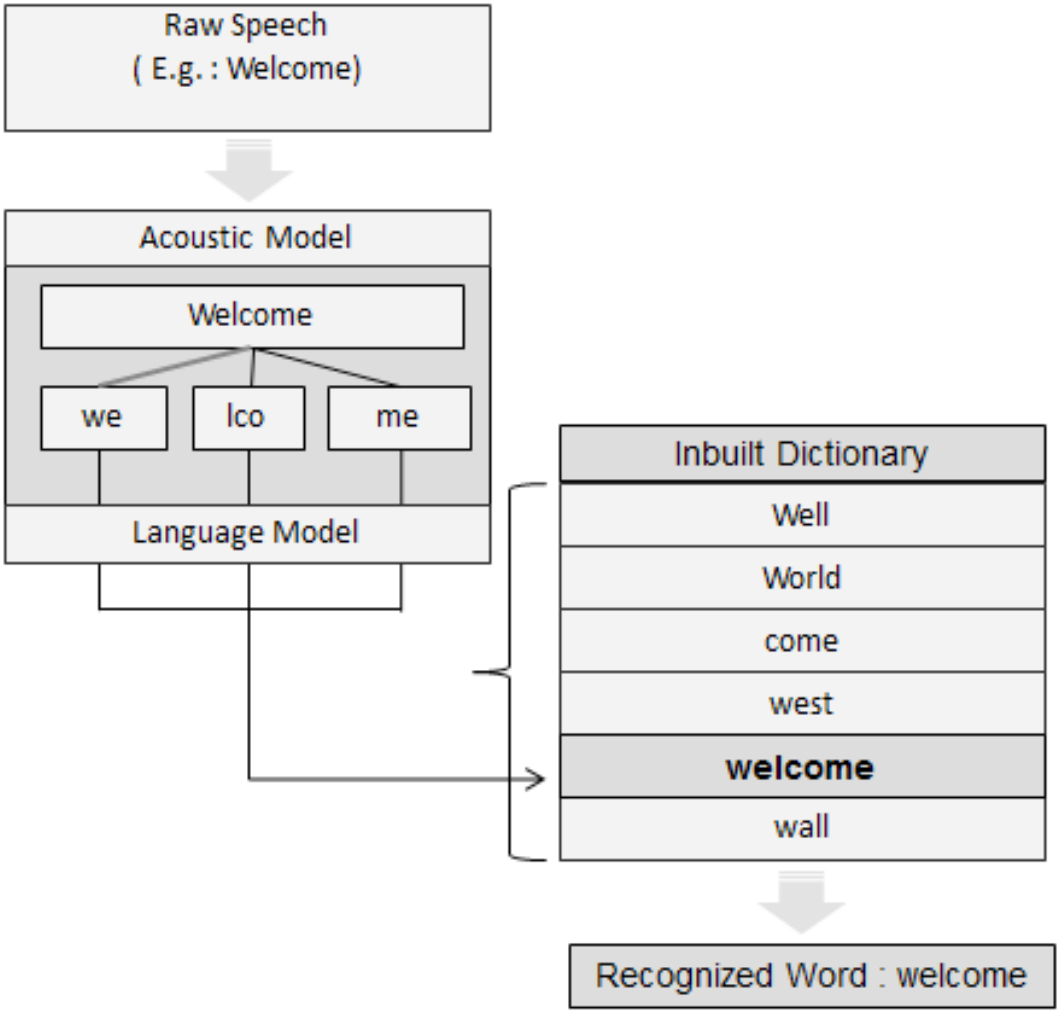

Figure 20 – A diagram representing the process of speech recognition with the Kinect SDK (Jana, 2012)

[20] http://en.wikipedia.org/wiki/Phoneme

4.2 The Kinect v2

On July 15th 2014, Microsoft began shipping their new Kinect sensor, simply named Kinect v2. They also released the public preview version of the Kinect SDK 2.0. The new Kinect has been improved on many fronts compared to the first Kinect.

4.2.1 The hardware

The color camera of the Kinect v2 now supports a resolution of 1080p running between 30 FPS and 15 FPS depending on the lighting conditions. The depth sensor has its resolution increased as well from 320x240 with the v1 to 512x424 with the v2. This has been made possible with what Microsoft calls Time of Flight. Where the v1 used an infrared grid pattern for depth sensing, the new infrared beam sensor is modulated and measures how long it takes for photons to travel to any object in its view. Additionally this new infrared technology ensures that the Kinect v2 sensor does not depend on the lighting conditions of the environment it is used in. The infrared image stream runs at 30 FPS as well.

The range and field of view have been improved as well. The field of view has been expanded in height and in width which in turn improved the range as well. The Kinect can now track between the 0,5 meters to 4,5 meters. The depth sensor can even measure up to 8 meters.

Concerning skeleton tracking, the Kinect v2 can track 6 full skeletons simultaneously where each skeleton consists of 25 joints. A few of these extra joints were put in the hands which makes it possible to track the thumb and other four fingers as two separate joints and the ability to discern between an open and closed hand. The skeleton has been corrected anatomically, particularly where the hips, shoulders and spine were concerned. These joints were located too high in the body.

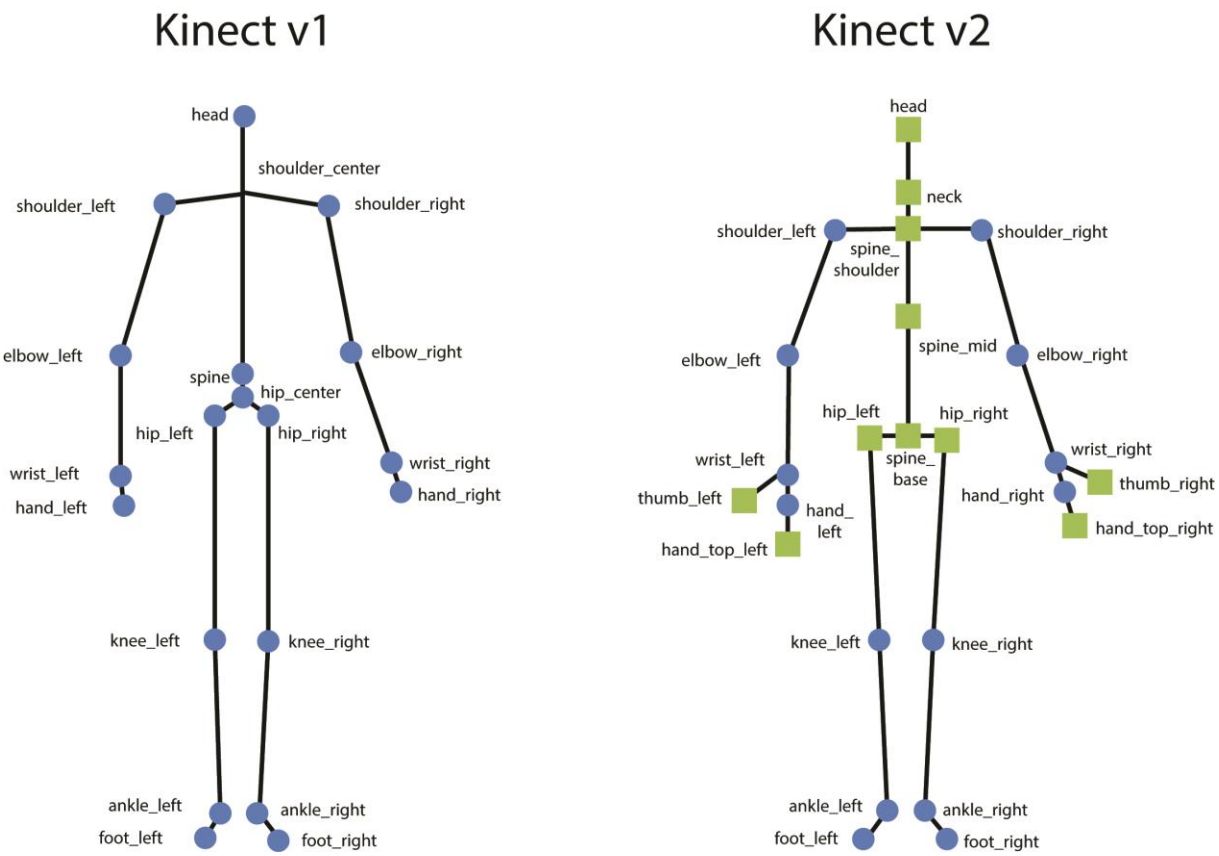

Figure 21 – The differences between the skeletons generated by the Kinect v1 and the Kinect v2

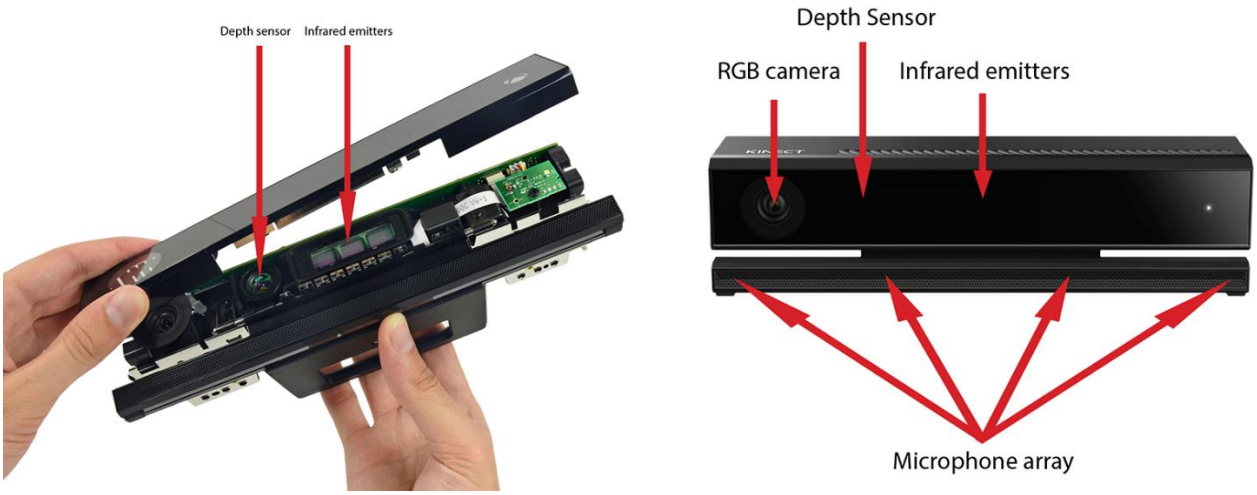

Figure 22[21] – The placement of all the important parts on the Kinect v2

[21] https://www.ifixit.com/Teardown/Xbox+One+Kinect+Teardown/19725

4.2.2 The software

Microsoft has taken care of the software as well. On the programming side it is now easier to develop applications for the Kinect v2 because it now takes less lines of code to initialize functions of the Kinect sensor or retrieving data from stream objects such as the skeleton or color stream.

The SDK has also been changed in that it is now possible to develop a Kinect application and compile it as a desktop application, Windows Store application or Xbox One application. Out of the box the new SDK will also include a Unity plugin to develop Kinect enabled games in the Unity 3D. But Microsoft also delivers a tool called Kinect Common Bridge. Kinect Common Bridge is essentially a toolkit that enables developers to access the Kinect API from within other programming environments. This could be an application such as MATLab where you would create a Kinect driver or creating wrappers for other programming languages.

Two other major features of the new SDK is advanced face tracking. The Kinect is now able to recognize a face and track its position and orientation. Out of the box it can also track mouth and eye movement. The second major feature concerns the face as well and it is an extension of the Fusion feature that was introduced in the SDK v1.7 and is called HD Face. This feature makes is possible to scan the face in very high resolution, opening up the possibility to create detailed 3D models of people.

The toolset of the Kinect has been improved and extended as well. The new Kinect Studio has an improved UI which can be organized to your own taste. It also provides better features to choose which data types you want to record, making it easier to just record the data you need and saving bandwidth at the same time. When testing Kinect v2 applications by playing back recordings with Kinect Studio, having a Kinect sensor attached to the computer is not required anymore. This makes it easier to develop your application anywhere you would want without having to bring along your sensor as well.

A new tool has been added as well, called Visual Gesture Builder. With this tool it becomes easier to create more complex gestures. These gestures can then be used to interact with your application. I will be discussing Visual Gesture Builder more in-depth in section 4.2.5.

5 Developing the prototype

For this thesis I wanted to develop a prototype that would make use of the Kinect sensor to validate the quality of certain movements that are done in CrossFit. The two major features of the prototype were to:

1. Provide an accessible manner for domain experts, trainers, coaches, etc., to add and edit movements that can be used for the exercises in the prototype.
2. Recognize incorrect posture types in order to be able to give relevant feedback.

As you know after reading chapter 3, I am not the first to create an application for the Kinect sensor to track people when they are exercising. Using the Kinect sensor for such cases is becoming more common since it can lower the cost of rehabilitation and personal training. These application however focus mainly on helping the users to adhere to the exercises by making exercising fun and accessible. In contrast my prototype would focus more on helping to improve the posture of the user. It would do this by being able to differentiate between correct and incorrect postures and determine what was done incorrectly.

Unfortunately I was not able to complete the prototype due to the limitations of the Kinect v1 which I discuss more in depth in chapter 6. In this chapter I will discuss the development process of the prototype. I will do this by:

- Giving a short introduction to CrossFit and why I chose this sport to test the prototype.
- Discussing the design of the prototype.
- Introducing what Microsoft calls gestures and how I analyzed the three movements I intended to use for the prototype.
- Discussing the implementation processes of the movements into the prototype with both the Kinect v1 and the Kinect v2.
- Finally discussing the results.

The prototype will be built using Microsoft Visual Studio and the official Kinect SDK v1.x. Even though there are other tools, wrappers and libraries available, they are all created and supported by third-parties and sometimes depend on the official Kinect SDK. All the features of the Kinect sensor and the Kinect SDK are fully supported in Visual Studio. This is also the fastest way to have access to new features and improvements that are made available my Microsoft.

5.1 CrossFit

CrossFit is a relatively young sport that has grown in popularity very quickly. Created in 1995 by Greg Glassman, CrossFit is a sport where its practitioners train in the broadest sense of fitness or in Glassmans words: *"Our specialty is not specializing. Combat, survival, many sports, and life reward this kind of fitness and, on average, punish the specialist"*[22]. This fitness as defined by CrossFit, to which athletes are measured, rests on three standards. The first standard is the ten general physical skills;

[22] http://www.crossfit.com/cf-info/what-crossfit.html

cardio respiratory endurance, stamina, strength, flexibility, power, speed, coordination, agility, balance, and accuracy. The second standard comes from the basic idea of how CrossFit defines fitness. The idea of the athletes' performance across a broad range of physical tasks. Within CrossFit this is referred to as the 'hopper'. The idea behind this is that you would put every physical task you can imagine in a hopper, mix them all up, take one out, perform the task and do so multiple times. The athlete would then be measured by how consistently he would perform at any of those pulled tasks. The third and last standard concerns three metabolic pathways. The phosphagen, glycolytic and oxidative pathways, which ensure that the human body receives the energy needed for any kind of activity. The fitness of an athlete is determined by how well he performs across all of the elements of every one of the standards.

If you want to perform well across all of these elements, you need to constantly perform varied, high intensity, functional movements. These movements can be divided into three categories, or modalities. These modalities are gymnastics, Olympic weightlifting and metabolic conditioning, also known as cardio. This is why typical CrossFit workouts vary from day to day by design. These workouts usually consist of three parts, where the first part is the warm-up. The second part is a skill or strength development segment where you focus on one particular movement in order to improve upon it. The third part is the 'Workout Of the Day', also known as the WOD. This WOD consists of various functional movements which are performed at high intensity. The WODs are designed with a competitive game element in mind. By designing rules and standards around the WODs, they can be scored and timed, turning it into a sport that can be fun to do against or with your fellow athletes. (Paine, Uptgraft, & Wylie, 2010)

5.1.1 Why CrossFit

So why did I choose CrossFit to test my system? Besides that I am a CrossFit participant myself which makes access to the trainers and facilities somewhat easier, using correct form during movements is something that is hammered upon during every WOD. Because of the competitive aspect, CrossFit athletes are often very motivated to put down the best score possible during WODs. This means that they often lift heavier weights and execute movements faster than they usually do. What often happens in those situations is that the athlete is less focused on the quality of his movement. This can result in not using the proper form when performing the movement, which in turn increases the chance on injury. That is why it is not just important to use the correct form, but also to use it consistently as well (Fuhr, 2013; Goyanes, 2013). A motion tracking system could help with teaching athletes to use the correct form without having a trainer to be constantly present. Also, a large part of CrossFit exercises consist of movements used with Olympic weightlifting and these movements are often performed in one spot. This makes it easy to use a Kinect to track the motion of the athlete because the area in which the athlete performs is small.

5.2 Design

As stated earlier, due to the hardware limitations of the Kinect v1 I was unable to develop a prototype. Additionally, creating gestures for the Kinect v1 is quite complex which I will discuss in section 5.3 and 5.4. The intention was to create an application that would validate the movements performed by a user. It would also support the feature for a trainer to capture, edit and save new and existing movements

5.2.1 The hardware setup

The setup of the prototype would have been quite simple. The Kinect sensor would handle the input from the user to the application with its built-in depth and audio sensors. The Kinect sensor would be connected through a USB 2.0 port to the Windows computer upon which the application would be running. The user would interact with the application by using gestures and voice commands that would be registered by the Kinect sensor. The feedback given by the application would be communicated back to the user through the computer screen and speakers that are connected to the Windows computer.

The trainer would use the traditional mouse and keyboard to edit the recordings of the movements that will be saved in the database. The trainer can see his or her actions through the user interface presented on the computer screen. The database itself would have been just a structured data file that would contain all the saved movements and the rules that would determine when a movement was executed correctly and if not, what the mistake was and how to correct it. Figure 35 shows a diagram of all the components of this setup.

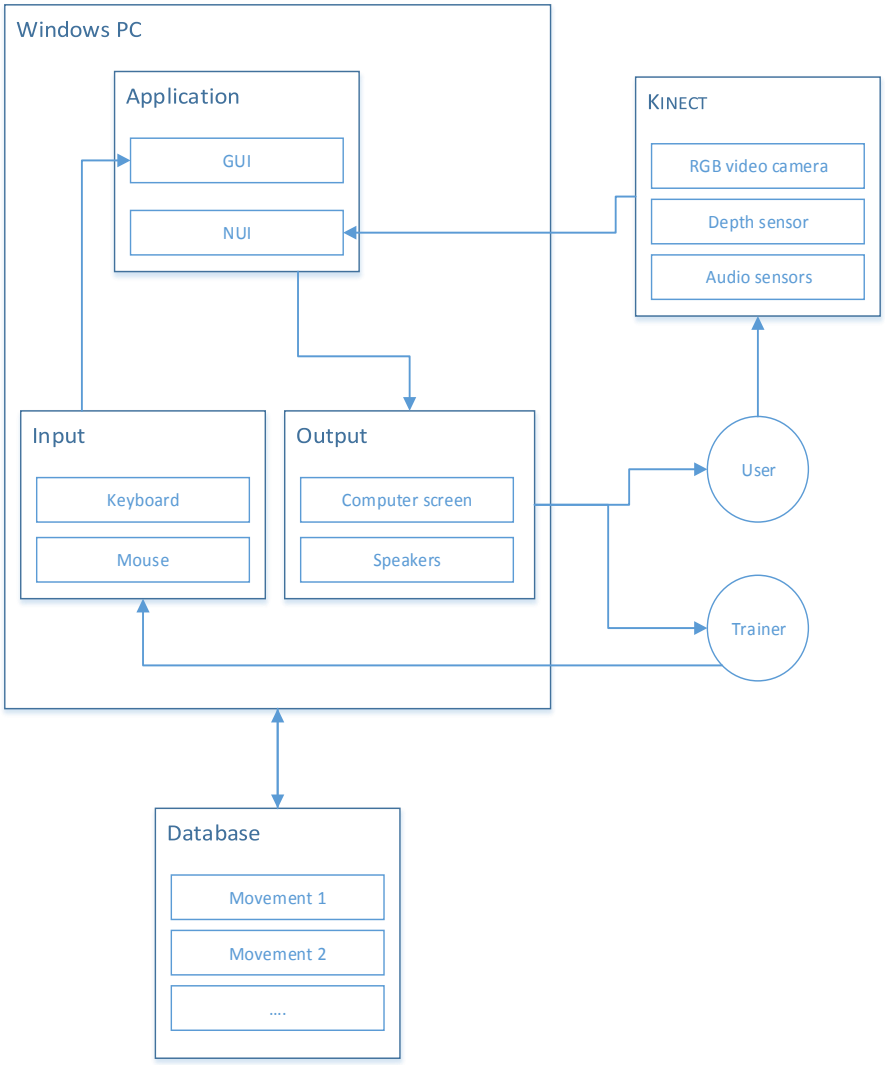

Figure 23 – A diagram of the components I used with my prototype

5.2.2 The application

The prototype would have initially focused on the trainer and how he or she could have created new movements that could then be used when exercising with the prototype. The feature to validate the quality of the movements performed by the user would also be implemented since the implementation of both are linked to each other. But it would not be as extensive as the movement editing feature. Below I have listed a set of requirements for the prototype and a use case diagram that visualizes the functional requirements.

5.2.2.1 Requirements

With the requirements I have made a few assumptions that need to be clarified. The trainer can initially create a movement. A movement can be a squat, a pushup, a clean and jerk, etc. When editing a movement the trainer chooses the start, end and potential midway positions that will determine one full repetition of that movement. The trainer can then create an exercise that can consist of one or more movements where each movement consists of a set amount of correct repetitions.

#	Short name	Short description
FR01	Check tracking state	The trainer should be able to see whether the SkeletonTrackingState of a joint is either Tracked, PositionOnly or NotTracked.
FR02	Turn tracking on/off	The trainer should be able to turn the tracking of individual joints on or off.
FR03	Create movement	The trainer should be able to create a new movement that can be used during a workout.
FR04	Edit movement	The trainer should be able to edit newly created and existing movements.
FR05	Create start position	The trainer should be able to set a start position for a movement.
FR06	Create midway position	The trainer should be able to set a midway position for a movement.
FR07	Create end position	The trainer should be able to set an end position for a movement.
FR08	Edit start position	The trainer should be able to edit the start position for a movement.
FR09	Edit midway position	The trainer should be able to edit the midway position for a movement.
FR10	Edit end position	The trainer should be able to edit the end position for a movement.
FR11	Create threshold	The trainer should be able to create one or more thresholds to which the joints must abide during the movement.
FR12	Edit threshold	The trainer should be able to edit the thresholds.
FR13	Create notification	The trainer should be able to create a notification for a threshold that will be shown when the threshold is exceeded.
FR14	Edit notification	The trainer should be able to edit a notification.
FR15	Save movement	The trainer should be able to save a movement.
FR16	Create exercise	The trainer should be able to create a new exercise.
FR17	Edit exercise	The trainer should be able to edit an exercise.
FR18	Add movement	The trainer should be able to add a movement to an exercise.
FR19	Remove movement	The trainer should be able to remove a movement from an exercise.

FR20	Set repetition amount	The trainer should be able to set the amount of correct repetitions the user must make for this movement in the respective exercise.
FR21	Save exercise	The trainer should be able to save the exercise.
FR22	Select exercise	The user must be able to select an exercise.
FR23	Perform exercise	The user must be able to perform the selected exercise.
FR24	Count correct repetitions	Count the correct repetitions during an exercise.
FR25	Count incorrect repetitions	Count the incorrect repetitions during an exercise.
FR26	Identify incorrect repetition	Identity the type of mistake the user has made during a repetition by checking which threshold was exceeded.
FR27	Communicate every repetition	The user must be made aware how he or she executed each repetition. If the repetition was executed incorrectly, the user must be told what he or she did wrong.
FR28	Show progress	The user must be able to see and/or hear how many correct repetitions must be done to complete the exercise.
FR29	Show summary	The user must be able to see a summary of his or her performance after the exercise.

Table 1 - Functional requirements

#	Short name	Short description
QR01	Quality	Joints that are obstructed by body parts cannot be detected correctly by the Kinect and can behave erratically as a consequence. This has to be taken into account to make sure that the accuracy stays within tolerable limits.

Table 2 - Quality requirements

5.2.2.2 Use case diagram

In the following use case diagram I have also added the Kinect sensor as an actor. I have done this because some of the use cases depend on the Kinect sensor. And since the sensor itself is something I cannot edit, it sits outside de system boundaries as an actor.

The reason that the Kinect sensor is only used for the use cases involving the user and not the trainer, is because I made the assumption that the recordings that will be used for creating and editing movements are made with Kinect Studio beforehand.

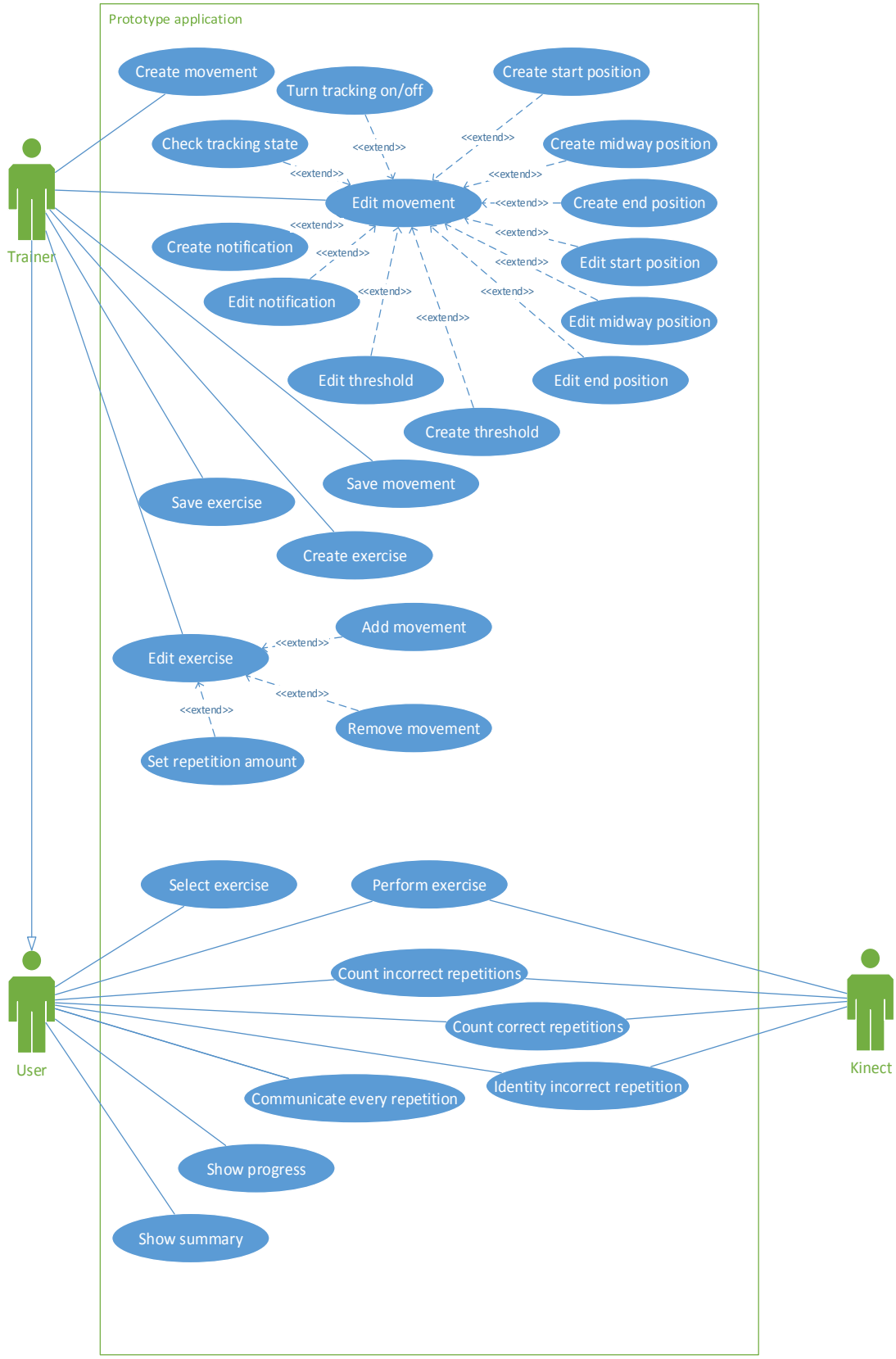

Figure 24 – The use case diagram of my prototype

5.2.2.3 Trainer interactions

One of the main features for this prototype was for a domain expert, the trainer, with no programming knowledge to be able to create the set of rules for each movement themselves. As soon as a new one is created, the domain expert is automatically being sent to the edit page. Here the domain expert can give the movement a name and set the conditions that determine a correct repetition of that movement. By playing back a recording made with Kinect Studio, the domain expert is able to set these conditions. These conditions consists of two (start and end) or more key positions and a number of thresholds that should be maintained during the whole repetition.

Let us take the air squat as an example. With the air squat, the arms can be used in various ways to help executing the air squat by improving stability and/or speed. But since it is not an important part of correct posture, the arms can be left out. So that leaves us the upper body and legs. First I will break down the start position, constraints, halfway position and end position of a well-executed air squat.

- **Start position**
  - The user stands right up with a straight back.
  - The legs are straight as well with the knees locked out.
  - The hips are opened up, which means that they are pushed forward slightly.
  - From the side the body should form a very straight line.
- **Constraints**
  - During the motion, the knees should not fall inwards.
  - The heels should stay on the floor.
  - The chest should remain up as much as possible.
  - The back has to remain straight. No hollowing or bulging of the back.
  - Feet need to be around shoulder width.
- **Halfway position**
  - The hip crease should be at least slightly lower than the knees. It is ok to be lower.
  - The pelvis/hips may not rotate inwards. An issue also known as a ‘butt-wink’.
- **End position**
  - The same as the start position.

The air squat consists of three key positions and five constraints that can be set as thresholds. First, the domain expert will go through the frames of the recording until the start position is found. By making a snapshot of that recording, the positions of the joints relative to each other and their orientation in 3D space are saved and will determine the start of a repetition of that exercise. The same will be done for the midway and end positions. To determine the thresholds, the GUI will provide the domain expert resizable rectangles which can be put over the video frame to create areas within which selected joints are allowed to be for a repetition to be correct. Take the left knee for example, it is ok for the knee to move to the left when going into the squat, but it is not allowed to go to the right. A rectangle can be placed and resized over the area within which the left knee is allowed to move. By linking this rectangle to the left knee joint of the skeleton created by the Kinect SDK, the correct position of this joint will be determined by its position within this rectangle, where the upper left corner of the rectangle will be 0,0. By selecting a side of the rectangle, the domain expert can create a notification that can be shown or

played to the user when the joint falls outside that side. Thus giving the domain expert an easy way to create different notifications depending on the type of incorrect execution of the exercise.

When the domain expert is done editing the exercise, the exercise will be saved in schema like file format such as XML. This file will be read in by the framework when a user wants to use the framework to track his or her air squats.

5.3 Analyzing movements

5.3.1 Gestures

For the application to be able to determine whether one of the movements is executed correctly or not, I needed to create a gesture for each of the movements. Microsoft uses the term gesture as *"any form of movement that can be used as an input or interaction to control or influence an application" (Microsoft, 2013)*. In other words, if the skeleton stream fulfills a set of predefined requirements, we can attach some action to it. There are several approaches you can take to create gestures. The approach to choose depends on the type, or rather, complexity of movement you would like to turn into a gesture.

5.3.1.1 Approaches

(Jana, 2012) has classified four approaches to create gestures:

- Basic approach
- Algorithmic approach
- Neural network approach
- Template-based approach

Basic approach

As the name states, this is the simplest of the four approaches and also the most straightforward. Creating gestures using this approach consists of creating a set of conditions, known as the result set which will be matched with the movement being performed. If the movement matches the result set, the gesture is performed successfully. This result set often consists of the joints that you want to track for the gesture and the value(s) distance, position and/or angle between them that results in a successful recognition of the gesture.

Algorithmic approach

The algorithmic approach is built on top of the basic gesture recognition. Where the basic gesture recognition just looks whether two or more joints of the skeleton stream fulfill a simple condition, the algorithmic approach extends this by validating the joints throughout the whole motion. When using this approach, the tracked joints in the result set also have a start position, must move within certain constraints and within certain parameters and have an end position. All of these requirements must be met for the gesture to be successfully recognized.

Template-based approach

Also known as the pattern-based approach, by using predefined gestures it is possible to measure the ratio of the movement being performed. This approach consists of three phases:

1. Template creation
2. Gesture tracking
3. Template matching

During the first phase, the gestures are recorded and stored. Alongside the usual data of the joints the recordings also contain metadata with the gesture's name, types of gestures, time interval for the gestures, minimum and maximum duration, etc. The second and third phase are usually done side-by-side. As the user is performing the movements in front of the sensor, the gesture recognizer is matching the movement with the predefined set of templates continuously. The template data can be stored in various ways such as a database, XML or flat files. A successful gesture is recognition if the movement matches the template exactly, strictly speaking. But it is best to set the recognition to the nearest values of the template.

Neural network approach

The artificial neural network, or just neural network, was inspired by examinations of the central nervous system of humans. Artificial networks are similar to biological ones in that they are able to perform functions collectively and in parallel. A typical neural network consists of nodes, also known as neurons, neurodes, processing elements or units. These nodes are connected to each other where they form a network which mimics the biological neural network. The connections between the nodes are associated with an adaptive weight which is tuned by a learning algorithm. These weights are capable of approximating non-linear functions of their inputs.[23] (Wang, 2003)

When applied to gesture recognition, the neural network typically consists of an input layer, a hidden layer and an output layer. Each node in the input layer evaluates a small element of the gesture that is being evaluated. The result of this evaluation and the weights of the connected links decides which node to move to next. The result of the calculation at the end of the complete network is called the calculated output for the user input action. This result is matched with the predefined expected output, also known as the best-values output for the detected gesture. Since the result is calculated based on probability, it is not uncommon that the result is never the same as the predefined expected output. It will at least have some difference or error. By setting a predefined boundary you can decide within which probability ratio the result should be accepted. (Jana, 2012)

5.3.1.2 Implementations

The basic and algorithmic approaches are the two that are the most applied when building Kinect applications. These two are the most straightforward of the four approaches and the developer only needs to tell which joints to track and what the condition should be to measure a successful gesture. This condition always comes down to the coordinates of the joints and where they should be relative to each other. For example, if would want to create a clapping gesture, all you would need to look for are

[23] http://en.wikipedia.org/wiki/Artificial_neural_network

the two hand joints and when they are within a certain range, using the x-axis, of each other. That would be the basic approach. In this case only the distance of the hand joints would be measured and the user would not actually have to clap for the gesture to work.

This could be improved upon with the algorithmic approach by giving the gesture a start position. For instance, the hand joints should have minimum distance between each other. It is also possible to take their position in relation to the shoulder and elbow joints into account. This could be done by saying that the clapping gesture is only valid when the hand joints are below the shoulder joints and above the elbow joints by comparing their y-coordinates. When the start position has been validated positively, the application should measure the distance between the hand joints and whether this distance decreases over time. This can be done by comparing the x-coordinates of the hand joints from one or more previous frames with the current one. Furthermore, the constraint that the hand joints should remain between the shoulder and elbow joints should be met during the entire motion. The gesture will then only be successful when the end position has been reached as well. Which could be the same condition as with the basic approach.

Now it is probably obvious by now that with the basic and algorithmic approach, you can choose which joints you would like to track for the gesture. This is great for simple motions such as waving, swiping, clapping, etc., because you can ignore the uninvolved joints and save some computing resources. Unfortunately these approaches are not sufficient the more dynamic movements, especially those related to physical exercises. Let us take the air squat as an example again as in section 5.2.2.3.

- **Start position**
  - The user stands right up with a straight back.
  - The legs are straight as well with the knees locked out.
  - The hips are opened up, which means that they are pushed forward slightly.
  - From the side the body should form a very straight line.
- **Constraints**
  - During the motion, the knees should not fall inwards.
  - The heels should stay on the floor.
  - The chest should remain up as much as possible.
  - The back has to remain straight. No hollowing or bulging of the back.
  - Feet need to be around shoulder width.
- **Halfway position**
  - The hip crease should be at least slightly lower than the knees. It is ok to be lower.
  - The pelvis/hips may not rotate inwards. An issue also known as a ‘butt-wink’.
- **End position**
  - The same as the start position.

As you can see, even when we disregard the arms, there are a lot parts that need to be followed and checked continuously. Not only that, but determining the values and thresholds of the involved joints and how they relate to each other can be very time consuming if done by hand using the algorithmic approach. This is a problem because even if only one user uses the application no two air squats will be

executed exactly the same. Let alone when different users use the application, since the application can give false negatives because of the slight anatomical differences between different people. Furthermore, it is very possible that not all joints will be visible the whole time during the motion.

This is where the neural network and template-based approaches are best suited for since both of them give probability ratios as a result instead of a Boolean. Unfortunately, both approaches are hard to implement in a Kinect application. The neural network approach requires a good understanding of fuzzy logic and artificial intelligence. Both of which fall outside the scope of this project and my field of expertise. The template-based approach requires an application that is able to record movements with the Kinect and convert them to a gesture template and export it to a file. This is a project on its own and creating one falls outside the scope of this project as well. There does however exist an application that records gestures called GesturePak[24]. GesturePak provides a user interface to record movements and edit them into gestures which can then be used in other Kinect application. It was created by Carl Franklin and he sells it through his own website. But by the time I discovered this application I already started experimenting with the Kinect v2.

5.3.2 The three movements

In order to keep this project manageable I decided I would try to validate only three movements on their quality. All three movements contain similar motions but differ in complexity. These movements are the air squat, front squat and overhead snatch.

The reason I chose the air squat is because it is a basic movement and the requirements for executing a good air squat are fairly easy to determine.

The front squat is similar to an air squat, but the athlete carries a barbell in front of him or her. Determining its quality is slightly more complex than the air squat because the positions of the arms have to be tracked as well. And even though most of the positions of the other joints should be similar to the air squat, the posture of the front squat is slightly different since there is a weight in front that is trying to pull the athlete forward.

The third exercise is the snatch. This is an exercise that is done in Olympic weightlifting as well, where the barbell is lifted from the ground to above the head of the athlete in one motion. During this motion the athlete often falls down into a squat to catch the weight. When the athlete executes this part successfully, the athlete then has to stand up straight while keeping the bar over his or her head. In addition to the squat, this movement contains a number of additional requirements that must be met in order to determine whether it was executed correctly. One of biggest difference from the air squat and front squat, Kinect wise, is the rotation of the arms along their lengths.

[24] http://www.franklins.net/gesturepak.aspx

Figure 25 - Air squat

Figure 26 - Front squat

Figure 27 - Snatch

5.3.3 Recording the movements

In order to help with the development of the prototype, I recorded the movements with Kinect Studio. The reasons for this were as follows:

1. I needed to determine the difference between a correct and incorrect movement.
2. To prevent me or someone else from performing a movement in front of the Kinect sensor every time I needed to test the prototype.

All the movements were performed by a man and a woman. Even though the Kinect sensor has been programmed to recognize a wide range of different body types and track them in the same manner, there are slight general differences between the way men and women move their bodies (Ford, Myer, &

Hewett, 2003; Graci, Van Dillen, & Salsich, 2012; Jacobs, Uhl, Mattacola, Shapiro, & Rayens, 2007; Zazulak et al., 2005). I wanted to see if there were differences in the tracking results between men and women and if so, if was necessary to create different sets of thresholds for both genders to determine when a movement was correct or not.

The reference movements were performed by coaches of CrossFit Amsterdam who are able to perform these movements correctly. These movements were recorded with one Kinect sensor and the Kinect Studio application included with the Kinect SDK. With the intention to use two Kinect sensors in an extended version of the prototype, I recorded the movements from the front and from the side. Each of the three movements consisted of four sets of nine repetitions. Each set will consist of three correct reps, three obvious incorrect reps and three reps that are slightly incorrect. Two sets will be recorded from the front and two from the side. The first of the two sets will be used as the reference set and the second will be used as the control set. At the end of the recording session, the recordings were analyzed to determine which repetitions were correct or incorrect and why.

5.3.4 Breaking down the movements

In order to create good gestures for the three exercises, I needed to determine the correct postures during each exercise and which parts of the body were involved. From there I tried to rationalize which joints to use and how to relate them to each other. To get an overview of the different postures during each exercise, I broke each of them down to their start position, constraints, end position and if applicable, their halfway position.

Exercise	Start position	Halfway position	End position	Constraints
Air squat	• The user stands right up with a straight back. • The legs are straight as well with the knees locked out. • The hips are opened up, which means that they are pushed forward slightly. • From the side the body should form a very straight line.	• The hip crease should be at least slightly lower than the knees. It is ok to be lower. • The pelvis/hips may not rotate inwards. An issue also known as a 'butt-wink'.	• The same as the start position.	• During the motion, the knees should not fall inwards. • The heels should stay on the floor. • The chest should remain up as much as possible. • The back has to remain straight. No hollowing or bulging of the back. • Feet need to be around shoulder width.
Front squat	• The same as the start position of the air squat.	• The same as the halfway	• The same as the start position.	• The same constraints as the air squat.

	• The elbows point as much forward as possible.	position as the air squat. • The (weighted) bar should have gone down in a straight line. Meaning, the horizontal position should be the same as at the starting position. • The elbows should still be pointing forward as much as possible.		• The elbows are not allowed to drop during any point in while executing the exercise. Ideally, the direction should remain the same.
Snatch	• Arms straight outside the knees, grabbing the bar. • Back straight and as vertical as possible. • The knees do not go over the toes. • The angle between the upper and lower leg is larger than 90 degrees.	• There is no halfway position.	• The body is completely straight and vertical, standing upright. • From the center of the bar through the side of the body, an imaginary right-angled line should be made with the floor. • The armpits are twisted forward, making them visible from the front.	• Ideally the back may not bend during the whole motion. • The arms may not bend before the body is completely straight after the initial pull. • The bar must remain as close to the body as possible.

Table 3 – The three movements broken down into their start position, end position, constraints and if applicable, their halway position

5.4 Implementing the movements

5.4.1 Kinect v1

5.4.1.1 Angle calculations

I have used the code from the application Advanced Skeleton Viewer from the book (Jana, 2012) as starting point. From there I started testing with my own application. My first goal was to find a way to determine when a movement was executed correctly. I first considered using the joint positions of the reference data of the correct movements. But since different body types cause the joints to have different positions relative to each other, this would not work. After reading a post on the Engineering Sport website (Choppin, 2011) I was inspired to try a different approach. In this post the writer used the Kinect, before the release of the official SDK for Windows, to test the quality of the motion analysis. To analyze the squat movement, they looked at the angle between the upper and lower leg. With the Kinect SDK it is possible to calculate this angle with just the information of the joints, so I tried this approach by first seeing whether I could calculate the angle myself.

In order to verify the correct pose of the subject, I calculated the angle between the bones. In order to calculate these angles I used the absolute position of the joints. It is important to use the coordinates in a 3D space because if you only use the x and y coordinates you might not get the correct angle because the angle might get distorted. It is also important to remember how the 3D space is setup by the Kinect. When determining the position of the joints the sensor uses the skeleton space to determine the coordinates. Within this coordinate system, the sensor is the origin or X: 0, Y: 0 and Z: 0 and everything is expressed in meters. From the sensor, the positive z-axis runs towards the direction the sensor is facing. So the Z value of a joint is also the distance from the joint to the sensor in meters. The positive Y-axis goes upwards and the positive x-axis goes to the left when looking in the same direction as the sensor.

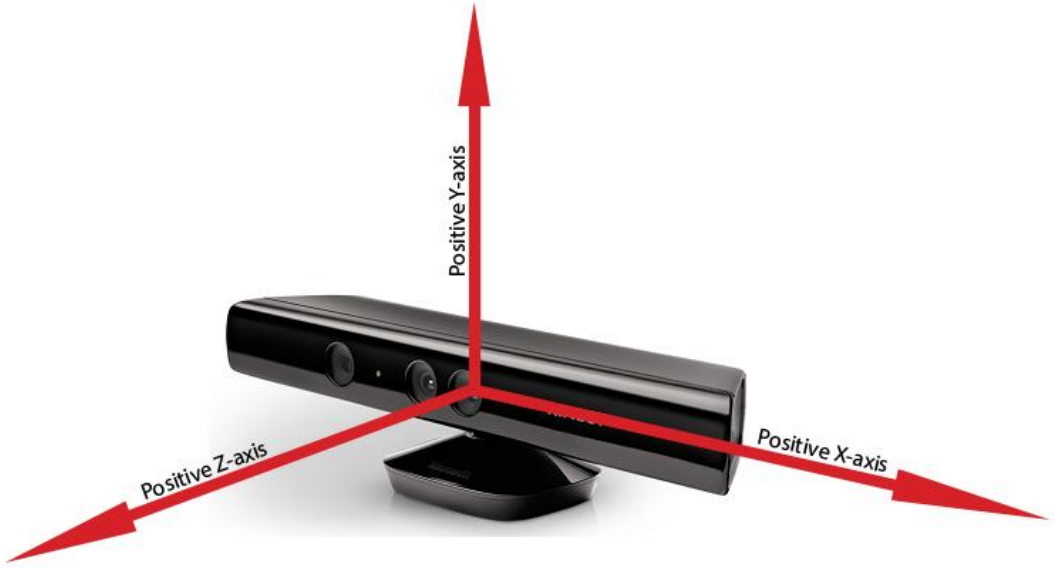

Figure 28 – The coordinate system of the Kinect. This is true for both the Kinect v1 and the Kinect v2.

With the use of trigonometry I was be able to calculate the length of the 'bone' between two joints and calculate the angle between two connected bones. To calculate the distance, or rather the length of the bone, v between two joints with x, y and z coordinates, I used the Pythagorean Theorem[25]:

$$|v| = \sqrt{(\Delta x)^2 + (\Delta y)^2 + (\Delta z)^2}$$

Using this formula I calculated the length of the two 'bones' that are connected to the joint of which we want know the angle, for example the knee. I also needed to calculate the distance between the two outer joints, the hip and ankle, which, together with the two bones, form a triangle of which all the sides are known. Since it is most likely that the triangle is often not right angled, I will use the Cosine Rule[26] to calculate the angle:

$$\cos(\alpha) = \frac{b^2 + c^2 - a^2}{2bc}$$

Figure 29 shows an example of the angle I wanted to calculate.

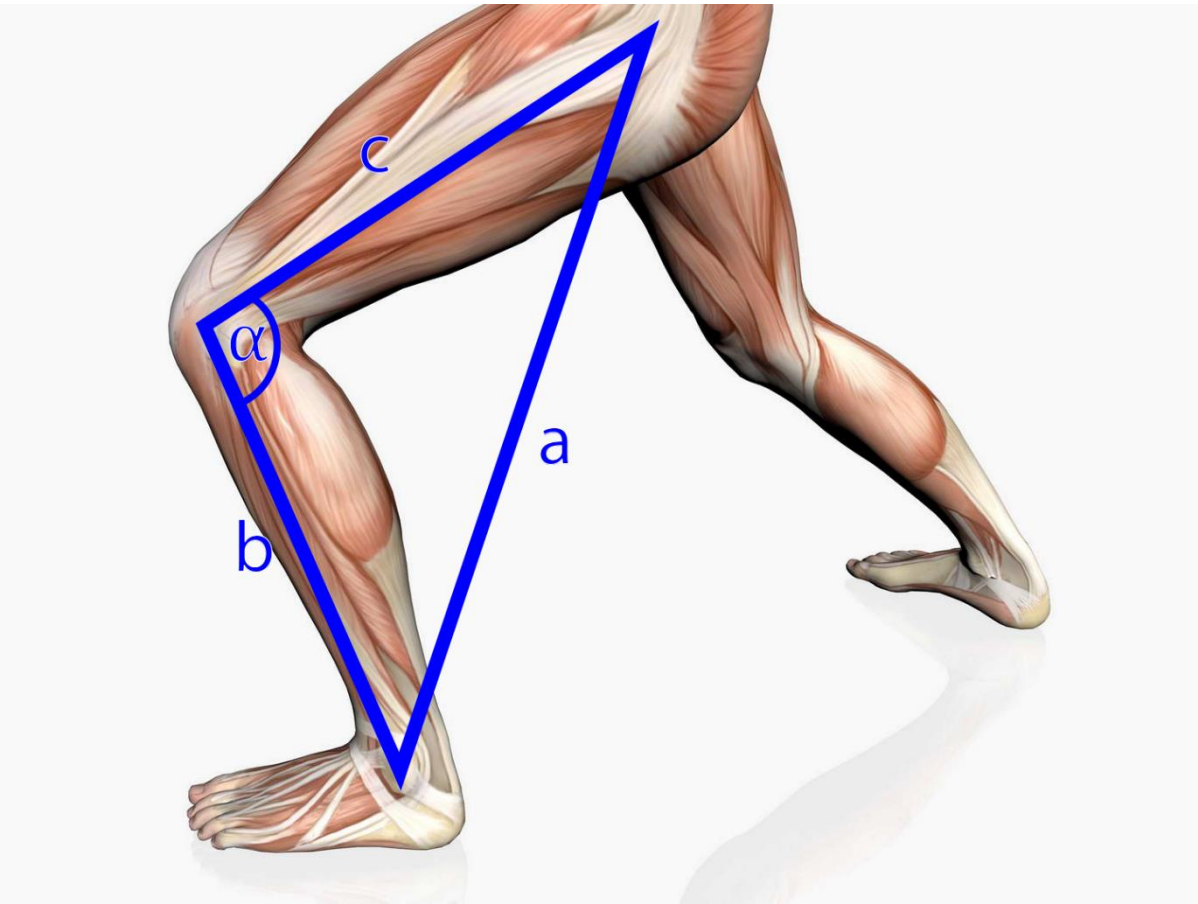

Figure 29 – The (imaginary) triangle used to calculate the angle of the left knee (alpha).

[25] http://www.altdevblogaday.com/2011/05/29/points-vertices-and-vectors/
[26] http://www.cimt.plymouth.ac.uk/projects/mepres/step-up/sect4/index.htm

Unfortunately the formulas generated results that I did not understand. Furthermore, since I wanted real-time feedback, the formulas were executed with every frame which visibly slowed down the application.

Using objects of the class type Vector3 might have been less resource heavy. The Vector3 class is defined in the DirectX and the XNA assemblies. Unfortunately neither of them were available for my situation. The DirectX libraries can only be used in C++ projects, whereas mine is a C# project. Since Visual Studio 2012 it is not possible to use the XNA framework within Visual Studio. It is possible to still reference XNA libraries through a little hack, but I would like something that works out of the box, hence the use of the Vector3D class which I got from a blog post[27].

5.4.1.2 Relative positions

After abandoning the angle approach and going for the positioning of the joints relative to each other, I implemented some conditions using the algorithmic approach for the air squat. During testing however I encountered some difficulties. When viewing the athlete from the front, the Kinect got confused when the athlete was down in the squat. It appeared that as soon as the upper legs disappeared behind the knees, the Kinect could not make out where the joints should be. As a result, the knee joints suddenly jumped to another, anatomically incorrect, position. This posed a problem since one of the constraints of the air squat is that the knees should keep their horizontal position as much as possible. More importantly, they are not allowed to fall inwards.

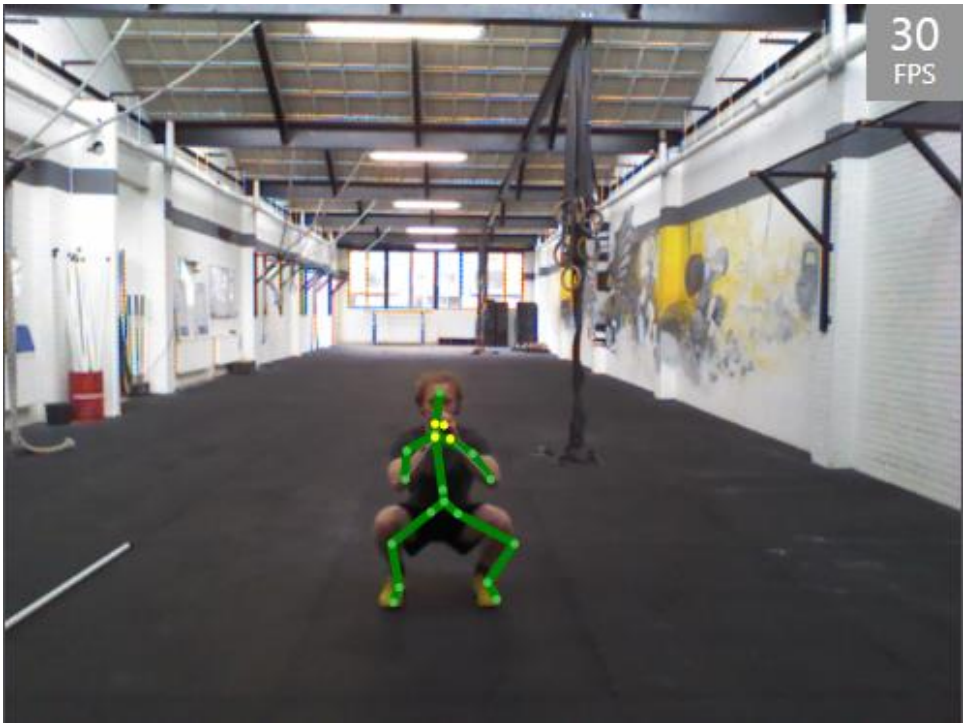

Figure 30 – The knees are not tracked correctly anymore

[27] http://social.msdn.microsoft.com/Forums/en-US/8516bab7-c28b-4834-82c9-b3ef911cd1f7/using-kinect-to-calculate-angles-between-human-body-joints?forum=kinectsdknuiapi

At first I suspected the cause to be that there was noise in the joint data. This is a common issue when developing Kinect applications. The SkeletonStream object, from which the frames with the skeleton data are retrieved, can receive parameters of the class TransformSmoothParameters to smooth out the resulting data. This is especially useful when the resulting skeleton appears jittery in the application. There is a tradeoff however between the amount of smoothing parameters applied and performance of the application. The heavier the smoothing parameters applied, the more resources it takes to apply the smoothing parameters and the slower the resulting skeleton will be rendered. Unfortunately the problem was not solved by applying the smoothing parameters. Although the skeleton moved smoother, the knee joints still jumped to positions outside the actual human body.

After reading (Azimi, 2012) I suspected that the problem was more a problem with accuracy than with precision since applying the smoothing parameters made placement of the joints more precise (less jitter) but did not solve the inaccuracy. The cause for this probably lies with the fact that the tracking state of the knee joints is inferred instead of actually tracked. The inferred state suggests that the Kinect sensor does not actually knows where the knees should be but instead makes an educated guess. The whitepaper (Azimi, 2012) does suggests that by applying Body Kinematics the knee joints might be forced in their position by imposing restrictions to how and where the joints can be positioned based on the limitations on how the human body can move.

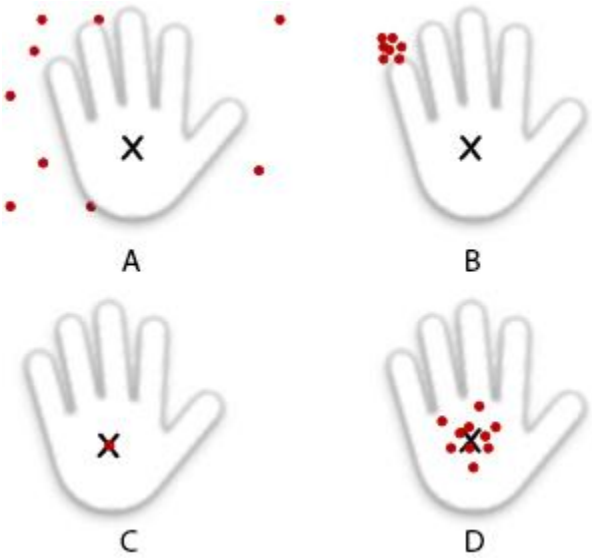

Figure 31 (Azimi, 2012)
A – Inaccurate and imprecise
B – Inaccurate but precise
C – Accurate and precise
D – Accurate and moderately precise

My biggest worry however is that if the actual data is inaccurate, by applying filters to make it look better I introduce the possibility that the user might seem to do the exercises better than he or she does in reality. In the end I abandoned the Kinect v1 in favor of the Kinect v2 due to better raw results.

5.4.2 Kinect v2

With the Kinect v2 I was able to receive better joint data due to the higher resolution of the new sensor. A quick comparison with the Kinect v1 showed that with the air squat the knees, and all other joints, stayed at their approximate positions. Unfortunately this did meant that I had to start over again since the clips that were recorded by Kinect Studio v1 are not compatible with the tools in the SDK v2. I did not record any reference material with the Kinect v2 at CrossFit Amsterdam yet. This because I have only recently been able to obtain this sensor that came out just 2 months before the time of this writing.

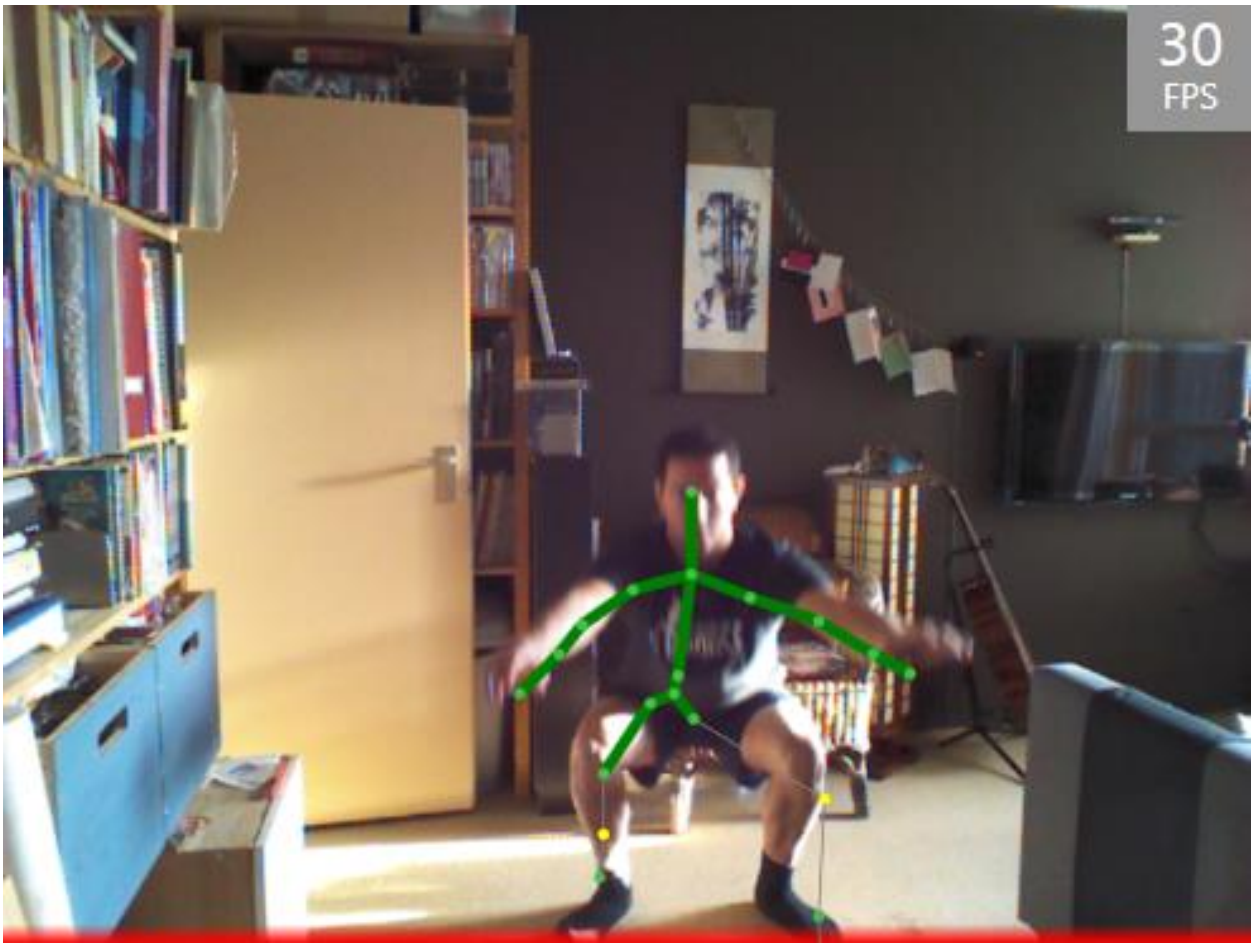

Figure 32 - Results with Kinect v1 and as you can see, the yellow dots indicate that the joints are being inferred, resulting in the joint going somewhere the knee is absolutely not.

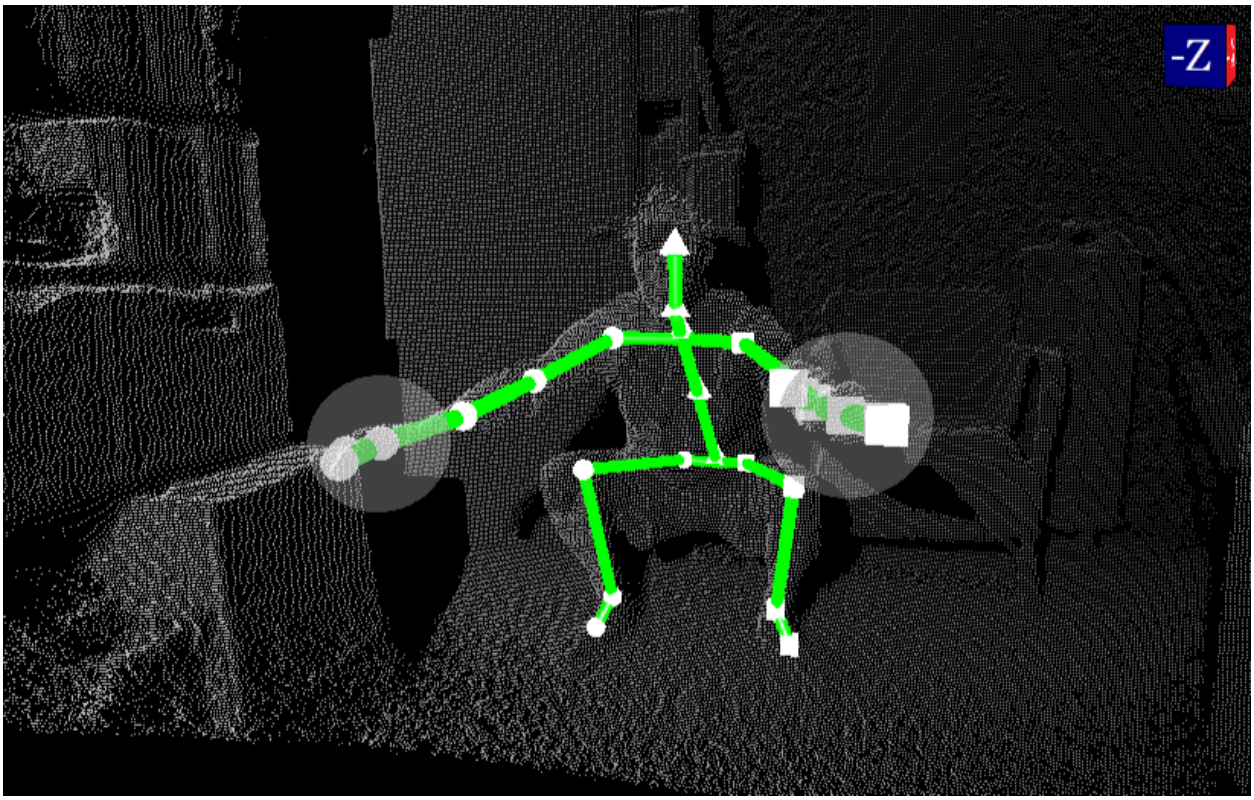

Figure 33 - Results with Kinect v2 and the joints are staying in place better than with the v1.

The SDK 2.0 comes with an improved Kinect Studio and a new tool called Visual Gesture Builder (VGB). With this tool it is now possible to create custom gestures quicker than with SDK 1.x. VGB offers two methods to create gestures. The first one is the discrete method using the algorithm AdaBoost[28] and the second one is the continuous method which applies Random Forest Regression[29] (RFR), both of which are machine learning algorithms. The difference between the two is that with the discrete method, I am able to mark which movements should return the Boolean value true. Everything else will return false. This method is well suited for relatively simple gestures of which you only need to know when they are executed or not. With the continuous method I can assign float values at certain (key) points of a movement and everything between those points automatically receives float values in a continuous fashion. This opens up the possibility to track a movement from beginning to end and at which point of the movement a user finds himself.

[28] http://en.wikipedia.org/wiki/AdaBoost
[29] http://en.wikipedia.org/wiki/Random_forest

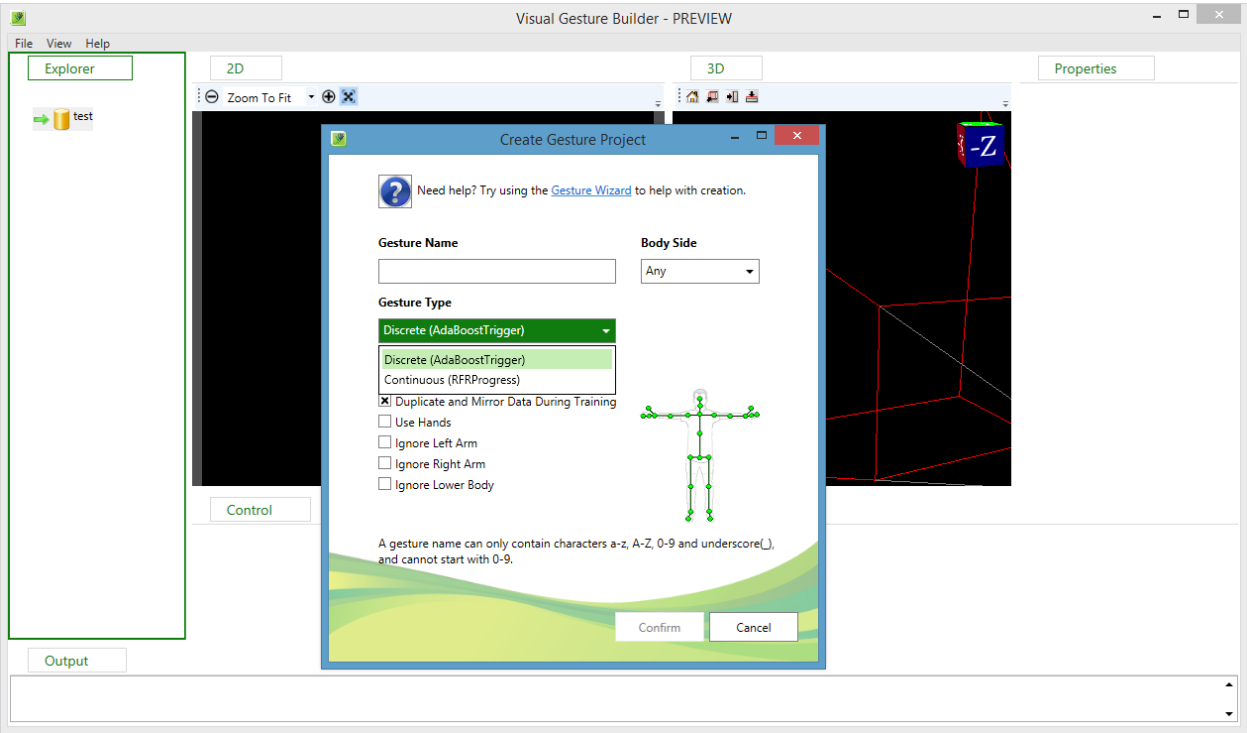

Figure 34 – Creating a new project in Visual Gesture Builder

To create gestures with VGB I first need to record a gesture with Kinect Studio v2. I then created a new project in VGB at which point I choose whether I wanted to apply the discrete method or the continuous method. The newly recorded video clips can then be imported in the project.

I recorded a test sample with myself as the test subject where I did a set of several air squats with the new Kinect Studio. A few I executed correctly and a lot of them I executed incorrectly in different ways. Having a lot of different bad samples is probably even more important than having good samples. Bad samples will help with determining when the air squat is executed incorrectly, even if only one part of the body does it wrong. I used these recordings in VGB where I choose AdaBoost to tag the frames of recording where I executed the air squat correctly. I then used the Discrete Gestures Basics sample from the SDK to test my compiled file and the application correctly returned true and false results most of the time. The VGB solution can be further enhanced by including more samples to the discrete project of the air squat, preferably with different test subjects so it can be trained to see how it looks when different body types execute the same exercise.

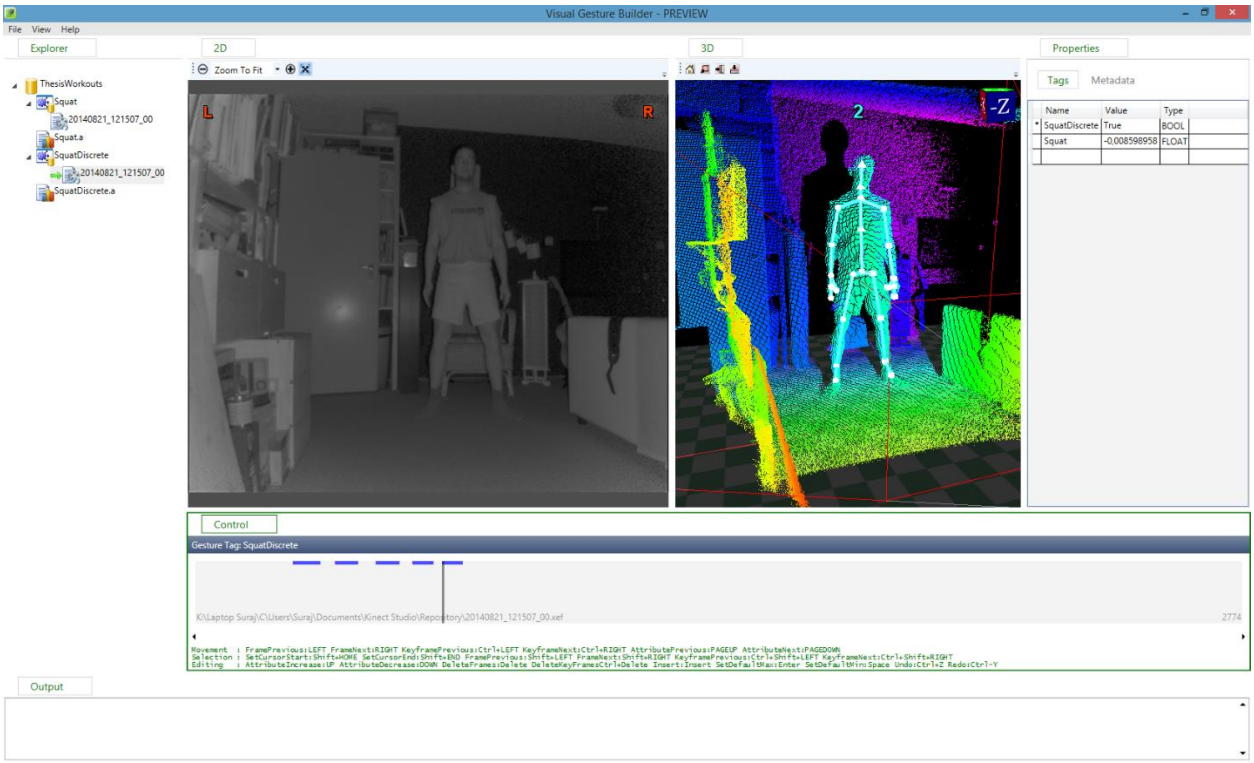

Figure 35 – In the control section, the purplish bars are the frames that are tagged true.

So now I was able to differentiate between correct and incorrect executions of the air squat but it was not possible to tell what I would do wrong if the application returned a false result. I added a continuous project to my existing VGB solution where I added floating point values at certain key frames. The start of the air squat was 0.0, the halfway point 0.5 and the end position was 1.0. This was only done for the ones that were marked as true with the discrete method. I edited the Discrete Gestures Basics sample to be able to retrieve the continuous data and present them through the interface. The interface showed the correct results when executing the air squat correctly but it still gave some kind of values between 0.0 and 1.0 when doing the air squat incorrectly. It appeared that VGB adds floating point values to every frame automatically.

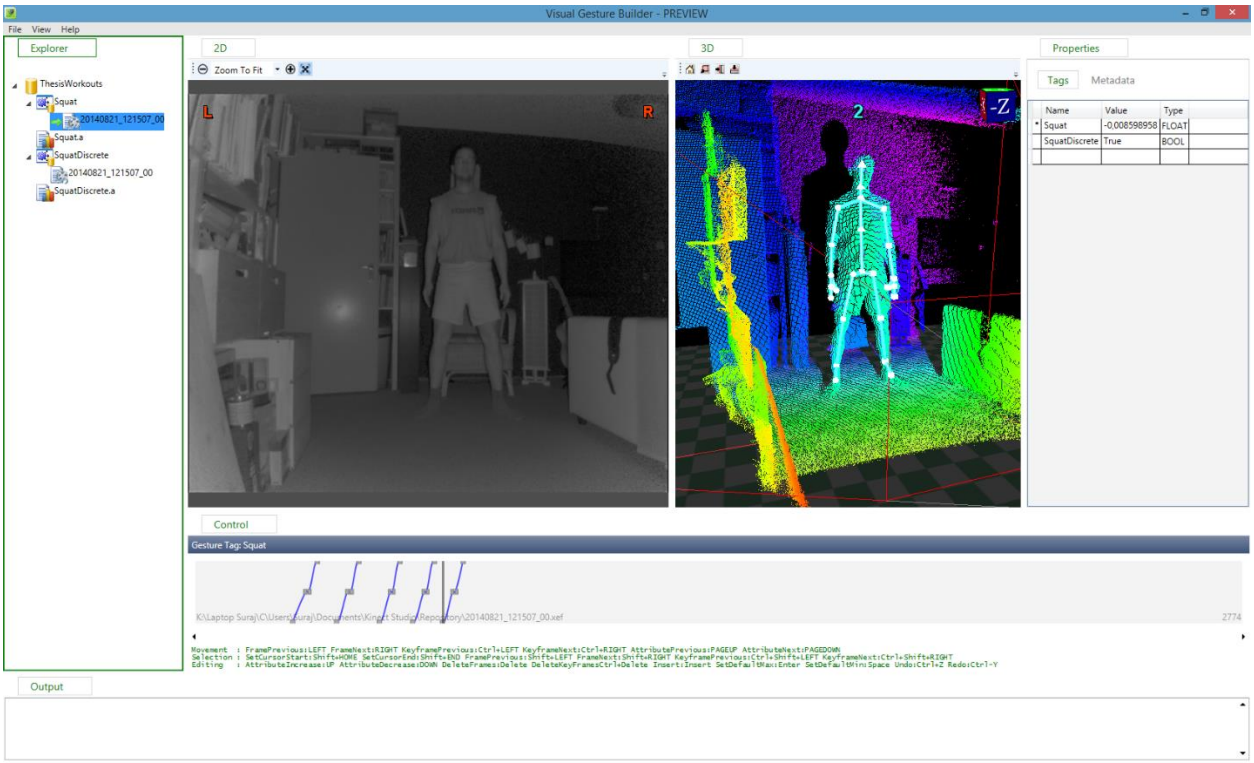

Figure 36 – In the control section you can see the frame that are tagged. The gray blocks are frames that have manually gotten floating point values inserted into them, the frames in between automatically receive values that lie between the two key frames and seem to be distributed in a Gaussian fashion.

5.5 Results

The Kinect v1 could unfortunately not deliver the quality I hoped for. This was mainly because as soon as joints start to come in front of the body, the sensor is not sure what to make of it and starts to guess where the inferred joints should be. The resulting skeleton is often not usable for the quality validation when executing an exercise, since the locations of the joints are not always accurate. This is not an uncommon problem with the Kinect v1 and in most cases, filters are used to prevent non-anatomical skeletons. Most of these filters help smooth out the jittery animation of the skeleton and make the locations of the joints more precise, but (Azimi, 2012) also suggest the use of body kinematics to help make it more accurate. My biggest concern with applying this method is that it might smooth out the skeleton too much and therefor correct incorrectly executed exercises.

The second generation of the Kinect sensor provides data with higher precision and accuracy, even when joints are not distinctively visible. The new SDK also provides an improved Kinect Studio and the new Visual Gesture Builder with which it is easier and quicker to create complex gestures. The initial results were promising. The air squat was correctly being validated when it was done correctly or not. Unfortunately, since the Kinect v2 has only been recently available it was not possible to develop the prototype any further with it due to time constraints.

I also conducted a brief experiment to show the difference in data quality between the Kinect v1 and v2. The results can be seen in figure 37. I tracked the left knee during two squats and its motion over all three axis. As you can see, the Kinect v2 does a much better job at keeping track of the knee than the v1.

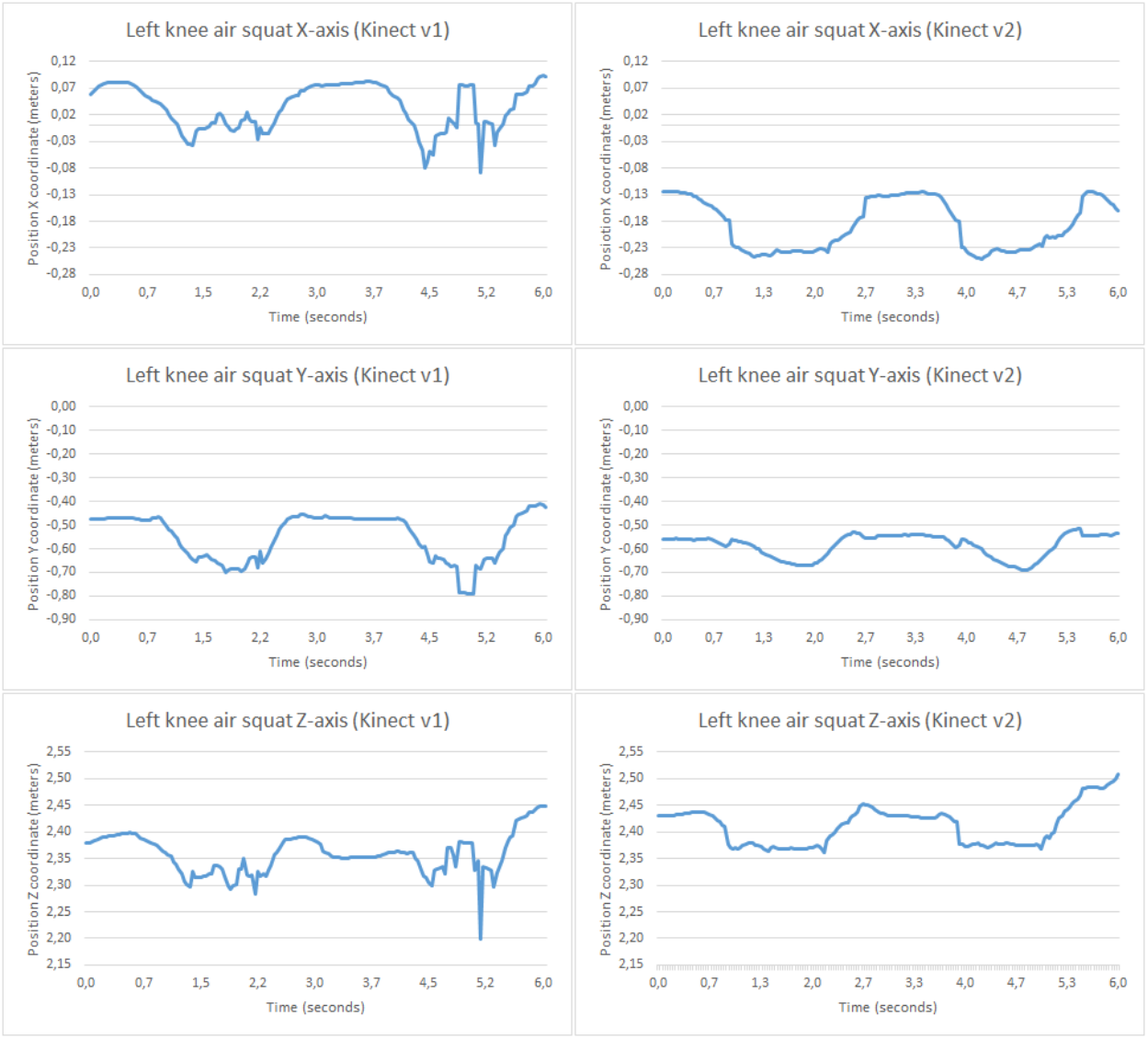

Figure 37 - Theses graphs show the position of the left knee joint along each of the axis during two air squats. It clearly shows that the Kinect v2 tracks the joint more smoothly than the Kinect v1.

6 Evaluation and discussion

This project did not turn out the way I expected it to be. But that is not uncommon during research. In this chapter I have evaluated and discuss the results of the project. Mainly the problems I have encountered, inaccurate data and difficult gesture creation, and their cause. The chapter will conclude with how this project could be continued using the Kinect v2.

6.1 Inaccurate data

The project ended up to be a research into the quality of the Kinect sensors for the validation of exercises. In this case it is not just important to see whether a repetition is being completed but also if it is done correctly. The quality of the raw data of the Kinect v1 was not good enough due the noise in the data. Another problem was that the human body could not be properly recognized due to other body parts obstructing some joints. In these cases these joints had to be inferred. This issue with quality was also determined by (Choppin, 2011; Obdrzalek et al., 2012; Ochi, 2012). It might be possible to filter this out but by introducing filters that "correct" the data it becomes harder to determine whether the filtered data is still useable to validate the exercises correctly.

There are applications, or rather exergames, which use the Kinect v1 for Xbox to help the user with his or her exercises, as seen in chapter 3. The best example is probably the Nike+ Kinect Training game for the Xbox 360[30]. The air squat is one of the many exercises present in this game and it requires the user to look at the sensor which means the sensor has a front view of the user. It is able to correctly determine when a squat is successful or not, but it does not give a great amount of detail what you did wrong when it determines you executed a repetition incorrectly. At most it will say you did not go deep enough but it will not look at your posture in general. This while posture is an important aspect of exercising (Fitzgerald et al., 2007; Zhou & Hu, 2008).

Another approach I intended to try to improve the quality of the skeleton data from the Kinect v1, was to use 2 sensors and combine the data of both. The idea behind this was that if one sensor was inferring one or more joints, the data from the other sensor could be used if that one was able to track those joints. The combine data would then consist the true positions of the joints because they would be visible by at least one sensor. The first issue with this approach is the interference between the two sensors. Each Kinect emits infrared light in order to measure the depth data. Unfortunately, the pattern that each sensor emits is the same and the infrared laser is not modulated. Which means that if there is more than one sensor in the room and they have overlapping views, they are unable to discern which infrared dot belongs to themselves. The Microsoft Research division solved this problem with Shake'n'Sense (Butler et al., 2012). With this method a small motor, such as in a pager, is stuck onto one of the two sensors. The vibration causes the pattern of the vibrating sensor to differ from the non-vibrating one. This way both sensors are able to discern their own infrared pattern.

When using two or more Kinect sensors, I had the choice between either handle each data stream individually or try and combine the multiple data streams into one. The first option would not solve my accuracy problem. Even though either sensor would be able to track different parts of the body, the

[30] http://en.wikipedia.org/wiki/Nike%2B_Kinect_Training

positions of the joints would be different between the sensors because each uses its own coordinate system. Combing data streams is not something that is supported by the SDK and is a research project on its own. There was a master student whom succeeded in combining the skeleton data from multiple Kinects by sending the data to a server where the data is combined (Wittenberg, 2013). But although he described how the decision is made which sensor' joint data is used, it was not made clear how the skeleton was reconstructed. The reason this part was important to me was because the joint positions and rotations in 3D space is relative to each sensor and I had no idea at that moment how to align the positions and rotations of the joints from both sensors. What I could have tried in hindsight was to designate one of the sensors as the main sensor and look at the joint positions and orientation of that sensor. Then if one of the joints is better tracked by the additional sensor I would need to transform that joint to align it with the joints of the main sensor.

With the Kinect v2, the infrared interference does not exist anymore since a modulated infrared beam sensor is used instead of an infrared grid. On the other side, the SDK v2 does not support multiple Kinect sensors. So using multiple Kinect sensors is out of the question at the moment.

Thankfully the Kinect v2 the depth sensor has a higher resolution than the v1, from 320 x 240 to 512 x 424. Additionally, the quality of the data does not depend on the lighting conditions of the environment anymore. Since the Kinect v2 filters all the light in the room to make the infrared images, which are used for skeleton tracking, look the same.

6.2 The difficulty of creating gestures

The approaches for creating the gestures to validate the exercises are either not sufficient (the basic and algorithmic approach) or too complex to apply by myself (the neural network and template-based approach). The reasons why the basic and algorithmic approach are not sufficient is because it is very labor intensive to determine the correct values of the positions of the joints relative to each other, that will correctly validate the exercises for every body type. I would either have to find out the greatest common divisor for all the necessary values or the values should be determined for every body type specifically, and then store them. In the last case, every time a new user would want to use the application, the body type would need to be selected. A neural network on the other hand was simply too complex to apply. In section 5.3.1.1 I have briefly explained how this works and that it lies in the realm of Artificial Intelligence and fuzzy logic. Both of which fall outside my field of expertise and outside the scope of this project. (Jana, 2012) also mentions that because neural network is very complex and hard to debug, which is why the majority of the Kinect developers do not bother using it. It is also why there are not many examples of Kinect applications that use this approach. If I wanted to apply the template-based approach I would have to create an application first to create these templates. That on its own could be a separate project. It was not until recently that I discovered that Carl Franklin created such an application, called GesturePak[31], and sells it online. By that time I already decided that I would use the Kinect v2 for this project.

[31] http://www.franklins.net/gesturepak.aspx

Creating complex gestures has improved with the Kinect v2. Kinect Studio has improved in usability. Not only to make recordings but for development as well. Now it is not required to have a sensor attached anymore to play back recordings. But more significant is the addition of the Visual Gesture Builder. With this application it is possible to import recordings made with Kinect Studio and tag the frames in two manners. You can either tag a range of frames as true, where the untagged frames are false by default. Or the outer frames of the range can be tagged with floating point values where the frames in-between will automatically get a floating point value that lies between those two tagged values. Through machine learning Visual Gesture Builder compiles database files that contain the information on when a gesture is true or not, by using AdaBoost[32], or when it should return certain floating point values by using Random Forest Regression[33]. The more recordings are used for a gesture, the better the gesture will be correctly recognized, provided the recordings are well tagged.

6.3 Future work

Creating a gesture that needed to return a true or false value was very simple using Visual Gesture Builder, which I did for the air squat. However, I also wanted to be able to track the progress of a repetition and identify the type of mistake the user made when false is returned. Since this is a new approach of creating gestures, there were no examples available and I was not able to test an approach that worked before finishing this thesis. Currently I have created a database file that was compiled from a project where I tagged the air squat using AdaBoost as well as Random Forest Regression. The prototype presents both the Boolean value as the floating point values to the user interface. The Boolean part works as intended but the Random Forest Regression seems to always return some floating point value, even if I did not tag the frame. What I could try is to tag the whole recording and assign certain values to certain movements. I had the following in mind:

- Tag every frame that contains the starting posture with the value 0.0.
- Tag every air squat starting with 0.0 and ending with 1.0. The halfway point, where the user reaches minimum depth, should be tagged as 0.5.
- Assign each type of mistake a floating point value that is higher than 1.0, for example:
    - Feet wide, knees normal = 2.0.
    - Feet wide, knees inward = 3.0.
    - Feet small, knees normal = 4.0.
    - Feet small, knees inward = 5.0.
    - Feet normal, knees inward = 6.0.
- The moment the user starts to make one of the above mistakes, tag that first frame with the value that automatically got assigned to it, then tag the next frame till the last frame of that mistake with the assigned value. This way, even if a user starts out correctly, the mistake will only be registered as soon it is being made.
- The repetition will be marked as no-rep and the user will have to get into the starting posture to start the new repetition.

[32] http://en.wikipedia.org/wiki/AdaBoost
[33] http://en.wikipedia.org/wiki/Random_forest

So although I have not tried this approach yet, this might solve the issue of recognizing which type of mistake is being made during the exercise.

7 Conclusion

In this final chapter I have concluded this thesis by answering the three research questions I posed in chapter 1. This chapter will end with a reflection on my experiences during this project.

7.1 Research questions answered

How can motion tracking be applied as a cost effective solution to sports and healthcare?

Of the three research questions this one is the easiest to answer. The Microsoft Kinect sensor is at the moment the most affordable piece of hardware for motion tracking that can be applied to people. Also, since the Kinect sensor tracks people by using a depth sensor, it is enough to use only one such sensor in most cases. This compared to an array of cameras when using some other motion capture systems. Furthermore, all the tools needed to develop Kinect enabled applications, the Kinect SDK and Visual Studio Express, are free.

How should motion tracking data be analyzed to provide recommendations for posture correction in real-time?

In order to provide real-time recommendations for posture correction during exercises, it is necessary to use, what Microsoft calls, gestures. By using a gestures you are essentially implementing a set a rules for each exercise. Every repetition of an exercise must abide by these rules for the execution to be correct. When a repetition fails to follow one or more of these rules, depending on which rule was broken, an appropriate recommendation should be returned. Thus, when good gestures have been created, motion tracking data can be analyzed using the created gestures as reference.

Creating gestures with the Kinect v1 could only be done manually. With the Kinect v2 this has been significantly improved with the introduction of the Visual Gesture Builder application.

What are the essential elements for a framework of motion and posture correction for CrossFit exercises?

Unfortunately I was not able to delve into this question during this project due to the change of course in the research. But I can still answer the question with the information I have. The most essential element for a framework that can be used for motion and posture correction is having high quality gestures. If the gesture is not of sufficient quality, the chances of determining the execution quality of an exercise correctly will decrease. By having a high quality gesture, it is possible to correctly determine when an exercise is executed correctly or not. But to motivate the user to correct his or her posture when it is determined as incorrect requires two other important elements. The first is the capability of the framework to give feedback to the user. This feedback comes in two forms:

1. Visual and/or audio cues during exercising when a repetition is done incorrectly and how to correct it.
2. An overview of all previous exercises and how they relate to each other, or in other words, to see whether the user is improving or not.

The second element is for the user to be able to set well defined, short-term goals. For example to be able to execute 30 air squats without one being executed incorrectly within one month. Both goal setting and receiving feedback have shown to motivate users to adhere to their exercises better (Leslie et al., 1999; Locke & Latham, 1985; Martin et al., 1984).

7.2 Reflection

With the Kinect sensor, Microsoft has made sophisticated motion tracking affordable. By also actively supporting the community that started using the first Kinect for Xbox for anything else but the Xbox console, the possibilities became endless. I started this project with no experience about programming with the language C# nor Visual Studio. My intention was to create an application that would help me and other people to do their exercises with the correct posture, without the need of having a human expert present. This would not only apply to CrossFit but to exercising in general. This could be related to sports or healthcare. It would also mean that coaches and physicians are better able to treat their athletes and patients more effectively. By tracking their exercises and collecting that data with the application, a coach or physician would have an overview of the progress of the user and see where possible problems might be.

Unfortunately I underestimated the difficulty of creating such an application. While development in Visual Studio and with C# is fairly simple, and creating simple Kinect applications is not very hard, it's the creation of gestures that is difficult. As I described in section 5.3.1.2, creating gestures for the Kinect v1 is largely done by manually coding all the rules a gesture should follow. This means that every exercise would have to be broken down into basic elements where:

- The positions of the joints relative to each other are determined at both key points of the movement to determine how far into a repetition the user is
- And during the whole movement itself to determine the constraints within which the positions should be at all times.

To make matters worse, the accuracy of the Kinect v1 sensor is not very high. Large, general movements or not a problem. But when small shifts in the positions of the joints can make the difference between a correctly executed or incorrectly executed exercise, the Kinect v1 is not accurate enough. This "low" accuracy was caused by the low resolution of the depth sensor, the dependence on an ideally lit environment and the inability to track joints when they are in front of other parts of the body. All of these created noise in the data of the positions of the joints which were sent by the sensor. Theoretical filtering algorithms were suggested in literature but those would alter the resulting skeleton to make it look more anatomical, introducing the possibility that the results might not correspond with what the user actually does.

Luckily I was able to get my hands on the new Kinect sensor which was improved on every front compared to the first sensor. The accuracy has been improved significantly, as can been seen in section 5.4.2. But creating gestures has also been made easier with the addition of Visual Gesture Builder. Unfortunately I ran out of time to create a whole new application with the new Kinect but in section

It is not hard to imagine that the course of this project has changed during this research. From trying to create a prototype to help people do their exercises with the correct posture, to an exploration into the limitations of the Kinect v1 and how the Kinect v2 has been improved concerning those limitations. This is why this thesis does not contain any user study or any real evaluations except for the accuracy comparison between the first and new Kinect sensors. Which might be an interesting subject matter to delve into in the future. Luckily all was not lost. During this project I have gained a lot of experience working with both the old and new Kinect sensors and with programming with C# in Visual Studio. Visiting the Kinect Hackathon[34] in Amsterdam greatly added to this experience as well, having met with some of the people of the Microsoft Kinect team and other Kinect developers from Europe. And now that I know that what I set out to achieve might be possible with the new Kinect sensor I will continue working on this project after having finished this thesis.

[34] http://blogs.msdn.com/b/kinectforwindows/archive/2014/09/19/hackers-display-their-creativity-at-amsterdam-hackathon.aspx

8 Acknowledgement

I would like to acknowledge CrossFit Amsterdam for providing the reference recordings of the three movements and for their expertise on correct and incorrect postures during these movements.

Bibliography

Aminian, K., & Najafi, B. (2004). *Capturing human motion using body-fixed sensors: outdoor measurement and clinical applications*. *Computer Animation and Virtual Worlds, 15*(2), 79-94.

The paper has described several studies in which kinematic sensors were used to measure the motions of the human body. These studies show that it is possible to conduct motion tracking without the need of very expensive equipment such has specialized motion tracking cameras and a specially prepared space. This shows that motion tracking is becoming available for a larger audience.

Archambault, P. S., Norouzi, N., Kairy, D., Solomon, J. M., & Levin, M. F. (2014). Towards Establishing Clinical Guidelines for an Arm Rehabilitation Virtual Reality System. In W. Jensen, O. K. Andersen & M. Akay (Eds.), *Replace, Repair, Restore, Relieve – Bridging Clinical and Engineering Solutions in Neurorehabilitation* (Vol. 7, pp. 263-270): Springer International Publishing.

This is an evaluation of the VR rehabilitation system developed by Jintronix. The objectives of this evaluation were to determine which activities and levels of difficulty are appropriate for rehabilitation of arm movements in stroke patients with different degrees of motor impairment. And to determine the ease of use and subjective experience of patients using the VR arm rehabilitation system.

Azimi, M. (2012). Skeletal Joint Smoothing White Paper. Retrieved from http://msdn.microsoft.com/en-us/library/jj131429.aspx This white paper discusses the issues with noise in the joint data and presents a collection of filters that could be applied to reduce this noise.

Butler, D. A., Izadi, S., Hilliges, O., Molyneaux, D., Hodges, S., & Kim, D. (2012). *Shake'n'sense: reducing interference for overlapping structured light depth cameras*. Paper presented at the Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. This paper presents a novel yet simple technique that mitigates the interference caused when multiple structured light depth cameras point at the same part of a scene.

Choppin, S. (2011). Kinect Biomechanics: Part 1. *Kinect Biomechanics*, from http://engineeringsport.co.uk/2011/05/09/kinect-biomechanics-part-1/

The first of a series of articles which will look at different possible applications of the Kinect in the field of Biomechanics. This first article compares the accuracy of determining the angle of the joints by the Kinect with a traditional analyzing tool.

Fitzgerald, D., Foody, J., Kelly, D., Ward, T., Markham, C., McDonald, J., et al. (2007, 22-26 Aug. 2007). *Development of a wearable motion capture suit and virtual reality biofeedback system for the instruction and analysis of sports rehabilitation exercises.* Paper presented at the Engineering in Medicine and Biology Society, 2007. EMBS 2007. 29th Annual International Conference of the IEEE.

During this case study the authors described their theories and the work they did during the developement of an exercise monitoring tool. Their tool uses motion capture technology to track the technique and posture of the athlete and provide automated cues when the athlete

executes and exercise incorrectly and how to do it right. This is important in order to prevent injury or to recover from one.

Ford, K. R., Myer, G. D., & Hewett, T. E. (2003). Valgus knee motion during landing in high school female and male basketball players. *Med Sci Sports Exerc, 35*(10), 1745-1750.

The purpose of this study was to utilize three-dimensional kinematic (motion) analysis to determine whether gender differences existed in knee valgus kinematics in high school basketball athletes when performing a landing maneuver.

Franklin, E. N. (2012). *Dynamic Alignment Through Imagery*: Human Kinetics Publishers.

The author of this book is a teacher, dancer, and choreographer. In his book he explains the importance of using the correct posture and techniques when moving, whether with sports or dance, and how this benefits your spine and back and prevents injuries. He does this by providing a large collection of illustration based exercises.

Fuhr, L. (2013). What the Heck Is This "CrossFit Disease" I'm Hearing About Everywhere? , from http://www.fitsugar.com/What-CrossFit-Disease-32016950

An short article about what rhabdomyolysis is and that it is not exclusively related to CrossFit. The article ends with some notes from Yumi Lee, co-owner of Reebok CrossFit LAB and an internationally renowned fitness expert, explaining how you can prevent injuries by using the correct form during exercises.

Goyanes, C. (2013). The 12 Biggest Myths About CrossFit. from http://www.shape.com/fitness/workouts/12-biggest-myths-about-crossfit

A short web article discussing some commen misconceptions about CrossFit and specifically from the point of view of women. Within the article Yumi Lee, co-owner of Reebok CrossFit LAB and an internationally renowned fitness expert, is often quoted to offer some insights about exercising responsibly.

Graci, V., Van Dillen, L. R., & Salsich, G. B. (2012). Gender differences in trunk, pelvis and lower limb kinematics during a single leg squat. *Gait Posture, 36*(3), 461-466.

The aim of this study was to investigate the existence of different multi-joints movement strategies between genders during a single leg squat.

Jacobs, C. A., Uhl, T. L., Mattacola, C. G., Shapiro, R., & Rayens, W. S. (2007). Hip abductor function and lower extremity landing kinematics: sex differences. *J Athl Train, 42*(1), 76-83.

The researchers of this paper tried to evaluate sex differences in hip abductor function in relation to lower extremity landing kinematics.

Jana, A. (2012). *Kinect for Windows SDK Programming Guide*: Packt Publishing, Limited.

Abhijit Jana is a .NET consultant at Microsoft and wrote this book in which he discusses the technical details of the Kinect and how its features are utilized with the Software Development Kit.

Leslie, E., Owen, N., Salmon, J., Bauman, A., Sallis, J. F., & Lo, S. K. (1999). *Insufficiently active Australian college students: perceived personal, social, and environmental influences*. *Preventive Medicine, 28*(1), 20-27.

In this article the authors performerd a research among college students to find out if and why they are or are not physically active. It is during this phase in life that a large number of people go from an active life style to a sedentary one. It turns out that social support from family and friends have a positive effect on the participation in physical activities.

Locke, E. A., & Latham, G. P. (1985). *The Application of Goal Setting to Sports*. *Journal of Sport Psychology, 7*(3), 205-222.

Locke and Latham are two of the most prominent researchers of the goal setting theory. This theory was developed and refined by Locke in the 1960s. This work helps me explain why goal setting is important and beneficial to keep adhering to your exercise schedule.

Martin, J. E., Dubbert, P. M., Katell, A. D., Thompson, J. K., Raczynski, J. R., Lake, M., et al. (1984). *Behavioral-Control of Exercise in Sedentary Adults - Studies 1 through 6*. *Journal of Consulting and Clinical Psychology, 52*(5), 795-811.

The authors conducted a series a six studies to research various training methods which could help prevent people from stopping stopping with their exercise program. They concluded that social support such as instructor feedback and praise, and flexible goal setting by the participant are important factors that can help people adhere to their exercise programs.

Microsoft. (2013). Human Interface Guidelines v1.8.

A guide with tips on how to create good interfaces when using the Kinect as user input.

Obdrzalek, S., Kurillo, G., Ofli, F., Bajcsy, R., Seto, E., Jimison, H., et al. (2012). *Accuracy and robustness of Kinect pose estimation in the context of coaching of elderly population.* Paper presented at the Engineering in medicine and biology society (EMBC), 2012 annual international conference of the IEEE.

A comparision of the Kinect pose estimation (skeletonization) with more established techniques for pose estimation from motion capture data, where the accuracy of joint localization and robustness of pose estimation with respect to the orientation and occlusions are examined.

Ochi, Y. (2012). Development of Air-squat Supporting System using Microsoft Kinect.

This paper descibes the development and operation verification of a Kinect application to provide support during the air squat.

Paine, M. J., Uptgraft, M. J., & Wylie, M. R. (2010). CrossFit study. *Command and General Staff College*, 1-34.

The purpose of this study is to test the efficacy of the CrossFit fitness program andmethodology to increase the physical fitness of U.S. Army Soldiers. This paper provides, amongst others, an explanation about what CrossFit is.

Reflexion Health, I. (2013). *Reflexion Health's Rehabilitation Measurement Tool is Advancing Physical Therapy with Kinect for Windows*: Microsoft.

This PDF outlines the problem Reflexion Health tries to solve with their Rehabilitation Measurement Tool. In this case study they also provide the features their solution contains.

Tseng, Y., Wu, C., Wu, F., Huang, C., King, C., Lin, C., et al. (2009, 18-20 May 2009). *A Wireless Human Motion Capturing System for Home Rehabilitation.* Paper presented at the Mobile Data Management: Systems, Services and Middleware, 2009. MDM '09. Tenth International Conference on.

The authors of the paper demonstrate the use of small sensors that can be used to track the motions of humans for use in home rehabilitation. They discuss the need to use the correct posture and technique during excersises and how such systems can lighten the load of therapists and make rehabilitation less painful and boring for the patients.

Wang, S. (2003). Artificial Neural Network *Interdisciplinary Computing in Java Programming* (Vol. 743, pp. 81-100): Springer US.

An introductory chapter to artificial neural networks.

Wittenberg, I. (2013). *Performance and Motion Capture using Multiple Xbox Kinects.* Rochester Institute of Technology.

A thesis describing the combination of data from multiple Kinects to improve the accuracy of the skeleton data.

Zazulak, B. T., Ponce, P. L., Straub, S. J., Medvecky, M. J., Avedisian, L., & Hewett, T. E. (2005). Gender comparison of hip muscle activity during single-leg landing. *J Orthop Sports Phys Ther, 35*(5), 292-299.

In this paper the researchers describe the results of their study in which they tried to determine whether gender differences in electromyographic (EMG) activity of hip-stabilizing muscles are present during single-leg landing.

Zhou, H., & Hu, H. (2008). *Human motion tracking for rehabilitation—A survey*. *Biomedical Signal Processing and Control, 3*(1), 1-18.

In this paper the authors review a large collection of different motion tracking technologies and how effective they are for physical rehabilitation programs. They discuss the usefulness of monitoring a patient during rehabilitation can help the patient doing the exercises with the correct posture and technique. This way the patient receives an effective rehabilitation program and decreases the chance of getting injured.

Appendix A - SkeletonRecorder2

This application was used to record the movements with the Kinect v1 at the beginning and to output the X, Y and Z positions of the left knee during two air squats and compare these with those recorded with the Kinect v2.

MainWindow.xaml

```
<Window x:Class="SkeletonRecorder2.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Width="1280" Height="1024">
    <Grid  Width="1280" HorizontalAlignment="Center" Margin="10,0,0,0">
        <Grid.RowDefinitions>
            <RowDefinition Height="480"></RowDefinition>
            <RowDefinition Height="100"></RowDefinition>
            <RowDefinition Height="100"></RowDefinition>
            <RowDefinition Height="100"></RowDefinition>
            <RowDefinition Height="100"></RowDefinition>
            <RowDefinition Height="100*"></RowDefinition>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <Image x:Name="ColorImage" Grid.Row="0" Grid.Column="3" Grid.ColumnSpan="6"/>
        <Image x:Name="DepthImage" Grid.Row="0" Grid.Column="3" Grid.ColumnSpan="6"
Visibility="Hidden"/>
        <Grid x:Name="LayoutRoot" Grid.Row="0" Grid.Column="3" Grid.ColumnSpan="6"
Background="Transparent" >
        </Grid>
        <Label Grid.Row="1" Grid.Column="0" Grid.RowSpan="2" Content="Rotations"
FontSize="20" VerticalAlignment="Center" HorizontalAlignment="Center"/>
        <GroupBox Grid.Row="1" Grid.Column="1" Header="Head">
            <TextBlock Name="rotationHead" Text="Test" TextAlignment="Center"
VerticalAlignment="Center" HorizontalAlignment="Center"/>
        </GroupBox>
        <GroupBox Grid.Row="1" Grid.Column="2" Header="Shldr Center">
            <TextBlock Name="rotationShoulderCenter" Text="Test" TextAlignment="Center"
VerticalAlignment="Center" HorizontalAlignment="Center"/>
        </GroupBox>
        <GroupBox Grid.Row="1" Grid.Column="3" Header="Shldr Left">
            <TextBlock Name="rotationShoulderLeft" Text="Test" TextAlignment="Center"
VerticalAlignment="Center" HorizontalAlignment="Center"/>
        </GroupBox>
        <GroupBox Grid.Row="1" Grid.Column="4" Header="Elbow Left">
```

```
            <TextBlock Name="rotationElbowLeft" Text="Test" TextAlignment="Center"
VerticalAlignment="Center" HorizontalAlignment="Center"/>
        </GroupBox>
        <GroupBox Grid.Row="1" Grid.Column="5" Header="Wrist Left">
            <TextBlock Name="rotationWristLeft" Text="Test" TextAlignment="Center"
VerticalAlignment="Center" HorizontalAlignment="Center"/>
        </GroupBox>
        <GroupBox Grid.Row="1" Grid.Column="6" Header="Hand Left">
            <TextBlock Name="rotationHandLeft" Text="Test" TextAlignment="Center"
VerticalAlignment="Center" HorizontalAlignment="Center"/>
        </GroupBox>
        <GroupBox Grid.Row="1" Grid.Column="7" Header="Hip Left">
            <TextBlock Name="rotationHipLeft" Text="Test" TextAlignment="Center"
VerticalAlignment="Center" HorizontalAlignment="Center"/>
        </GroupBox>
        <GroupBox Grid.Row="1" Grid.Column="8" Header="Knee Left">
            <TextBlock Name="rotationKneeLeft" Text="Test" TextAlignment="Center"
VerticalAlignment="Center" HorizontalAlignment="Center"/>
        </GroupBox>
        <GroupBox Grid.Row="1" Grid.Column="9" Header="Ankle Left">
            <TextBlock Name="rotationAnkleLeft" Text="Test" TextAlignment="Center"
VerticalAlignment="Center" HorizontalAlignment="Center"/>
        </GroupBox>
        <GroupBox Grid.Row="1" Grid.Column="10" Header="Foot Left">
            <TextBlock Name="rotationFootLeft" Text="Test" TextAlignment="Center"
VerticalAlignment="Center" HorizontalAlignment="Center"/>
        </GroupBox>
        <GroupBox Grid.Row="2" Grid.Column="1" Header="Spine">
            <TextBlock Name="rotationSpine" Text="Test" TextAlignment="Center"
VerticalAlignment="Center" HorizontalAlignment="Center"/>
        </GroupBox>
        <GroupBox Grid.Row="2" Grid.Column="2" Header="Hip Center">
            <TextBlock Name="rotationHipCenter" Text="Test" TextAlignment="Center"
VerticalAlignment="Center" HorizontalAlignment="Center"/>
        </GroupBox>
        <GroupBox Grid.Row="2" Grid.Column="3" Header="Shldr Right">
            <TextBlock Name="rotationShoulderRight" Text="Test" TextAlignment="Center"
VerticalAlignment="Center" HorizontalAlignment="Center"/>
        </GroupBox>
        <GroupBox Grid.Row="2" Grid.Column="4" Header="Elbow Right">
            <TextBlock Name="rotationElbowRight" Text="Test" TextAlignment="Center"
VerticalAlignment="Center" HorizontalAlignment="Center"/>
        </GroupBox>
        <GroupBox Grid.Row="2" Grid.Column="5" Header="Wrist Right">
            <TextBlock Name="rotationWristRight" Text="Test" TextAlignment="Center"
VerticalAlignment="Center" HorizontalAlignment="Center"/>
        </GroupBox>
        <GroupBox Grid.Row="2" Grid.Column="6" Header="Hand Right">
            <TextBlock Name="rotationHandRight" Text="Test" TextAlignment="Center"
VerticalAlignment="Center" HorizontalAlignment="Center"/>
        </GroupBox>
        <GroupBox Grid.Row="2" Grid.Column="7" Header="Hip Right">
            <TextBlock Name="rotationHipRight" Text="Test" TextAlignment="Center"
VerticalAlignment="Center" HorizontalAlignment="Center"/>
        </GroupBox>
        <GroupBox Grid.Row="2" Grid.Column="8" Header="Knee Right">
            <TextBlock Name="rotationKneeRight" Text="Test" TextAlignment="Center"
VerticalAlignment="Center" HorizontalAlignment="Center"/>
```

```
        </GroupBox>
        <GroupBox Grid.Row="2" Grid.Column="9" Header="Ankle Right">
            <TextBlock Name="rotationAnkleRight" Text="Test" TextAlignment="Center"
VerticalAlignment="Center" HorizontalAlignment="Center"/>
        </GroupBox>
        <GroupBox Grid.Row="2" Grid.Column="10" Header="Foot Right">
            <TextBlock Name="rotationFootRight" Text="Test" TextAlignment="Center"
VerticalAlignment="Center" HorizontalAlignment="Center"/>
        </GroupBox>
        <Label Grid.Row="3" Grid.Column="0" Grid.RowSpan="2" Content="Angles"
FontSize="20" VerticalAlignment="Center" HorizontalAlignment="Center"/>
        <GroupBox Grid.Row="3" Grid.Column="1" Header="Left Leg">
            <TextBlock Name="angleLeftLeg" Text="Test" TextAlignment="Center"
VerticalAlignment="Center" HorizontalAlignment="Center"/>
        </GroupBox>
        <GroupBox Grid.Row="3" Grid.Column="2" Header="Left Arm">
            <TextBlock Name="angleLeftArm" Text="Test" TextAlignment="Center"
VerticalAlignment="Center" HorizontalAlignment="Center"/>
        </GroupBox>
        <GroupBox Grid.Row="4" Grid.Column="1" Header="Right Leg">
            <TextBlock Name="angleRightLeg" Text="Test" TextAlignment="Center"
VerticalAlignment="Center" HorizontalAlignment="Center"/>
        </GroupBox>
        <GroupBox Grid.Row="4" Grid.Column="2" Header="Right Arm">
            <TextBlock Name="angleRightArm" Text="Test" TextAlignment="Center"
VerticalAlignment="Center" HorizontalAlignment="Center"/>
        </GroupBox>
        <Button x:Name="exportListButton" Content="Export list" Grid.Column="10"
HorizontalAlignment="Left" Margin="10,10,0,0" Grid.Row="4" VerticalAlignment="Top"
Width="86" Height="80" Click="ExportListToFile"/>
        <TextBox x:Name="frameCounterText" Grid.Column="10" HorizontalAlignment="Left"
Height="23" Margin="0,67,0,0" Grid.Row="3" TextWrapping="Wrap" Text="TextBox"
VerticalAlignment="Top" Width="106"/>
    </Grid>
</Window>
```

MainWindow.xaml.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
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.Kinect;
using System.Windows.Media.Media3D;

namespace SkeletonRecorder2
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        private KinectSensor kinectDevice;
        private readonly Brush[] skeletonBrushes;

        private WriteableBitmap depthImageBitMap;
        private Int32Rect depthImageBitmapRect;
        private Int32 depthImageStride;
        private DepthImageFrame lastDepthFrame;

        private WriteableBitmap colorImageBitmap;
        private Int32Rect colorImageBitmapRect;
        private int colorImageStride;
        private byte[] colorImagePixelData;

        private Skeleton[] frameSkeletons;
        /// <summary>
        /// The skeleton object
        /// </summary>
        private Skeleton skeleton2;

        private Vector3D orientationVectorX;
        private Vector3D orientationVectorY;
        private Vector3D orientationVectorZ;

        /// <summary>
        /// These lists are used to store the positions of the X, Y and Z coordinates of
the knee joint
        /// </summary>
        private List<string> positionsLeftKneeX = new List<string>();
        private List<string> positionsLeftKneeY = new List<string>();
        private List<string> positionsLeftKneeZ = new List<string>();

        public MainWindow()
```

```
        {
            InitializeComponent();

            skeletonBrushes = new Brush[] { Brushes.Red };

            KinectSensor.KinectSensors.StatusChanged += KinectSensors_StatusChanged;
            this.KinectDevice = KinectSensor.KinectSensors.FirstOrDefault(x => x.Status
== KinectStatus.Connected);
        }

        public KinectSensor KinectDevice
        {
            get { return this.kinectDevice; }
            set
            {
                if (this.kinectDevice != value)
                {
                    //Uninitialize
                    if (this.kinectDevice != null)
                    {
                        this.kinectDevice.Stop();
                        this.kinectDevice.SkeletonFrameReady -=
kinectDevice_SkeletonFrameReady;
                        this.kinectDevice.ColorFrameReady -=
kinectDevice_ColorFrameReady;
                        this.kinectDevice.DepthFrameReady -=
kinectDevice_DepthFrameReady;
                        this.kinectDevice.SkeletonStream.Disable();
                        this.kinectDevice.DepthStream.Disable();
                        this.kinectDevice.ColorStream.Disable();
                        this.frameSkeletons = null;
                    }

                    this.kinectDevice = value;

                    //Initialize
                    if (this.kinectDevice != null)
                    {
                        if (this.kinectDevice.Status == KinectStatus.Connected)
                        {
                            this.kinectDevice.SkeletonStream.Enable();

this.kinectDevice.ColorStream.Enable(ColorImageFormat.RgbResolution640x480Fps30);

this.kinectDevice.DepthStream.Enable(DepthImageFormat.Resolution640x480Fps30);
                            this.frameSkeletons = new
Skeleton[this.kinectDevice.SkeletonStream.FrameSkeletonArrayLength];
                            this.kinectDevice.SkeletonFrameReady +=
kinectDevice_SkeletonFrameReady;
                            this.kinectDevice.ColorFrameReady +=
kinectDevice_ColorFrameReady;
                            this.kinectDevice.DepthFrameReady +=
kinectDevice_DepthFrameReady;
                            this.kinectDevice.Start();

                            DepthImageStream depthStream = kinectDevice.DepthStream;
                            depthStream.Enable();
```

```
                            depthImageBitMap = new
WriteableBitmap(depthStream.FrameWidth, depthStream.FrameHeight, 96, 96,
PixelFormats.Gray16, null);
                            depthImageBitmapRect = new Int32Rect(0, 0,
depthStream.FrameWidth, depthStream.FrameHeight);
                            depthImageStride = depthStream.FrameWidth *
depthStream.FrameBytesPerPixel;

                            ColorImageStream colorStream = kinectDevice.ColorStream;
                            colorStream.Enable();
                            colorImageBitmap = new
WriteableBitmap(colorStream.FrameWidth, colorStream.FrameHeight,

96, 96, PixelFormats.Bgr32, null);
                            this.colorImageBitmapRect = new Int32Rect(0, 0,
colorStream.FrameWidth, colorStream.FrameHeight);
                            this.colorImageStride = colorStream.FrameWidth *
colorStream.FrameBytesPerPixel;
                            ColorImage.Source = this.colorImageBitmap;

                            DepthImage.Source = depthImageBitMap;
                        }
                    }
                }
            }
        }

        void kinectDevice_DepthFrameReady(object sender, DepthImageFrameReadyEventArgs e)
        {
            using (DepthImageFrame depthFrame = e.OpenDepthImageFrame())
            {
                if (depthFrame != null)
                {
                    short[] depthPixelDate = new short[depthFrame.PixelDataLength];
                    depthFrame.CopyPixelDataTo(depthPixelDate);
                    depthImageBitMap.WritePixels(depthImageBitmapRect, depthPixelDate,
depthImageStride, 0);
                }
            }
        }

        void kinectDevice_ColorFrameReady(object sender, ColorImageFrameReadyEventArgs e)
        {
            using (ColorImageFrame frame = e.OpenColorImageFrame())
            {
                if (frame != null)
                {
                    byte[] pixelData = new byte[frame.PixelDataLength];
                    frame.CopyPixelDataTo(pixelData);
                    this.colorImageBitmap.WritePixels(this.colorImageBitmapRect,
pixelData, this.colorImageStride, 0);
                }
            }
        }

        void kinectDevice_SkeletonFrameReady(object sender, SkeletonFrameReadyEventArgs
e)
        {
```

```
            using (SkeletonFrame frame = e.OpenSkeletonFrame())
            {
                frameCounterText.Text = frame.FrameNumber.ToString();

                if (frame != null)
                {
                    Polyline figure;
                    Brush userBrush;
                    Skeleton skeleton;

                    LayoutRoot.Children.Clear();
                    frame.CopySkeletonDataTo(this.frameSkeletons);

                    for (int i = 0; i < this.frameSkeletons.Length; i++)
                    {
                        skeleton = this.frameSkeletons[i];

                        if (skeleton.TrackingState == SkeletonTrackingState.Tracked)
                        {
                            userBrush = this.skeletonBrushes[i %
this.skeletonBrushes.Length];

                            //draw head and body
                            figure = CreateFigure(skeleton, userBrush, new[] {
JointType.Head, JointType.ShoulderCenter, JointType.ShoulderLeft, JointType.Spine,
                                                                    JointType.ShoulderRight,
JointType.ShoulderCenter, JointType.HipCenter
                                                                    });
                            LayoutRoot.Children.Add(figure);

                            figure = CreateFigure(skeleton, userBrush, new[] {
JointType.HipLeft, JointType.HipRight });
                            LayoutRoot.Children.Add(figure);

                            //draw left leg
                            figure = CreateFigure(skeleton, userBrush, new[] {
JointType.HipCenter, JointType.HipLeft, JointType.KneeLeft, JointType.AnkleLeft,
JointType.FootLeft });
                            LayoutRoot.Children.Add(figure);

                            //draw right leg
                            figure = CreateFigure(skeleton, userBrush, new[] {
JointType.HipCenter, JointType.HipRight, JointType.KneeRight, JointType.AnkleRight,
JointType.FootRight });
                            LayoutRoot.Children.Add(figure);

                            //draw left arm
                            figure = CreateFigure(skeleton, userBrush, new[] {
JointType.ShoulderLeft, JointType.ElbowLeft, JointType.WristLeft, JointType.HandLeft });
                            LayoutRoot.Children.Add(figure);

                            //draw right arm
                            figure = CreateFigure(skeleton, userBrush, new[] {
JointType.ShoulderRight, JointType.ElbowRight, JointType.WristRight, JointType.HandRight
});
                            LayoutRoot.Children.Add(figure);
                        }
                    }
```

```
                }


                skeleton2 = (from trackSkeleton in this.frameSkeletons where
trackSkeleton.TrackingState == SkeletonTrackingState.Tracked select
trackSkeleton).FirstOrDefault();

                if (skeleton2 != null)
                {
                    positionsLeftKneeX.Add(frame.FrameNumber.ToString() + " - " +
skeleton2.Joints[JointType.KneeLeft].Position.X.ToString());
                    positionsLeftKneeY.Add(frame.FrameNumber.ToString() + " - " +
skeleton2.Joints[JointType.KneeLeft].Position.Y.ToString());
                    positionsLeftKneeZ.Add(frame.FrameNumber.ToString() + " - " +
skeleton2.Joints[JointType.KneeLeft].Position.Z.ToString());
                }
                /*
                if (skeleton2 != null)
                {
                    angleLeftLeg.Text =
CalculateJointAngleVector(skeleton2.Joints[JointType.HipLeft],
skeleton2.Joints[JointType.KneeLeft], skeleton2.Joints[JointType.AnkleLeft]);
                    angleLeftArm.Text =
CalculateJointAngleVector(skeleton2.Joints[JointType.ShoulderLeft],
skeleton2.Joints[JointType.ElbowLeft], skeleton2.Joints[JointType.WristLeft]);
                    angleRightLeg.Text =
CalculateJointAngleVector(skeleton2.Joints[JointType.HipRight],
skeleton2.Joints[JointType.KneeRight], skeleton2.Joints[JointType.AnkleRight]);
                    angleRightArm.Text =
CalculateJointAngleVector(skeleton2.Joints[JointType.ShoulderRight],
skeleton2.Joints[JointType.ElbowRight], skeleton2.Joints[JointType.WristRight]);

                    foreach(BoneOrientation boneOrientation in
skeleton2.BoneOrientations){
                        orientationVectorX.X =
RoundMatrixElement(boneOrientation.HierarchicalRotation.Matrix.M11);
                        orientationVectorX.Y =
RoundMatrixElement(boneOrientation.HierarchicalRotation.Matrix.M12);
                        orientationVectorX.Z =
RoundMatrixElement(boneOrientation.HierarchicalRotation.Matrix.M13);

                        orientationVectorY.X =
RoundMatrixElement(boneOrientation.HierarchicalRotation.Matrix.M21);
                        orientationVectorY.Y =
RoundMatrixElement(boneOrientation.HierarchicalRotation.Matrix.M22);
                        orientationVectorY.Z =
RoundMatrixElement(boneOrientation.HierarchicalRotation.Matrix.M23);

                        orientationVectorZ.X =
RoundMatrixElement(boneOrientation.HierarchicalRotation.Matrix.M31);
                        orientationVectorZ.Y =
RoundMatrixElement(boneOrientation.HierarchicalRotation.Matrix.M32);
                        orientationVectorZ.Z =
RoundMatrixElement(boneOrientation.HierarchicalRotation.Matrix.M33);

                        switch (boneOrientation.StartJoint)
                        {
                            case JointType.Head:
```

```
                                rotationHead.Text = orientationVectorX.ToString() + "\n"
+ orientationVectorY.ToString() + "\n" + orientationVectorZ.ToString();
                                break;
                            case JointType.ShoulderCenter:
                                rotationShoulderCenter.Text =
orientationVectorX.ToString() + "\n" + orientationVectorY.ToString() + "\n" +
orientationVectorZ.ToString();
                                break;
                            case JointType.Spine:
                                rotationSpine.Text = orientationVectorX.ToString() + "\n"
+ orientationVectorY.ToString() + "\n" + orientationVectorZ.ToString();
                                break;
                            case JointType.HipCenter:
                                rotationHipCenter.Text = orientationVectorX.ToString() +
"\n" + orientationVectorY.ToString() + "\n" + orientationVectorZ.ToString();
                                break;
                            case JointType.ShoulderLeft:
                                rotationShoulderLeft.Text = orientationVectorX.ToString()
+ "\n" + orientationVectorY.ToString() + "\n" + orientationVectorZ.ToString();
                                break;
                            case JointType.ElbowLeft:
                                rotationElbowLeft.Text = orientationVectorX.ToString() +
"\n" + orientationVectorY.ToString() + "\n" + orientationVectorZ.ToString();
                                break;
                            case JointType.WristLeft:
                                rotationWristLeft.Text = orientationVectorX.ToString() +
"\n" + orientationVectorY.ToString() + "\n" + orientationVectorZ.ToString();
                                break;
                            case JointType.HandLeft:
                                rotationHandLeft.Text = orientationVectorX.ToString() +
"\n" + orientationVectorY.ToString() + "\n" + orientationVectorZ.ToString();
                                break;
                            case JointType.HipLeft:
                                rotationHipLeft.Text = orientationVectorX.ToString() +
"\n" + orientationVectorY.ToString() + "\n" + orientationVectorZ.ToString();
                                break;
                            case JointType.KneeLeft:
                                rotationKneeLeft.Text = orientationVectorX.ToString() +
"\n" + orientationVectorY.ToString() + "\n" + orientationVectorZ.ToString();
                                break;
                            case JointType.AnkleLeft:
                                rotationAnkleLeft.Text = orientationVectorX.ToString() +
"\n" + orientationVectorY.ToString() + "\n" + orientationVectorZ.ToString();
                                break;
                            case JointType.FootLeft:
                                rotationFootLeft.Text = orientationVectorX.ToString() +
"\n" + orientationVectorY.ToString() + "\n" + orientationVectorZ.ToString();
                                break;
                            case JointType.ShoulderRight:
                                rotationShoulderRight.Text =
orientationVectorX.ToString() + "\n" + orientationVectorY.ToString() + "\n" +
orientationVectorZ.ToString();
                                break;
                            case JointType.ElbowRight:
                                rotationElbowRight.Text = orientationVectorX.ToString() +
"\n" + orientationVectorY.ToString() + "\n" + orientationVectorZ.ToString();
                                break;
                            case JointType.WristRight:
```

```
                                rotationWristRight.Text = orientationVectorX.ToString() +
"\n" + orientationVectorY.ToString() + "\n" + orientationVectorZ.ToString();
                                break;
                            case JointType.HandRight:
                                rotationHandRight.Text = orientationVectorX.ToString() +
"\n" + orientationVectorY.ToString() + "\n" + orientationVectorZ.ToString();
                                break;
                            case JointType.HipRight:
                                rotationHipRight.Text = orientationVectorX.ToString() +
"\n" + orientationVectorY.ToString() + "\n" + orientationVectorZ.ToString();
                                break;
                            case JointType.KneeRight:
                                rotationKneeRight.Text = orientationVectorX.ToString() +
"\n" + orientationVectorY.ToString() + "\n" + orientationVectorZ.ToString();
                                break;
                            case JointType.AnkleRight:
                                rotationAnkleRight.Text = orientationVectorX.ToString() +
"\n" + orientationVectorY.ToString() + "\n" + orientationVectorZ.ToString();
                                break;
                            case JointType.FootRight:
                                rotationFootRight.Text = orientationVectorX.ToString() +
"\n" + orientationVectorY.ToString() + "\n" + orientationVectorZ.ToString();
                                break;
                            default:
                                break;
                        }
                    }


                }*/
            }
        }

        private Polyline CreateFigure(Skeleton skeleton, Brush brush, JointType[] joints)
        {
            Polyline figure = new Polyline();

            figure.StrokeThickness = 8;
            figure.Stroke = brush;

            for (int i = 0; i < joints.Length; i++)
            {
                figure.Points.Add(GetJointPoint(skeleton.Joints[joints[i]]));
            }

            return figure;
        }

        private Point GetJointPoint(Joint joint)
        {
            CoordinateMapper cm = new CoordinateMapper(kinectDevice);

            DepthImagePoint point = cm.MapSkeletonPointToDepthPoint(joint.Position,
this.KinectDevice.DepthStream.Format);
            //ColorImagePoint point = cm.MapSkeletonPointToColorPoint(joint.Position,
this.KinectDevice.ColorStream.Format);
            point.X *= (int)this.LayoutRoot.ActualWidth /
KinectDevice.DepthStream.FrameWidth;
```

```
            point.Y *= (int)this.LayoutRoot.ActualHeight /
KinectDevice.DepthStream.FrameHeight;

            return new Point(point.X, point.Y);
        }

        private void KinectSensors_StatusChanged(object sender, StatusChangedEventArgs e)
        {
            switch (e.Status)
            {
                case KinectStatus.Initializing:
                case KinectStatus.Connected:
                case KinectStatus.NotPowered:
                case KinectStatus.NotReady:
                case KinectStatus.DeviceNotGenuine:
                    this.KinectDevice = e.Sensor;
                    break;
                case KinectStatus.Disconnected:
                    //TODO: Give the user feedback to plug-in a Kinect device.
                    this.KinectDevice = null;
                    break;
                default:
                    //TODO: Show an error state
                    break;
            }
        }

        private string CalculateJointAngleVector(Joint joint1, Joint joint2, Joint
joint3)
        {
            Vector3D vectorJoint1 = new Vector3D(joint1.Position.X, joint1.Position.Y,
joint1.Position.Z);
            Vector3D vectorJoint2 = new Vector3D(joint2.Position.X, joint2.Position.Y,
joint2.Position.Z);
            Vector3D vectorJoint3 = new Vector3D(joint3.Position.X, joint3.Position.Y,
joint3.Position.Z);

            Vector3D bone1 = vectorJoint3 - vectorJoint2;
            Vector3D bone2 = vectorJoint1 - vectorJoint2;



            double angle = Math.Round(Vector3D.AngleBetween(bone1, bone2),2);

            return angle.ToString();
        }

        private double RoundMatrixElement(float element)
        {
            return Math.Round(element,2);
        }

        /// <summary>
        /// Export each of the lists to a txt file
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void ExportListToFile(object sender, RoutedEventArgs e)
```

```
        {
            using (System.IO.StreamWriter file = new
System.IO.StreamWriter(@"D:\Output_v1_X.txt"))
            {
                foreach(string coordinateX in positionsLeftKneeX)
                {
                    file.WriteLine(coordinateX);
                }
            }

            using (System.IO.StreamWriter file = new
System.IO.StreamWriter(@"D:\Output_v1_Y.txt"))
            {
                foreach (string coordinateY in positionsLeftKneeY)
                {
                    file.WriteLine(coordinateY);
                }
            }

            using (System.IO.StreamWriter file = new
System.IO.StreamWriter(@"D:\Output_v1_Z.txt"))
            {
                foreach (string coordinateZ in positionsLeftKneeZ)
                {
                    file.WriteLine(coordinateZ);
                }
            }
        }
    }
}
```

Appendix B - BodyBasics-WPF

This application was used to output the X, Y and Z positions of the left knee during two air squats and compare these with the Kinect v1.

MainWindow.xaml

```
<Window x:Class="MyOwnVGBTest2.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:MyOwnVGBTest2"
        Title="Discrete Gesture Basics"
        Height="650" Width="750"
        Loaded="MainWindow_Loaded"
        Closing="MainWindow_Closing">
    <Window.Resources>
        <SolidColorBrush x:Key="MediumGreyBrush" Color="#ff6e6e6e" />
        <SolidColorBrush x:Key="KinectPurpleBrush" Color="#ff52318f" />
        <SolidColorBrush x:Key="KinectBlueBrush" Color="#ff00BCF2" />

        <DataTemplate DataType="{x:Type local:GestureResultView}">
            <Grid Width="Auto" Margin="5" Background="{Binding BodyColor}">
                <Grid.RowDefinitions>
                    <RowDefinition Height="Auto"/>
                    <RowDefinition />
                    <RowDefinition Height="Auto" />
                    <RowDefinition Height="Auto" />
                    <RowDefinition Height="Auto" />
                </Grid.RowDefinitions>
                <TextBlock HorizontalAlignment="Stretch" Text="{Binding BodyIndex,
StringFormat='Body Index: {0}'}" FontSize="14" FontFamily="Segoe UI"
FontWeight="SemiBold" Margin="5"/>
                <Image Source="{Binding ImageSource}" Stretch="Uniform" Grid.Row="1"
Margin="5"/>
                <TextBlock Text="{Binding Detected, StringFormat='Squatting: {0}'}"
FontSize="14" FontFamily="Segoe UI" FontWeight="SemiBold" Grid.Row="2" Margin="5 5 0 0"/>
                <TextBlock Text="{Binding Confidence, StringFormat='Confidence: {0}'}"
FontSize="14" FontFamily="Segoe UI" FontWeight="SemiBold" Grid.Row="3" Margin="5 0 0 0"/>
                <TextBlock Text="{Binding Progress, StringFormat='Progress: {0}'}"
FontSize="14" FontFamily="Segoe UI" FontWeight="SemiBold" Grid.Row="4" Margin="5 0 0 0"/>
            </Grid>
        </DataTemplate>

    </Window.Resources>

    <Grid Margin="10 0 10 0">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <Image Grid.Row="0" Source="Images\Logo.png" HorizontalAlignment="Left"
Stretch="Fill" Height="32" Width="81" Margin="0 10 0 5" />
        <TextBlock x:Name="continuousOutputBox" Grid.Row="0" Margin="0 0 -1 0"
HorizontalAlignment="Right" VerticalAlignment="Bottom" Foreground="{StaticResource
MediumGreyBrush}" FontFamily="Segoe UI" FontSize="18">Discrete and Continuous Gesture
Basics</TextBlock>
```

```
        <Image Grid.Row="0" Source="Images\Status.png" Stretch="None"
HorizontalAlignment="Center" Margin="0 0 0 5" />

        <Grid x:Name="contentGrid" Grid.Row="1" >
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="1*"/>
                <ColumnDefinition Width="1*"/>
                <ColumnDefinition Width="2*"/>
            </Grid.ColumnDefinitions>
            <Grid.RowDefinitions>
                <RowDefinition />
                <RowDefinition />
                <RowDefinition />
            </Grid.RowDefinitions>

            <StackPanel Orientation="Vertical" Grid.Column="2" Grid.RowSpan="3"
Margin="20 10 0 10" HorizontalAlignment="Center" VerticalAlignment="Center">
                <Viewbox x:Name="kinectBodyViewbox">
                    <Image Source="{Binding ImageSource}" Stretch="UniformToFill" />
                </Viewbox>
                <TextBlock Text="This program can track up to 6 people simultaneously.
Stand in front of the sensor to get tracked." TextWrapping="Wrap" Margin="5 10 5 5"
Foreground="{StaticResource MediumGreyBrush}" FontFamily="Segoe UI" FontSize="14"/>
            </StackPanel>
        </Grid>

        <StatusBar Grid.Row="2" HorizontalAlignment="Stretch" Name="statusBar"
VerticalAlignment="Bottom" Background="White" Foreground="{StaticResource
MediumGreyBrush}">
            <StatusBarItem Content="{Binding StatusText}" />
        </StatusBar>
    </Grid>
</Window>
```

MainWindow.xaml.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
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 System.Collections.ObjectModel;
using System.ComponentModel;
using Microsoft.Kinect;
using Microsoft.Kinect.VisualGestureBuilder;

namespace MyOwnVGBTest2
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window, INotifyPropertyChanged
    {
        /// <summary> Active Kinect sensor </summary>
        private KinectSensor kinectSensor = null;

        /// <summary> Array for the bodies (Kinect will track up to 6 people
simultaneously) </summary>
        private Body[] bodies = null;

        /// <summary> Reader for body frames </summary>
        private BodyFrameReader bodyFrameReader = null;

        /// <summary> Current status text to display </summary>
        private string statusText = null;

        /// <summary> KinectBodyView object which handles drawing the Kinect bodies to a
View box in the UI </summary>
        private KinectBodyView kinectBodyView = null;

        /// <summary> List of gesture detectors, there will be one detector created for
each potential body (max of 6) </summary>
        private List<GestureDetector> gestureDetectorList = null;

        /// <summary>
        /// Initializes a new instance of the MainWindow class
        /// </summary>
        public MainWindow()
        {
            // only one sensor is currently supported
            this.kinectSensor = KinectSensor.GetDefault();

            // set IsAvailableChanged event notifier
```

```
            this.kinectSensor.IsAvailableChanged += this.Sensor_IsAvailableChanged;

            // open the sensor
            this.kinectSensor.Open();

            // set the status text
            this.StatusText = this.kinectSensor.IsAvailable ?
Properties.Resources.RunningStatusText
                                                            :
Properties.Resources.NoSensorStatusText;

            // open the reader for the body frames
            this.bodyFrameReader = this.kinectSensor.BodyFrameSource.OpenReader();

            // initialize the BodyViewer object for displaying tracked bodies in the UI
            this.kinectBodyView = new KinectBodyView(this.kinectSensor);

            // initialize the gesture detection objects for our gestures
            this.gestureDetectorList = new List<GestureDetector>();

            // initialize the MainWindow
            this.InitializeComponent();

            // set our data context objects for display in UI
            this.DataContext = this;
            this.kinectBodyViewbox.DataContext = this.kinectBodyView;

            // create a gesture detector for each body (6 bodies => 6 detectors) and
create content controls to display results in the UI
            int col0Row = 0;
            int col1Row = 0;
            int maxBodies = this.kinectSensor.BodyFrameSource.BodyCount;
            for (int i = 0; i < maxBodies; ++i)
            {
                GestureResultView result = new GestureResultView(i, false, false, 0.0f,
0.0f);
                GestureDetector detector = new GestureDetector(this.kinectSensor,
result);
                this.gestureDetectorList.Add(detector);

                // split gesture results across the first two columns of the content grid
                ContentControl contentControl = new ContentControl();
                contentControl.Content = this.gestureDetectorList[i].GestureResultView;

                if (i % 2 == 0)
                {
                    // Gesture results for bodies: 0, 2, 4
                    Grid.SetColumn(contentControl, 0);
                    Grid.SetRow(contentControl, col0Row);
                    ++col0Row;
                }
                else
                {
                    // Gesture results for bodies: 1, 3, 5
                    Grid.SetColumn(contentControl, 1);
                    Grid.SetRow(contentControl, col1Row);
                    ++col1Row;
                }
```

```
                this.contentGrid.Children.Add(contentControl);
            }
        }

        /// <summary>
        /// INotifyPropertyChangedPropertyChanged event to allow window controls to bind
to changeable data
        /// </summary>
        public event PropertyChangedEventHandler PropertyChanged;

        /// <summary>
        /// Gets or sets the current status text to display
        /// </summary>
        public string StatusText
        {
            get
            {
                return this.statusText;
            }

            set
            {
                if (this.statusText != value)
                {
                    this.statusText = value;

                    // notify any bound elements that the text has changed
                    if (this.PropertyChanged != null)
                    {
                        this.PropertyChanged(this, new
PropertyChangedEventArgs("StatusText"));
                    }
                }
            }
        }

        /// <summary>
        /// Execute start up tasks
        /// </summary>
        /// <param name="sender">object sending the event</param>
        /// <param name="e">event arguments</param>
        private void MainWindow_Loaded(object sender, RoutedEventArgs e)
        {
            if (this.bodyFrameReader != null)
            {
                this.bodyFrameReader.FrameArrived += this.Reader_BodyFrameArrived;
            }
        }

        /// <summary>
        /// Execute shutdown tasks
        /// </summary>
        /// <param name="sender">object sending the event</param>
        /// <param name="e">event arguments</param>
        private void MainWindow_Closing(object sender, CancelEventArgs e)
        {
            if (this.bodyFrameReader != null)
```

```
            {
                // BodyFrameReader is IDisposable
                this.bodyFrameReader.FrameArrived -= this.Reader_BodyFrameArrived;
                this.bodyFrameReader.Dispose();
                this.bodyFrameReader = null;
            }

            if (this.gestureDetectorList != null)
            {
                // The GestureDetector contains disposable members
(VisualGestureBuilderFrameSource and VisualGestureBuilderFrameReader)
                foreach (GestureDetector detector in this.gestureDetectorList)
                {
                    detector.Dispose();
                }

                this.gestureDetectorList.Clear();
                this.gestureDetectorList = null;
            }

            if (this.kinectSensor != null)
            {
                this.kinectSensor.IsAvailableChanged -= this.Sensor_IsAvailableChanged;
                this.kinectSensor.Close();
                this.kinectSensor = null;
            }
        }

        /// <summary>
        /// Handles the event when the sensor becomes unavailable (e.g. paused, closed,
unplugged).
        /// </summary>
        /// <param name="sender">object sending the event</param>
        /// <param name="e">event arguments</param>
        private void Sensor_IsAvailableChanged(object sender, IsAvailableChangedEventArgs
e)
        {
            // on failure, set the status text
            this.StatusText = this.kinectSensor.IsAvailable ?
Properties.Resources.RunningStatusText
                                                            :
Properties.Resources.SensorNotAvailableStatusText;
        }

        /// <summary>
        /// Handles the body frame data arriving from the sensor and updates the
associated gesture detector object for each body
        /// </summary>
        /// <param name="sender">object sending the event</param>
        /// <param name="e">event arguments</param>
        private void Reader_BodyFrameArrived(object sender, BodyFrameArrivedEventArgs e)
        {
            bool dataReceived = false;

            using (BodyFrame bodyFrame = e.FrameReference.AcquireFrame())
            {
                if (bodyFrame != null)
                {
```

```
                    if (this.bodies == null)
                    {
                        // creates an array of 6 bodies, which is the max number of
bodies that Kinect can track simultaneously
                        this.bodies = new Body[bodyFrame.BodyCount];
                    }

                    // The first time GetAndRefreshBodyData is called, Kinect will
allocate each Body in the array.
                    // As long as those body objects are not disposed and not set to null
in the array,
                    // those body objects will be re-used.
                    bodyFrame.GetAndRefreshBodyData(this.bodies);
                    dataReceived = true;
                }
            }

            if (dataReceived)
            {
                // visualize the new body data
                this.kinectBodyView.UpdateBodyFrame(this.bodies);

                // we may have lost/acquired bodies, so update the corresponding gesture
detectors
                if (this.bodies != null)
                {
                    // loop through all bodies to see if any of the gesture detectors
need to be updated
                    int maxBodies = this.kinectSensor.BodyFrameSource.BodyCount;
                    for (int i = 0; i < maxBodies; ++i)
                    {
                        Body body = this.bodies[i];
                        ulong trackingId = body.TrackingId;

                        // if the current body TrackingId changed, update the
corresponding gesture detector with the new value
                        if (trackingId != this.gestureDetectorList[i].TrackingId)
                        {
                            this.gestureDetectorList[i].TrackingId = trackingId;

                            // if the current body is tracked, unpause its detector to
get VisualGestureBuilderFrameArrived events
                            // if the current body is not tracked, pause its detector so
we don't waste resources trying to get invalid gesture results
                            this.gestureDetectorList[i].IsPaused = trackingId == 0;
                        }
                    }
                }
            }
        }
    }
}
```

Appendix C - MyOwnVGBTest2

This application was used to test the database files created with Visual Gesture Builder.

MainWindow.xaml

```
<Window x:Class="MyOwnVGBTest2.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:MyOwnVGBTest2"
        Title="Discrete Gesture Basics"
        Height="650" Width="750"
        Loaded="MainWindow_Loaded"
        Closing="MainWindow_Closing">
    <Window.Resources>
        <SolidColorBrush x:Key="MediumGreyBrush" Color="#ff6e6e6e" />
        <SolidColorBrush x:Key="KinectPurpleBrush" Color="#ff52318f" />
        <SolidColorBrush x:Key="KinectBlueBrush" Color="#ff00BCF2" />

        <DataTemplate DataType="{x:Type local:GestureResultView}">
            <Grid Width="Auto" Margin="5" Background="{Binding BodyColor}">
                <Grid.RowDefinitions>
                    <RowDefinition Height="Auto"/>
                    <RowDefinition />
                    <RowDefinition Height="Auto" />
                    <RowDefinition Height="Auto" />
                    <RowDefinition Height="Auto" />
                </Grid.RowDefinitions>
                <TextBlock HorizontalAlignment="Stretch" Text="{Binding BodyIndex,
StringFormat='Body Index: {0}'}" FontSize="14" FontFamily="Segoe UI"
FontWeight="SemiBold" Margin="5"/>
                <Image Source="{Binding ImageSource}" Stretch="Uniform" Grid.Row="1"
Margin="5"/>
                <TextBlock Text="{Binding Detected, StringFormat='Squatting: {0}'}"
FontSize="14" FontFamily="Segoe UI" FontWeight="SemiBold" Grid.Row="2" Margin="5 5 0 0"/>
                <TextBlock Text="{Binding Confidence, StringFormat='Confidence: {0}'}"
FontSize="14" FontFamily="Segoe UI" FontWeight="SemiBold" Grid.Row="3" Margin="5 0 0 0"/>
                <TextBlock Text="{Binding Progress, StringFormat='Progress: {0}'}"
FontSize="14" FontFamily="Segoe UI" FontWeight="SemiBold" Grid.Row="4" Margin="5 0 0 0"/>
            </Grid>
        </DataTemplate>

    </Window.Resources>

    <Grid Margin="10 0 10 0">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <Image Grid.Row="0" Source="Images\Logo.png" HorizontalAlignment="Left"
Stretch="Fill" Height="32" Width="81" Margin="0 10 0 5" />
        <TextBlock x:Name="continuousOutputBox" Grid.Row="0" Margin="0 0 -1 0"
HorizontalAlignment="Right" VerticalAlignment="Bottom" Foreground="{StaticResource
MediumGreyBrush}" FontFamily="Segoe UI" FontSize="18">Discrete and Continuous Gesture
Basics</TextBlock>
```

```
        <Image Grid.Row="0" Source="Images\Status.png" Stretch="None"
HorizontalAlignment="Center" Margin="0 0 0 5" />

        <Grid x:Name="contentGrid" Grid.Row="1" >
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="1*"/>
                <ColumnDefinition Width="1*"/>
                <ColumnDefinition Width="2*"/>
            </Grid.ColumnDefinitions>
            <Grid.RowDefinitions>
                <RowDefinition />
                <RowDefinition />
                <RowDefinition />
            </Grid.RowDefinitions>

            <StackPanel Orientation="Vertical" Grid.Column="2" Grid.RowSpan="3"
Margin="20 10 0 10" HorizontalAlignment="Center" VerticalAlignment="Center">
                <Viewbox x:Name="kinectBodyViewbox">
                    <Image Source="{Binding ImageSource}" Stretch="UniformToFill" />
                </Viewbox>
                <TextBlock Text="This program can track up to 6 people simultaneously.
Stand in front of the sensor to get tracked." TextWrapping="Wrap" Margin="5 10 5 5"
Foreground="{StaticResource MediumGreyBrush}" FontFamily="Segoe UI" FontSize="14"/>
            </StackPanel>
        </Grid>

        <StatusBar Grid.Row="2" HorizontalAlignment="Stretch" Name="statusBar"
VerticalAlignment="Bottom" Background="White" Foreground="{StaticResource
MediumGreyBrush}">
            <StatusBarItem Content="{Binding StatusText}" />
        </StatusBar>
    </Grid>
</Window>
```

MainWindow.xaml.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
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 System.Collections.ObjectModel;
using System.ComponentModel;
using Microsoft.Kinect;
using Microsoft.Kinect.VisualGestureBuilder;

namespace MyOwnVGBTest2
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window, INotifyPropertyChanged
    {
        /// <summary> Active Kinect sensor </summary>
        private KinectSensor kinectSensor = null;

        /// <summary> Array for the bodies (Kinect will track up to 6 people
simultaneously) </summary>
        private Body[] bodies = null;

        /// <summary> Reader for body frames </summary>
        private BodyFrameReader bodyFrameReader = null;

        /// <summary> Current status text to display </summary>
        private string statusText = null;

        /// <summary> KinectBodyView object which handles drawing the Kinect bodies to a
View box in the UI </summary>
        private KinectBodyView kinectBodyView = null;

        /// <summary> List of gesture detectors, there will be one detector created for
each potential body (max of 6) </summary>
        private List<GestureDetector> gestureDetectorList = null;

        /// <summary>
        /// Initializes a new instance of the MainWindow class
        /// </summary>
        public MainWindow()
        {
            // only one sensor is currently supported
            this.kinectSensor = KinectSensor.GetDefault();

            // set IsAvailableChanged event notifier
```

```
            this.kinectSensor.IsAvailableChanged += this.Sensor_IsAvailableChanged;

            // open the sensor
            this.kinectSensor.Open();

            // set the status text
            this.StatusText = this.kinectSensor.IsAvailable ?
Properties.Resources.RunningStatusText
                                                            :
Properties.Resources.NoSensorStatusText;

            // open the reader for the body frames
            this.bodyFrameReader = this.kinectSensor.BodyFrameSource.OpenReader();

            // initialize the BodyViewer object for displaying tracked bodies in the UI
            this.kinectBodyView = new KinectBodyView(this.kinectSensor);

            // initialize the gesture detection objects for our gestures
            this.gestureDetectorList = new List<GestureDetector>();

            // initialize the MainWindow
            this.InitializeComponent();

            // set our data context objects for display in UI
            this.DataContext = this;
            this.kinectBodyViewbox.DataContext = this.kinectBodyView;

            // create a gesture detector for each body (6 bodies => 6 detectors) and
create content controls to display results in the UI
            int col0Row = 0;
            int col1Row = 0;
            int maxBodies = this.kinectSensor.BodyFrameSource.BodyCount;
            for (int i = 0; i < maxBodies; ++i)
            {
                GestureResultView result = new GestureResultView(i, false, false, 0.0f,
0.0f);
                GestureDetector detector = new GestureDetector(this.kinectSensor,
result);
                this.gestureDetectorList.Add(detector);

                // split gesture results across the first two columns of the content grid
                ContentControl contentControl = new ContentControl();
                contentControl.Content = this.gestureDetectorList[i].GestureResultView;

                if (i % 2 == 0)
                {
                    // Gesture results for bodies: 0, 2, 4
                    Grid.SetColumn(contentControl, 0);
                    Grid.SetRow(contentControl, col0Row);
                    ++col0Row;
                }
                else
                {
                    // Gesture results for bodies: 1, 3, 5
                    Grid.SetColumn(contentControl, 1);
                    Grid.SetRow(contentControl, col1Row);
                    ++col1Row;
                }
```

```
                this.contentGrid.Children.Add(contentControl);
            }
        }

        /// <summary>
        /// INotifyPropertyChangedPropertyChanged event to allow window controls to bind
to changeable data
        /// </summary>
        public event PropertyChangedEventHandler PropertyChanged;

        /// <summary>
        /// Gets or sets the current status text to display
        /// </summary>
        public string StatusText
        {
            get
            {
                return this.statusText;
            }

            set
            {
                if (this.statusText != value)
                {
                    this.statusText = value;

                    // notify any bound elements that the text has changed
                    if (this.PropertyChanged != null)
                    {
                        this.PropertyChanged(this, new
PropertyChangedEventArgs("StatusText"));
                    }
                }
            }
        }

        /// <summary>
        /// Execute start up tasks
        /// </summary>
        /// <param name="sender">object sending the event</param>
        /// <param name="e">event arguments</param>
        private void MainWindow_Loaded(object sender, RoutedEventArgs e)
        {
            if (this.bodyFrameReader != null)
            {
                this.bodyFrameReader.FrameArrived += this.Reader_BodyFrameArrived;
            }
        }

        /// <summary>
        /// Execute shutdown tasks
        /// </summary>
        /// <param name="sender">object sending the event</param>
        /// <param name="e">event arguments</param>
        private void MainWindow_Closing(object sender, CancelEventArgs e)
        {
            if (this.bodyFrameReader != null)
```

```
            {
                // BodyFrameReader is IDisposable
                this.bodyFrameReader.FrameArrived -= this.Reader_BodyFrameArrived;
                this.bodyFrameReader.Dispose();
                this.bodyFrameReader = null;
            }

            if (this.gestureDetectorList != null)
            {
                // The GestureDetector contains disposable members
(VisualGestureBuilderFrameSource and VisualGestureBuilderFrameReader)
                foreach (GestureDetector detector in this.gestureDetectorList)
                {
                    detector.Dispose();
                }

                this.gestureDetectorList.Clear();
                this.gestureDetectorList = null;
            }

            if (this.kinectSensor != null)
            {
                this.kinectSensor.IsAvailableChanged -= this.Sensor_IsAvailableChanged;
                this.kinectSensor.Close();
                this.kinectSensor = null;
            }
        }

        /// <summary>
        /// Handles the event when the sensor becomes unavailable (e.g. paused, closed,
unplugged).
        /// </summary>
        /// <param name="sender">object sending the event</param>
        /// <param name="e">event arguments</param>
        private void Sensor_IsAvailableChanged(object sender, IsAvailableChangedEventArgs
e)
        {
            // on failure, set the status text
            this.StatusText = this.kinectSensor.IsAvailable ?
Properties.Resources.RunningStatusText
                                                            :
Properties.Resources.SensorNotAvailableStatusText;
        }

        /// <summary>
        /// Handles the body frame data arriving from the sensor and updates the
associated gesture detector object for each body
        /// </summary>
        /// <param name="sender">object sending the event</param>
        /// <param name="e">event arguments</param>
        private void Reader_BodyFrameArrived(object sender, BodyFrameArrivedEventArgs e)
        {
            bool dataReceived = false;

            using (BodyFrame bodyFrame = e.FrameReference.AcquireFrame())
            {
                if (bodyFrame != null)
                {
```

```
                    if (this.bodies == null)
                    {
                        // creates an array of 6 bodies, which is the max number of
bodies that Kinect can track simultaneously
                        this.bodies = new Body[bodyFrame.BodyCount];
                    }

                    // The first time GetAndRefreshBodyData is called, Kinect will
allocate each Body in the array.
                    // As long as those body objects are not disposed and not set to null
in the array,
                    // those body objects will be re-used.
                    bodyFrame.GetAndRefreshBodyData(this.bodies);
                    dataReceived = true;
                }
            }

            if (dataReceived)
            {
                // visualize the new body data
                this.kinectBodyView.UpdateBodyFrame(this.bodies);

                // we may have lost/acquired bodies, so update the corresponding gesture
detectors
                if (this.bodies != null)
                {
                    // loop through all bodies to see if any of the gesture detectors
need to be updated
                    int maxBodies = this.kinectSensor.BodyFrameSource.BodyCount;
                    for (int i = 0; i < maxBodies; ++i)
                    {
                        Body body = this.bodies[i];
                        ulong trackingId = body.TrackingId;

                        // if the current body TrackingId changed, update the
corresponding gesture detector with the new value
                        if (trackingId != this.gestureDetectorList[i].TrackingId)
                        {
                            this.gestureDetectorList[i].TrackingId = trackingId;

                            // if the current body is tracked, unpause its detector to
get VisualGestureBuilderFrameArrived events
                            // if the current body is not tracked, pause its detector so
we don't waste resources trying to get invalid gesture results
                            this.gestureDetectorList[i].IsPaused = trackingId == 0;
                        }
                    }
                }
            }
        }
    }
}
```

KinectBodyView.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Media;
using Microsoft.Kinect;

namespace MyOwnVGBTest2
{
    /// <summary>
    /// Visualizes the Kinect Body stream for display in the UI
    /// </summary>
    class KinectBodyView
    {
        /// <summary>
        /// Radius of drawn hand circles
        /// </summary>
        private const double HandSize = 30;

        /// <summary>
        /// Thickness of drawn joint lines
        /// </summary>
        private const double JointThickness = 3;

        /// <summary>
        /// Thickness of clip edge rectangles
        /// </summary>
        private const double ClipBoundsThickness = 10;

        /// <summary>
        /// Constant for clamping Z values of camera space points from being negative
        /// </summary>
        private const float InferredZPositionClamp = 0.1f;

        /// <summary>
        /// Brush used for drawing hands that are currently tracked as closed
        /// </summary>
        private readonly Brush handClosedBrush = new SolidColorBrush(Color.FromArgb(128,
255, 0, 0));

        /// <summary>
        /// Brush used for drawing hands that are currently tracked as opened
        /// </summary>
        private readonly Brush handOpenBrush = new SolidColorBrush(Color.FromArgb(128, 0,
255, 0));

        /// <summary>
        /// Brush used for drawing hands that are currently tracked as in lasso (pointer)
position
        /// </summary>
        private readonly Brush handLassoBrush = new SolidColorBrush(Color.FromArgb(128,
0, 0, 255));

        /// <summary>
```

```
        /// Brush used for drawing joints that are currently tracked
        /// </summary>
        private readonly Brush trackedJointBrush = new
SolidColorBrush(Color.FromArgb(255, 68, 192, 68));

        /// <summary>
        /// Brush used for drawing joints that are currently inferred
        /// </summary>
        private readonly Brush inferredJointBrush = Brushes.Yellow;

        /// <summary>
        /// Pen used for drawing bones that are currently inferred
        /// </summary>
        private readonly Pen inferredBonePen = new Pen(Brushes.Gray, 1);

        /// <summary>
        /// Drawing group for body rendering output
        /// </summary>
        private DrawingGroup drawingGroup;

        /// <summary>
        /// Drawing image that we will display
        /// </summary>
        private DrawingImage imageSource;

        /// <summary>
        /// Coordinate mapper to map one type of point to another
        /// </summary>
        private CoordinateMapper coordinateMapper = null;

        /// <summary>
        /// definition of bones
        /// </summary>
        private List<Tuple<JointType, JointType>> bones;

        /// <summary>
        /// Width of display (depth space)
        /// </summary>
        private int displayWidth;

        /// <summary>
        /// Height of display (depth space)
        /// </summary>
        private int displayHeight;

        /// <summary>
        /// List of colors for each body tracked
        /// </summary>
        private List<Pen> bodyColors;

        /// <summary>
        /// Initializes a new instance of the KinectBodyView class
        /// </summary>
        /// <param name="kinectSensor">Active instance of the KinectSensor</param>
        public KinectBodyView(KinectSensor kinectSensor)
        {
            if (kinectSensor == null)
            {
```

```
                throw new ArgumentNullException("kinectSensor");
            }

            // get the coordinate mapper
            this.coordinateMapper = kinectSensor.CoordinateMapper;

            // get the depth (display) extents
            FrameDescription frameDescription =
kinectSensor.DepthFrameSource.FrameDescription;

            // get size of joint space
            this.displayWidth = frameDescription.Width;
            this.displayHeight = frameDescription.Height;

            // a bone defined as a line between two joints
            this.bones = new List<Tuple<JointType, JointType>>();

            // Torso
            this.bones.Add(new Tuple<JointType, JointType>(JointType.Head,
JointType.Neck));
            this.bones.Add(new Tuple<JointType, JointType>(JointType.Neck,
JointType.SpineShoulder));
            this.bones.Add(new Tuple<JointType, JointType>(JointType.SpineShoulder,
JointType.SpineMid));
            this.bones.Add(new Tuple<JointType, JointType>(JointType.SpineMid,
JointType.SpineBase));
            this.bones.Add(new Tuple<JointType, JointType>(JointType.SpineShoulder,
JointType.ShoulderRight));
            this.bones.Add(new Tuple<JointType, JointType>(JointType.SpineShoulder,
JointType.ShoulderLeft));
            this.bones.Add(new Tuple<JointType, JointType>(JointType.SpineBase,
JointType.HipRight));
            this.bones.Add(new Tuple<JointType, JointType>(JointType.SpineBase,
JointType.HipLeft));

            // Right Arm
            this.bones.Add(new Tuple<JointType, JointType>(JointType.ShoulderRight,
JointType.ElbowRight));
            this.bones.Add(new Tuple<JointType, JointType>(JointType.ElbowRight,
JointType.WristRight));
            this.bones.Add(new Tuple<JointType, JointType>(JointType.WristRight,
JointType.HandRight));
            this.bones.Add(new Tuple<JointType, JointType>(JointType.HandRight,
JointType.HandTipRight));
            this.bones.Add(new Tuple<JointType, JointType>(JointType.WristRight,
JointType.ThumbRight));

            // Left Arm
            this.bones.Add(new Tuple<JointType, JointType>(JointType.ShoulderLeft,
JointType.ElbowLeft));
            this.bones.Add(new Tuple<JointType, JointType>(JointType.ElbowLeft,
JointType.WristLeft));
            this.bones.Add(new Tuple<JointType, JointType>(JointType.WristLeft,
JointType.HandLeft));
            this.bones.Add(new Tuple<JointType, JointType>(JointType.HandLeft,
JointType.HandTipLeft));
            this.bones.Add(new Tuple<JointType, JointType>(JointType.WristLeft,
JointType.ThumbLeft));
```

```
            // Right Leg
            this.bones.Add(new Tuple<JointType, JointType>(JointType.HipRight,
JointType.KneeRight));
            this.bones.Add(new Tuple<JointType, JointType>(JointType.KneeRight,
JointType.AnkleRight));
            this.bones.Add(new Tuple<JointType, JointType>(JointType.AnkleRight,
JointType.FootRight));

            // Left Leg
            this.bones.Add(new Tuple<JointType, JointType>(JointType.HipLeft,
JointType.KneeLeft));
            this.bones.Add(new Tuple<JointType, JointType>(JointType.KneeLeft,
JointType.AnkleLeft));
            this.bones.Add(new Tuple<JointType, JointType>(JointType.AnkleLeft,
JointType.FootLeft));

            // populate body colors, one for each BodyIndex
            this.bodyColors = new List<Pen>();

            this.bodyColors.Add(new Pen(Brushes.Red, 6));
            this.bodyColors.Add(new Pen(Brushes.Orange, 6));
            this.bodyColors.Add(new Pen(Brushes.Green, 6));
            this.bodyColors.Add(new Pen(Brushes.Blue, 6));
            this.bodyColors.Add(new Pen(Brushes.Indigo, 6));
            this.bodyColors.Add(new Pen(Brushes.Violet, 6));

            // Create the drawing group we'll use for drawing
            this.drawingGroup = new DrawingGroup();

            // Create an image source that we can use in our image control
            this.imageSource = new DrawingImage(this.drawingGroup);
        }

        /// <summary>
        /// Gets the bitmap to display
        /// </summary>
        public ImageSource ImageSource
        {
            get
            {
                return this.imageSource;
            }
        }

        /// <summary>
        /// Updates the body array with new information from the sensor
        /// Should be called whenever a new BodyFrameArrivedEvent occurs
        /// </summary>
        /// <param name="bodies">Array of bodies to update</param>
        public void UpdateBodyFrame(Body[] bodies)
        {
            if (bodies != null)
            {
                using (DrawingContext dc = this.drawingGroup.Open())
                {
                    // Draw a transparent background to set the render size
```

```
                    dc.DrawRectangle(Brushes.Black, null, new Rect(0.0, 0.0,
this.displayWidth, this.displayHeight));

                    int penIndex = 0;
                    foreach (Body body in bodies)
                    {
                        Pen drawPen = this.bodyColors[penIndex++];

                        if (body.IsTracked)
                        {
                            this.DrawClippedEdges(body, dc);

                            IReadOnlyDictionary<JointType, Joint> joints = body.Joints;

                            // convert the joint points to depth (display) space
                            Dictionary<JointType, Point> jointPoints = new
Dictionary<JointType, Point>();

                            foreach (JointType jointType in joints.Keys)
                            {
                                // sometimes the depth(Z) of an inferred joint may show
as negative

                                // clamp down to 0.1f to prevent coordinatemapper from
returning (-Infinity, -Infinity)
                                CameraSpacePoint position = joints[jointType].Position;
                                if (position.Z < 0)
                                {
                                    position.Z = InferredZPositionClamp;
                                }

                                DepthSpacePoint depthSpacePoint =
this.coordinateMapper.MapCameraPointToDepthSpace(position);
                                jointPoints[jointType] = new Point(depthSpacePoint.X,
depthSpacePoint.Y);
                            }

                            this.DrawBody(joints, jointPoints, dc, drawPen);

                            this.DrawHand(body.HandLeftState,
jointPoints[JointType.HandLeft], dc);
                            this.DrawHand(body.HandRightState,
jointPoints[JointType.HandRight], dc);
                        }
                    }

                    // prevent drawing outside of our render area
                    this.drawingGroup.ClipGeometry = new RectangleGeometry(new Rect(0.0,
0.0, this.displayWidth, this.displayHeight));
                }
            }
        }

        /// <summary>
        /// Draws a body
        /// </summary>
        /// <param name="joints">joints to draw</param>
        /// <param name="jointPoints">translated positions of joints to draw</param>
        /// <param name="drawingContext">drawing context to draw to</param>
```

```
        /// <param name="drawingPen">specifies color to draw a specific body</param>
        private void DrawBody(IReadOnlyDictionary<JointType, Joint> joints,
IDictionary<JointType, Point> jointPoints, DrawingContext drawingContext, Pen drawingPen)
        {
            // Draw the bones
            foreach (var bone in this.bones)
            {
                this.DrawBone(joints, jointPoints, bone.Item1, bone.Item2,
drawingContext, drawingPen);
            }

            // Draw the joints
            foreach (JointType jointType in joints.Keys)
            {
                Brush drawBrush = null;

                TrackingState trackingState = joints[jointType].TrackingState;

                if (trackingState == TrackingState.Tracked)
                {
                    drawBrush = this.trackedJointBrush;
                }
                else if (trackingState == TrackingState.Inferred)
                {
                    drawBrush = this.inferredJointBrush;
                }

                if (drawBrush != null)
                {
                    drawingContext.DrawEllipse(drawBrush, null, jointPoints[jointType],
JointThickness, JointThickness);
                }
            }
        }

        /// <summary>
        /// Draws one bone of a body (joint to joint)
        /// </summary>
        /// <param name="joints">joints to draw</param>
        /// <param name="jointPoints">translated positions of joints to draw</param>
        /// <param name="jointType0">first joint of bone to draw</param>
        /// <param name="jointType1">second joint of bone to draw</param>
        /// <param name="drawingContext">drawing context to draw to</param>
        /// /// <param name="drawingPen">specifies color to draw a specific bone</param>
        private void DrawBone(IReadOnlyDictionary<JointType, Joint> joints,
IDictionary<JointType, Point> jointPoints, JointType jointType0, JointType jointType1,
DrawingContext drawingContext, Pen drawingPen)
        {
            Joint joint0 = joints[jointType0];
            Joint joint1 = joints[jointType1];

            // If we can't find either of these joints, exit
            if (joint0.TrackingState == TrackingState.NotTracked ||
                joint1.TrackingState == TrackingState.NotTracked)
            {
                return;
            }
```

```
            // We assume all drawn bones are inferred unless BOTH joints are tracked
            Pen drawPen = this.inferredBonePen;
            if ((joint0.TrackingState == TrackingState.Tracked) && (joint1.TrackingState
== TrackingState.Tracked))
            {
                drawPen = drawingPen;
            }

            drawingContext.DrawLine(drawPen, jointPoints[jointType0],
jointPoints[jointType1]);
        }

        /// <summary>
        /// Draws a hand symbol if the hand is tracked: red circle = closed, green circle
= opened; blue circle = lasso
        /// </summary>
        /// <param name="handState">state of the hand</param>
        /// <param name="handPosition">position of the hand</param>
        /// <param name="drawingContext">drawing context to draw to</param>
        private void DrawHand(HandState handState, Point handPosition, DrawingContext
drawingContext)
        {
            switch (handState)
            {
                case HandState.Closed:
                    drawingContext.DrawEllipse(this.handClosedBrush, null, handPosition,
HandSize, HandSize);
                    break;

                case HandState.Open:
                    drawingContext.DrawEllipse(this.handOpenBrush, null, handPosition,
HandSize, HandSize);
                    break;

                case HandState.Lasso:
                    drawingContext.DrawEllipse(this.handLassoBrush, null, handPosition,
HandSize, HandSize);
                    break;
            }
        }

        /// <summary>
        /// Draws indicators to show which edges are clipping body data
        /// </summary>
        /// <param name="body">body to draw clipping information for</param>
        /// <param name="drawingContext">drawing context to draw to</param>
        private void DrawClippedEdges(Body body, DrawingContext drawingContext)
        {
            FrameEdges clippedEdges = body.ClippedEdges;

            if (clippedEdges.HasFlag(FrameEdges.Bottom))
            {
                drawingContext.DrawRectangle(
                    Brushes.Red,
                    null,
                    new Rect(0, this.displayHeight - ClipBoundsThickness,
this.displayWidth, ClipBoundsThickness));
            }
```

```
            if (clippedEdges.HasFlag(FrameEdges.Top))
            {
                drawingContext.DrawRectangle(
                    Brushes.Red,
                    null,
                    new Rect(0, 0, this.displayWidth, ClipBoundsThickness));
            }

            if (clippedEdges.HasFlag(FrameEdges.Left))
            {
                drawingContext.DrawRectangle(
                    Brushes.Red,
                    null,
                    new Rect(0, 0, ClipBoundsThickness, this.displayHeight));
            }

            if (clippedEdges.HasFlag(FrameEdges.Right))
            {
                drawingContext.DrawRectangle(
                    Brushes.Red,
                    null,
                    new Rect(this.displayWidth - ClipBoundsThickness, 0,
ClipBoundsThickness, this.displayHeight));
            }
        }
    }
}
```

GestureDectector.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Microsoft.Kinect;
using Microsoft.Kinect.VisualGestureBuilder;

namespace MyOwnVGBTest2
{
    /// <summary>
    /// Gesture Detector class which listens for VisualGestureBuilderFrame events from
the service
    /// and updates the associated GestureResultView object with the latest results for
the 'Seated' gesture
    /// </summary>
    class GestureDetector : IDisposable
    {
        /// <summary> Path to the gesture database that was trained with VGB </summary>
        private readonly string gestureDatabase = @"Database\ThesisWorkouts.gbd";

        /// <summary> Name of the discrete gesture in the database that we want to track
</summary>
        private readonly string seatedGestureName = "SquatDiscrete";

        /// <summary> Name of the continuous gesture in the database that we want to
track </summary>
        private readonly string continuousGestureName = "Squat"; //ADDED BY ME

        /// <summary> Gesture frame source which should be tied to a body tracking ID
</summary>
        private VisualGestureBuilderFrameSource vgbFrameSource = null;

        /// <summary> Gesture frame reader which will handle gesture events coming from
the sensor </summary>
        private VisualGestureBuilderFrameReader vgbFrameReader = null;

        /// <summary> The variable that holds the value of the progress output of the
continuous gesture </summary>
        public float progressResult = -10.0f; //ADDED BY ME

        /// <summary>
        /// Initializes a new instance of the GestureDetector class along with the
gesture frame source and reader
        /// </summary>
        /// <param name="kinectSensor">Active sensor to initialize the
VisualGestureBuilderFrameSource object with</param>
        /// <param name="gestureResultView">GestureResultView object to store gesture
results of a single body to</param>
        public GestureDetector(KinectSensor kinectSensor, GestureResultView
gestureResultView)
        {
            if (kinectSensor == null)
            {
                throw new ArgumentNullException("kinectSensor");
            }
```

```
            if (gestureResultView == null)
            {
                throw new ArgumentNullException("gestureResultView");
            }

            this.GestureResultView = gestureResultView;

            // create the vgb source. The associated body tracking ID will be set when a
valid body frame arrives from the sensor.
            this.vgbFrameSource = new VisualGestureBuilderFrameSource(kinectSensor, 0);
            this.vgbFrameSource.TrackingIdLost += this.Source_TrackingIdLost;

            // open the reader for the vgb frames
            this.vgbFrameReader = this.vgbFrameSource.OpenReader();
            if (this.vgbFrameReader != null)
            {
                this.vgbFrameReader.IsPaused = true;
                this.vgbFrameReader.FrameArrived += this.Reader_GestureFrameArrived;
            }

            // load the 'Seated' gesture from the gesture database
            using (VisualGestureBuilderDatabase database = new
VisualGestureBuilderDatabase(this.gestureDatabase))
            {
                // we could load all available gestures in the database with a call to
vgbFrameSource.AddGestures(database.AvailableGestures),
                // but for this program, we only want to track one discrete gesture from
the database, so we'll load it by name
                /*foreach (Gesture gesture in database.AvailableGestures)
                {
                    if (gesture.Name.Equals(this.seatedGestureName))
                    {
                        this.vgbFrameSource.AddGesture(gesture);
                    }
                }*///COMMENTED OUT BY ME

                //Adding all the gestures available in the gestureDatabase instead of
adding them seperately
                this.vgbFrameSource.AddGestures(database.AvailableGestures); //ADDED BY
ME
            }
        }

        /// <summary> Gets the GestureResultView object which stores the detector results
for display in the UI </summary>
        public GestureResultView GestureResultView { get; private set; }

        /// <summary>
        /// Gets or sets the body tracking ID associated with the current detector
        /// The tracking ID can change whenever a body comes in/out of scope
        /// </summary>
        public ulong TrackingId
        {
            get
            {
                return this.vgbFrameSource.TrackingId;
            }
```

```
            set
            {
                if (this.vgbFrameSource.TrackingId != value)
                {
                    this.vgbFrameSource.TrackingId = value;
                }
            }
        }

        /// <summary>
        /// Gets or sets a value indicating whether or not the detector is currently
paused
        /// If the body tracking ID associated with the detector is not valid, then the
detector should be paused
        /// </summary>
        public bool IsPaused
        {
            get
            {
                return this.vgbFrameReader.IsPaused;
            }

            set
            {
                if (this.vgbFrameReader.IsPaused != value)
                {
                    this.vgbFrameReader.IsPaused = value;
                }
            }
        }

        /// <summary>
        /// Disposes all unmanaged resources for the class
        /// </summary>
        public void Dispose()
        {
            this.Dispose(true);
            GC.SuppressFinalize(this);
        }

        /// <summary>
        /// Disposes the VisualGestureBuilderFrameSource and
VisualGestureBuilderFrameReader objects
        /// </summary>
        /// <param name="disposing">True if Dispose was called directly, false if the GC
handles the disposing</param>
        protected virtual void Dispose(bool disposing)
        {
            if (disposing)
            {
                if (this.vgbFrameReader != null)
                {
                    this.vgbFrameReader.FrameArrived -= this.Reader_GestureFrameArrived;
                    this.vgbFrameReader.Dispose();
                    this.vgbFrameReader = null;
                }
```

```
                if (this.vgbFrameSource != null)
                {
                    this.vgbFrameSource.TrackingIdLost -= this.Source_TrackingIdLost;
                    this.vgbFrameSource.Dispose();
                    this.vgbFrameSource = null;
                }
            }
        }

        /// <summary>
        /// Handles gesture detection results arriving from the sensor for the associated
body tracking Id
        /// </summary>
        /// <param name="sender">object sending the event</param>
        /// <param name="e">event arguments</param>
        private void Reader_GestureFrameArrived(object sender,
VisualGestureBuilderFrameArrivedEventArgs e)
        {
            VisualGestureBuilderFrameReference frameReference = e.FrameReference;
            using (VisualGestureBuilderFrame frame = frameReference.AcquireFrame())
            {
                if (frame != null)
                {
                    // get the discrete gesture results which arrived with the latest
frame
                    IReadOnlyDictionary<Gesture, DiscreteGestureResult> discreteResults =
frame.DiscreteGestureResults;

                    // get the continuous gesture result which arrived with the latest
frame
                    IReadOnlyDictionary<Gesture, ContinuousGestureResult>
continuousResults = frame.ContinuousGestureResults; //ADDED BY ME

                    if (discreteResults != null)
                    {
                        // we only have one gesture in this source object, but you can
get multiple gestures
                        foreach (Gesture gesture in this.vgbFrameSource.Gestures)
                        {
                            if (gesture.Name.Equals(this.seatedGestureName) &&
gesture.GestureType == GestureType.Discrete)
                            {
                                DiscreteGestureResult result = null;
                                discreteResults.TryGetValue(gesture, out result);

                                if (result != null)
                                {
                                    // update the GestureResultView object with new
gesture result values
                                    this.GestureResultView.UpdateGestureResult(true,
result.Detected, result.Confidence);
                                }
                            }
                        }
                    }

                    //ADDED BY ME
                    if (continuousResults != null)
```

```
                    {
                        foreach (Gesture gesture in this.vgbFrameSource.Gestures)
                        {
                            if (gesture.Name.Equals(this.continuousGestureName) &&
gesture.GestureType == GestureType.Continuous)
                            {
                                ContinuousGestureResult result = null;
                                continuousResults.TryGetValue(gesture, out result);

                                if (result != null)
                                {
                                    this.progressResult = result.Progress;
                                    //Console.WriteLine("progressResult: " +
progressResult.ToString());

this.GestureResultView.UpdateContinuousGestureResult(result.Progress);
                                }
                                else
                                {
                                    this.progressResult = -10.0f;
                                }
                            }
                        }
                    }
                }
            }

        /// <summary>
        /// Handles the TrackingIdLost event for the VisualGestureBuilderSource object
        /// </summary>
        /// <param name="sender">object sending the event</param>
        /// <param name="e">event arguments</param>
        private void Source_TrackingIdLost(object sender, TrackingIdLostEventArgs e)
        {
            // update the GestureResultView object to show the 'Not Tracked' image in the
UI
            this.GestureResultView.UpdateGestureResult(false, false, 0.0f);
        }
    }
}
```

GestureResultView.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.ComponentModel;
using System.Runtime.CompilerServices;
using System.Windows.Media;
using System.Windows.Media.Imaging;

namespace MyOwnVGBTest2
{
    /// <summary>
    /// Stores discrete gesture results for the GestureDetector.
    /// Properties are stored/updated for display in the UI.
    /// </summary>
    class GestureResultView : INotifyPropertyChanged
    {
        /// <summary> Image to show when the 'detected' property is true for a tracked
body </summary>
        private readonly ImageSource seatedImage = new BitmapImage(new
Uri(@"Images\Seated.png", UriKind.Relative));

        /// <summary> Image to show when the 'detected' property is false for a tracked
body </summary>
        private readonly ImageSource notSeatedImage = new BitmapImage(new
Uri(@"Images\NotSeated.png", UriKind.Relative));

        /// <summary> Image to show when the body associated with the GestureResultView
object is not being tracked </summary>
        private readonly ImageSource notTrackedImage = new BitmapImage(new
Uri(@"Images\NotTracked.png", UriKind.Relative));

        /// <summary> Array of brush colors to use for a tracked body; array position
corresponds to the body colors used in the KinectBodyView class </summary>
        private readonly Brush[] trackedColors = new Brush[] { Brushes.Red,
Brushes.Orange, Brushes.Green, Brushes.Blue, Brushes.Indigo, Brushes.Violet };

        /// <summary> Brush color to use as background in the UI </summary>
        private Brush bodyColor = Brushes.Gray;

        /// <summary> The body index (0-5) associated with the current gesture detector
</summary>
        private int bodyIndex = 0;

        /// <summary> Current confidence value reported by the discrete gesture
</summary>
        private float confidence = 0.0f;

        /// <summary> True, if the discrete gesture is currently being detected
</summary>
        private bool detected = false;

        /// <summary> Image to display in UI which corresponds to tracking/detection
state </summary>
        private ImageSource imageSource = null;
```

```
        /// <summary> True, if the body is currently being tracked </summary>
        private bool isTracked = false;

        /// <summary> The float value of the continuous progress </summary>
        private float progress = 0.0f; //ADDED BY ME

        /// <summary>
        /// Initializes a new instance of the GestureResultView class and sets initial
property values
        /// </summary>
        /// <param name="bodyIndex">Body Index associated with the current gesture
detector</param>
        /// <param name="isTracked">True, if the body is currently tracked</param>
        /// <param name="detected">True, if the gesture is currently detected for the
associated body</param>
        /// <param name="confidence">Confidence value for detection of the 'Seated'
gesture</param>
        public GestureResultView(int bodyIndex, bool isTracked, bool detected, float
confidence, float progress)
        {
            this.BodyIndex = bodyIndex;
            this.IsTracked = isTracked;
            this.Detected = detected;
            this.Confidence = confidence;
            this.ImageSource = this.notTrackedImage;
            this.Progress = progress;
        }

        /// <summary>
        /// INotifyPropertyChangedPropertyChanged event to allow window controls to bind
to changeable data
        /// </summary>
        public event PropertyChangedEventHandler PropertyChanged;

        /// <summary>
        /// Gets the body index associated with the current gesture detector result
        /// </summary>
        public int BodyIndex
        {
            get
            {
                return this.bodyIndex;
            }

            private set
            {
                if (this.bodyIndex != value)
                {
                    this.bodyIndex = value;
                    this.NotifyPropertyChanged();
                }
            }
        }

        /// <summary>
        /// Gets the body color corresponding to the body index for the result
        /// </summary>
```

```
        public Brush BodyColor
        {
            get
            {
                return this.bodyColor;
            }

            private set
            {
                if (this.bodyColor != value)
                {
                    this.bodyColor = value;
                    this.NotifyPropertyChanged();
                }
            }
        }

        /// <summary>
        /// Gets a value indicating whether or not the body associated with the gesture
detector is currently being tracked
        /// </summary>
        public bool IsTracked
        {
            get
            {
                return this.isTracked;
            }

            private set
            {
                if (this.IsTracked != value)
                {
                    this.isTracked = value;
                    this.NotifyPropertyChanged();
                }
            }
        }

        /// <summary>
        /// Gets a value indicating whether or not the discrete gesture has been detected
        /// </summary>
        public bool Detected
        {
            get
            {
                return this.detected;
            }

            private set
            {
                if (this.detected != value)
                {
                    this.detected = value;
                    this.NotifyPropertyChanged();
                }
            }
        }
```

```
        /// <summary>
        /// Gets a float value which indicates the detector's confidence that the gesture
is occurring for the associated body
        /// </summary>
        public float Confidence
        {
            get
            {
                return this.confidence;
            }

            private set
            {
                if (this.confidence != value)
                {
                    this.confidence = value;
                    this.NotifyPropertyChanged();
                }
            }
        }

        /// <summary>
        /// Gets an image for display in the UI which represents the current gesture
result for the associated body
        /// </summary>
        public ImageSource ImageSource
        {
            get
            {
                return this.imageSource;
            }

            private set
            {
                if (this.ImageSource != value)
                {
                    this.imageSource = value;
                    this.NotifyPropertyChanged();
                }
            }
        }

        public float Progress
        {
            get
            {
                return this.progress;
            }

            private set
            {
                if (this.progress != value)
                {
                    this.progress = value;
                    this.NotifyPropertyChanged();
                }
            }
        }
```

```
        /// <summary>
        /// Updates the values associated with the discrete gesture detection result
        /// </summary>
        /// <param name="isBodyTrackingIdValid">True, if the body associated with the
GestureResultView object is still being tracked</param>
        /// <param name="isGestureDetected">True, if the discrete gesture is currently
detected for the associated body</param>
        /// <param name="detectionConfidence">Confidence value for detection of the
discrete gesture</param>
        public void UpdateGestureResult(bool isBodyTrackingIdValid, bool
isGestureDetected, float detectionConfidence)
        {
            this.IsTracked = isBodyTrackingIdValid;
            this.Confidence = 0.0f;

            if (!this.IsTracked)
            {
                this.ImageSource = this.notTrackedImage;
                this.Detected = false;
                this.BodyColor = Brushes.Gray;
            }
            else
            {
                this.Detected = isGestureDetected;
                this.BodyColor = this.trackedColors[this.BodyIndex];

                if (this.Detected)
                {
                    this.Confidence = detectionConfidence;
                    this.ImageSource = this.seatedImage;
                }
                else
                {
                    this.ImageSource = this.notSeatedImage;
                }
            }
        }

        public void UpdateContinuousGestureResult(float progressResult)
        {
            if(!this.IsTracked){
                this.Progress = 0.0f;
            }
            else
            {
                this.Progress = progressResult;
                Console.WriteLine(this.Progress.ToString());
            }
        }

        /// <summary>
        /// Notifies UI that a property has changed
        /// </summary>
        /// <param name="propertyName">Name of property that has changed</param>
        private void NotifyPropertyChanged([CallerMemberName] string propertyName = "")
        {
            if (this.PropertyChanged != null)
```

```
            {
                this.PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
            }
        }
    }
}
```