Kinect Development

for Stroke Rehabilitation

Computer Science

Edward Jack Chandler (120232420)

Project Supervisors: Dr Graham Morgan,

Dr Gary Ushaw, Dr William Blewitt

Abstract

This dissertation will be look at how a game using the Microsoft Kinect as a motion tracker, can help detect physical ailments and furthermore provide physiotherapeutic exercises that a user can follow. It will involve looking into how stroke victims' movements differ from others and how to create a fun game based upon rehabilitative exercises.

Contents

Declaration

"I declare that this dissertation represents my own work, except where otherwise stated."

Signed: _____

Dated: _____

Acknowledgements

I would like to thank my supervisors, Dr Gary Ushaw, Dr William Blewitt and Dr Graham Morgan, for providing me support and advice throughout my project and writing this dissertation. I would also like to thank Aaron Stalley, for helping me film instructional videos during the implementation phase.

1 Introduction

This section introduces the reader to the basis of the project, and includes the structure of the dissertation.

1.1 Motivation

In England alone, a study from 2013 showed that 900,000 people were living with the effects of a stroke [**1**]. Out of these people, 'around 80% of stroke survivors experienced movement problems' [**2**], coming from the weaknesses in their muscles leading to difficulty moving their limbs. One of the current ways that a stroke survivor can help regain movement to their body, is to seek help from physiotherapy. This involves performing specific exercises with the impaired parts of the body, over a period of recovery.

The exercises that stroke victims perform are very repetitive, from the nature of their purpose; to strengthen specific parts of the body. Because of this, survivors might find them boring or tedious and could be tempted to skip doing their exercises for the day. This might lead to a cycle where people will not recover as quickly or as fully as they could've done.

Obviously there is support for stroke survivors once they have left the hospital and have gone through their initial steps of recovery, however, once on their own without the help of a trained professional, a person might find it hard to repeat the exercises once in a home environment. A clear visual reference is needed which you can achieve with a video of the exercises, however even then, there is no recognition that the exercises are being performed correctly. Furthermore, a patient might have to visit the hospital more often to get advice on the exercises if they are not performing them correctly, which will cost time and money.

1.2 Purpose

In my project, I have chosen to tackle these problems by using motion tracking with a game for the Microsoft Kinect that can be used in physiotherapy. The game would address the fact that usual physiotherapy exercises can be boring, by trying to be fun and entertain the patient whilst concurrently helping the person recover. The game will give a point of reference to the patient by showing how to do the exercises each time a new one is suggested. Exercises will be split into upper, and lower exercises, in which the lower exercises would only be attempted if the subject can support themselves. The Microsoft Kinect is a relatively cheap piece of hardware, which is able to track skeletons accurately enough for a game such as this, so is highly accessible.

Because many people are affected by strokes, it seems sensible to have different ways of performing the exercises given; some might find it reassuring that they will see a doctor each week to help them with the exercises, however some might find that a game will help and motivate them to complete their exercises and be on the road to recovery.

Something that has rarely been touched upon with games in rehabilitation is the fact that stroke patients often fall during their period of rehabilitation. A study showed that as many as '39% of patients suffered falls' [**3**] during rehabilitation, of which some fell many times. Because of this, my

project will use a feature of the Kinect called ‘Seated Mode’, which allows skeleton tracking to track the upper half of the skeleton whilst the subject is seated.

1.3 Project Aim and Objectives

Project Aim:

- **Aid stroke rehabilitation by creating a game that can act as physiotherapy.**

Project Objectives:

- *Understand how the physical movements of a stroke victim differs to a non-victim.*
- *Build an application for Kinect that can record a body movement twice, and compare each action.*
- *Create a fun game to differ from regular physiotherapy and provide positive feedback for motivational purposes.*
- *Simulate physiotherapy in a stroke rehabilitation situation using the Kinect sensor and game.*
- *Evaluate my findings.*

1.4 Dissertation Structure

Introduction

Introduces the dissertation providing the current problems involved with the subject matter and proceeds to explain the project's aim and objectives.

- Motivation
- Purpose
- Project Aim and Objectives
- Dissertation Structure

Research

Features research into the project area.

- Motion Tracking Technologies including Kinect
- Stroke Rehabilitation Issues
- Stroke Rehabilitation Exercises

Design

An explanation of the design of my application and algorithms, including my software engineering approach.

Implementation

Here I will explain the execution of my designs, as well as show any tools and technologies I have used in the project.

Testing and Evaluation

An evaluation of my application

- Unit Testing
- Application Testing
- Exercise Testing

Conclusion

- Fulfilment of objectives
- Problems Encountered
- Thoughts
- Future Work

2 Research

This section will show the research that has been done for this dissertation.

My research was split into three main parts:

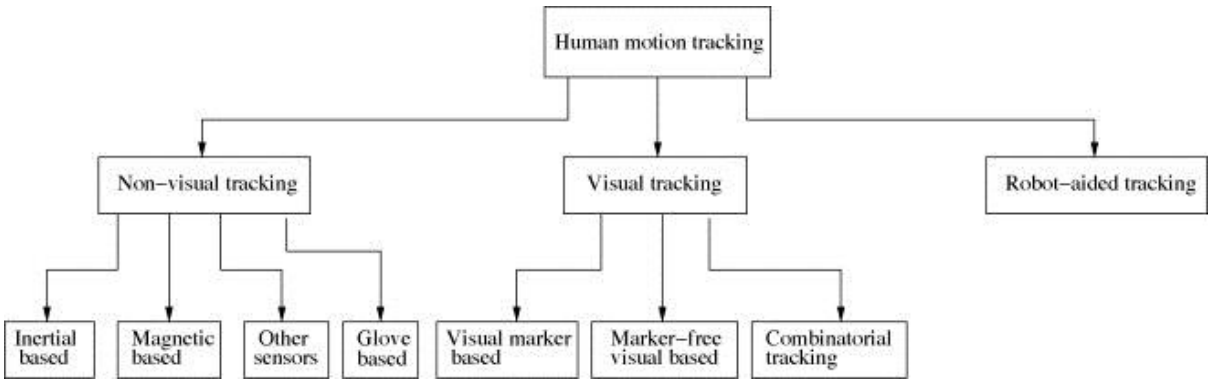
- Motion Tracking Technologies and Kinect
- Stroke Rehabilitation Issues
- Stroke Rehabilitation Exercises
- Research on papers about stroke rehabilitation, motion tracking technologies, the Kinect, and rehabilitation exercises, was done mostly by searching through google scholar. Websites found for stroke exercises were found via the 'Stroke Association' [**13**]; a hub of information and support for stroke victims.

2.1 Background Research

2.11 Motion Tracking Technologies in a Home Based Stroke Rehabilitation System

Rehabilitation is the act of restoring something to its original state, and in the case of stroke rehabilitation, is a dynamic process that can take a lot of time and hard work. The main basis of stroke rehabilitation is to "correct any undesired motion behaviour in order to reach an expectation (eg. Ideal position)". [**3**] As with many things, technology can improve, and possibly "accelerate recovery" [**3**] during stroke rehabilitation.

The desire for motion tracking in a stroke rehabilitation setting comes from the need for stroke patients to be monitored, and prompted or assisted to change to meet a correct motion. This, in several sessions over a period of time, can help a patient regain motion in their paralysed limbs, and be a great tool in recovery. How well can these tracking systems keep track of human motion though? And which can be used best in a stroke rehabilitation setting?



*"Classification of human motion tracking using sensor technologies" [**3**]*

Several different types of human motion tracking systems exist, divided into three categories. Non-visual tracking uses sensors attached to the human body, to track motion. This can include attaching

accelerometers to the body and analysing the data to track the motion. Magnetic sensors can also be used, as well as glove based analysis, which "transduces finger flexion and abduction into electrical signals, to determine hand pose" [**3**]. Whilst these techniques have many advantages, they also have some drawbacks, for example, using accelerometers for inertial based tracking can suffer from the "drift problem", which is when an "accumulation of small errors occur" [**9**] and extra computation to correct the errors is needed. Also, magnetic based systems can suffer from latency and jitter, which is poor for proving real time comparisons of motions.

Visual tracking, features two main subcategories, marker based and marker free. Marker based tracking involves attaching reflective markers to a body, often seen in filmography motion capture. The markers either reflect incoming light in a 'passive' method, or produce infrared light in an 'active' method. In both methods the light is gathered by a camera, and the data is used to track motion. Due to the unique appearance of the markers, the systems are "capable of minimising the uncertainty of subject's movements", however they also require a lot of space for camera set up, as well as involving the impracticality of the markers themselves.
Marker free tracking on the other hand, focuses on tracking specific features of human bodies. A single, or multiple cameras will use specific algorithms to analyse a body's structure, and track motion. 2D or 3D approaches can be undertaken, using "prior knowledge of human movement" [**3**] to extract significant characteristics of a body, and match them across images. Marker free tracking is seen as more convenient than tracking with markers, since it does not require the user to don a suit of some sort, however if multiple cameras are needed for tracking, this can also be impractical because of the amount of space needed.

2.12 Gaming Rehabilitation

The tracking of a human obviously involves human interaction, therefore video games, another human interaction, can go hand in hand with it, not only to entertain users with hardware such as the famous Wii remote [**24**], but to also help users through rehabilitative gaming. Rehabilitative gaming systems [**23**] can "combine hypotheses on the aftermath" of stroke, or other conditions that affect the physical systems of the body, and the mechanisms of recovery, with an interactive game, often using the "parameters and timing during the gaming process for further medical analysis". [**25**] This means that rehabilitative gaming can be used as a powerful tool to correct undesired motions as stated earlier. Another benefit of gaming in a rehabilitation setting, is that difficulty can be incorporated to act as a learning tool, as well as "the potential to produce a unique rehabilitation learning experience" that can provide motivation for the user.

2.13 Kinect for Windows

Microsoft Kinect Sensor [**10**]

For my application, I have decided to use the Microsoft Kinect, a sensor that can provide marker free tracking via its depth camera. Here I will introduce the Kinect as a hardware, and show the capabilities the technology has to aid the stroke rehabilitation process

The Kinect sensor developed by Microsoft is an advanced game controller, which can be used for means further than gaming. The Kinect contains a depth sensor, a colour camera, and four-microphone setup "that can provide full-body 3D motion

capture". With this technology, an infrared camera can obtain a depth map of a human body, and use this data to infer body position, which can then "accurately predict 3D positions of body joints." With these joints, the Kinect is able to entirely track the full skeleton of a human body. Furthermore, the Kinect has two tracking modes: seated, and default. "The seated tracking mode is designed to track people who are seated on a chair or couch, or whose lower body is not entirely visible to the sensor. The default tracking mode, in contrast, is optimized to recognize and track people who are standing and fully visible to the sensor."[**16**] These modes are integral to different exercises that will be shown later in the dissertation.

Since the Kinect was designed to fit into homes as a game controller, the sensor is able to see people standing between 0.8 meters and 4.0 meters away, which includes the average living room space able to a user. In terms of space this makes the Kinect a great tool to use at home for stroke rehabilitation. The device is also incredibly cheap compared to other types of motion capturing technology discussed earlier.

During my research, I found a paper specifically comparing the low cost Kinect to a high fidelity marker based system: the 'OptiTrack optical motion capture system'. In a physical rehabilitation setting, the Kinect and OptiTrack system are compared in several tests to investigate if the low-cost Kinect can be used 'competitively' against the high-cost and non-portable OptiTrack. Apart from movement at the shoulder, the Kinect does track the body competitively against the OptiTrack, and shows that my project can use the Kinect as a low-cost tool in the physiotherapeutic game, and will be accurate enough for it to work well.

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

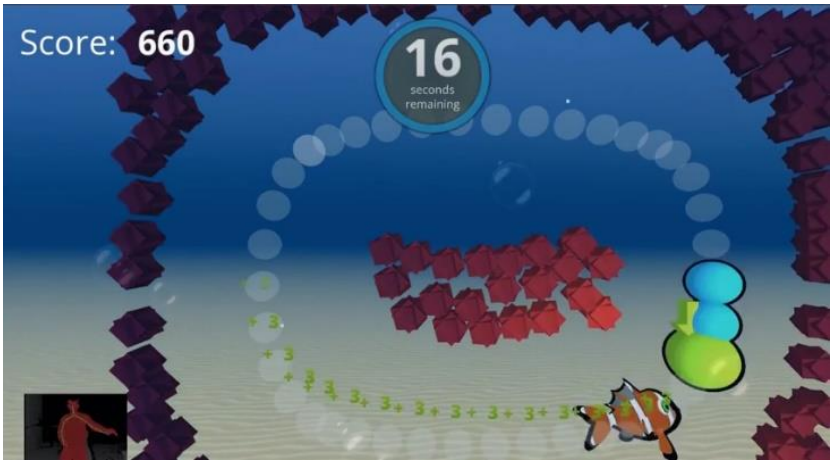
Since I had never used the Kinect before, I researched some tools and guides that could help set up a basic skeletal tracking environment. This led me to find the Kinect for Windows Toolkit within the Kinect SDK [**6**], which provided some sample Kinect applications and their source code which would help me develop my own application. I also found the Kinect for Windows online 'Quickstart Series' [**21**], by the Microsoft community site 'Channel 9'. This series gives some tutorials in how to set up a development environment, as well as how to get started with tracking a skeleton.

2.14 Existing Systems for Kinect

A range of existing tools already exist in the market of rehabilitative gaming. 'SeeMe' is a "clinician-controlled" rehabilitative gaming tool, designed to aid the rehabilitation process, providing several games for different areas of rehabilitation, such as balance and equilibrium, or even games for neurological problems such as movement awareness. Research was put into the use of a novelty system [**28**], in this case 'SeeMe', for rehabilitation, and the results were positive, showing that the system could detect evidence that standard paper and pencil tests could not, and that "the treatment was well received by subjects, who enjoyed the experience." The downside is, this system is designed to be controlled by a clinician at a clinic, and therefore cannot be used by someone on their own, in their home.

'SeeMe' Rehabilitative *Gaming Tool*

One of the most popular reasons to use the Kinect for rehabilitative gaming, stems from the fact that it is an affordable piece of equipment to have in a user's home [**26**]. 'Jintronix' [**28**] is a platform that uses the Kinect for Windows v2 designed for physical occupational therapy. The system uses virtual games with the motion sensing capabilities of the Kinect to enforce common rehabilitation movements, to try and offer a "fun and effective" tool for physical rehabilitation. The program also allows a user to track their progress during their rehabilitation period.



'Jintronix' Rehabilitative *Gaming Tool*

2.15 Problems after a Stroke and Rehabilitation

Having suffered from a stroke, victims can face many difficulties, including cognitive problems, problems with their balance, fatigue, and emotional problems, but the most common effects of a stroke are physical. In fact, as stated earlier, "around 80% of stroke survivors experience movement problems". [**13**] Weakness, numbness and stiffness are all something a stroke victim can suffer from, in any body part, and mostly the arms or legs.
A side effect from these ailments is spasticity, when changes in muscle tone have occurred in their body. This leads to developing "very tight, stiff muscles". [**13**] If muscles are too stiff, sometimes a "permanent shortening of the muscles" [**13**] can occur called a contracture.

After a stroke, whilst most people will make significant improvements “in the first few months”, [**13**] the recovery process can continue for a long time, and will need continuous therapy sessions to help strengthen weak muscles in the affected limbs. This is often carried out in one-to-one physiotherapy sessions, but can also feature having exercises to practice on your own.

Another problem stroke survivors can have whilst in the recovery process, is having the motivation to perform these exercises, not only because they are exhausting, but if a survivor is performing exercises on their own it can be hard to find the drive to perform the exercises every day. A paper, *“Qualitative analysis of stroke patients' motivation for rehabilitation”* goes further into the motivational aspects of a patient during rehabilitation.
This paper goes deep into the motivational thoughts and status (high or low) of a stroke patient, whilst in hospital. A sample of stroke patients are interviewed on a number of topics, such as ‘confidence about making a good recovery’ and ‘ideas about the patient’s role in rehabilitation’. All patients deemed at a high level of motivation ‘believed that rehabilitation had the most important role’ (in their recovery), and that ‘learning to perform rehabilitation exercises in the manner specified by therapists’ was very important. This paper shows that all high motivation patients during rehabilitation have the desire to understand the exercises, and it is clear that showing patients how to perform an exercise each time in my application will potentially increase their motivation.

Further research into the stroke rehabilitation process, highlighted another problem that can face patients during recovery: the danger of a patient falling. A study [**15**] investigating falls in rehabilitation, took one hundred and sixty one patients, of which sixty-two fell during the rehabilitation period. 153 falls were recorded during the study and the most frequent location for falls was the patient’s room. The study showed that there was a need for including seated exercises in my application, since many falls do occur during the rehabilitation period especially in the subject’s room, on their own. Ideally a patient would be able to use my application on their own, because this was one of the reasons the Kinect was chosen as a cheap alternative that could reside in their homes. So utilising the Kinect’s seated tracking mode, if a patient can support their weight on a chair, a fall would not be able to happen.

My application is designed to be used as a tool to help physiotherapy during rehabilitation, and not as a complete replacement of using a physiotherapist or seeing a doctor. It is designed as a motivational tool to encourage the user to keep exercising their weak limbs at home.

2.16 Types of Stroke Rehabilitation Exercises

I studied the traditional methods and exercises that a stroke survivor could go through to design the types of motion I would feature in my application. Firstly, I discovered there were three types of motion that could be used in rehabilitation.

<u>Passive Range of Motion</u>

The first range of motion that a stroke survivor may experience with a particular limb, is the passive range of motion. This motion refers to “an external force moving a body part rather than it moving on its own volition.” [**4**] A survivor with full paralysis in a given limb will have only a passive range of motion in that limb, and will have to have something to move it, to “maintain flexible joints and prevent joint contracture”. [**4**]

Active Assistive Range of Motion

The active assistive range of motion occurs when a patient's weak limb has some or most of its motion, but has to be assisted through movement. For example, a person can lift their arm seventy degrees up, but needs to use their other arm to push it to a full ninety degree angle. This range of motion can help strengthen "a limb that does not yet have full range of motion." [**4**]

Active Range of Motion

The active range of motion is when a patient can move a given body part on their own *without* assistance. The body part will not be at full strength and may tire, but the patient will have the capability to move it in a full range. Exercises in the active range of motion "helps promote joint flexibility, strengthening, and increased muscular endurance". [**4**]

Ranges of Motion Summary

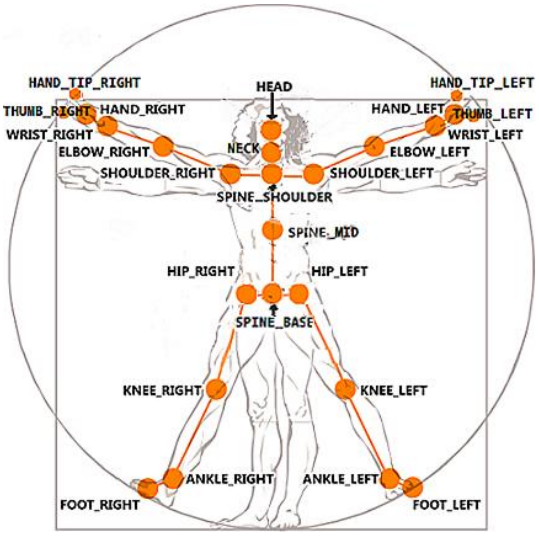
Whilst the *passive* range of motion is an important range to consider, a therapist is often needed to help with the movement of limbs, and even though the Kinect has the capabilities to track two skeletons, the Kinect might have a problem differentiating the joints of the patient and therapist, since they would be in such close proximity. For this reason I rejected using the passive range of motion in the exercises my application will track.

I chose the *active assistive* range of motion to be the basis around my exercises. Since my application is based on the accuracy of a user's motion compared to a specific 'model' motion based on my own actions, a score will appear depending on how close you have come to reaching this motion. With a user that has an active assistive range of motion, the idea is that they will not receive assistance, and will be given a score purely on their merit. This way progress can be tracked if exercises are performed regularly, and an accurate score reflecting on a user's ability will be given. Exceptions can be made if, for example, a person needed help raising their weak arm, they could push it with their strong arm until they can start the exercise. This way, a user's confidence is not knocked because of the fact they couldn't even start one of the exercises.

The *active* range of motion, is also considered in my application, since a user has full movement of the specific limb. A user performing the active range of motion should get close to a full score in the exercises, since they have the ability to match the model motion. Resistance training can also be incorporated into the active range of motion. Instead of getting a full score with no assistance, the user will incorporate small weights to increase the strength of the muscles in the limb, and will increase the difficulty of the motion. Dumbbells have been used in Kinect games before, such as 'Nike+ Kinect Training', which shows that the use of small dumbbells will not interfere with the tracking of certain joints during the application. [**5**]

2.17 Accessible Joints vs Exercises

To look further into the range of motion I would be dealing with in my application, I created a table compiling of the list of joints that the Kinect could track, and a list of body parts that a stroke survivor might have issues with. From these I reviewed whether it would be relevant to track the joint in my application.



"This figure shows the skeleton that is made up of each of the (Kinect) Joint Types."

Kinect JointType	Patient Body Part Affected	Review	Used for exercises?
AnkleLeft	Left Foot	The ankle is the pivot of which the foot can move up and down from, tracking it does not allow for comparison of foot movements.	No
AnkleRight	Right Foot	*See above*	No
ElbowLeft	Left Arm	The elbow is the pivot of the lower arm, it could be used to compare whole movements of an arm if linear, however the Hand JointType is best suited for this.	No
ElbowRIght	Right Arm	*See above*	No
FootLeft	Left Foot / Left Leg / Hip	Comparing movements of the foot with the Kinect is difficult since the foot's movement range is small, and comparing small actions needs accurate precision. However, foot joints can be tracked and used well as an extremity to compare leg movements, since the foot has a wide range of movement when the whole leg or part of the leg is moved, which is good for comparing movements.	Yes
FootRight	Right Foot / Right Leg / Hip	*See above*	Yes
HandLeft	Left Hand / Left Arm	Comparing movements of the hand with the Kinect is difficult since the hand's movement range is small, and comparing small actions needs accurate precision. However, hand joints can be tracked and used well as an extremity to compare leg movements, since the hand has a wide range of movement when the whole arm or part of the arm is moved, which is good for comparing movements..	Yes
HandRight	Right Hand / Right Arm	*See above*	Yes
HandTipLeft	Left Hand	Comparing movements of the hand with the Kinect is difficult since the hand's movement range is small, and comparing small actions needs accurate precision.	No
HandTipRight	Right Hand	*See above*	No
Head	Face	Muscles in the face do suffer from weakness but since it is only the head that can be tracked the Kinect is not accurate enough to notice small facial movements.	No
HipLeft	NA	NA	No
HipRight	NA	NA	No
HipCentre	NA	NA	No
KneeLeft	Left Leg	The knee is the pivot of the lower leg, it could be used to compare whole movements of a leg if linear, however the Foot JointType is best suited for this.	No
KneeRight	Right Leg	*See above*	No
Neck	Neck	Patients improve their neck strength by turning their heads in an exercise, but the Kinect cannot detect rotation in the neck, and therefore the movement cannot be compared.	No
ShoulderLeft	Left Shoulder	Patients improve their shoulder strength by shrugging their shoulder in an exercise, but since the shoulder's movement range is small, comparing the actions needs accurate precision that the Kinect cannot provide.	No
ShoulderRight	Right Shoulder	*See above*	No
SpineBase	NA	NA	No
SpineMid	NA	NA	No
SpineShoulder	NA	NA	No
ThumbLeft	Left Hand	Comparing movements of the thumb with the Kinect is difficult since the thumb's movement range is small, and comparing small actions needs accurate precision.	No
ThumbRight	Right Hand	*See above*	No

WristLeft	Left Hand	The wrist is the pivot of the hand, and so if comparing hand movements, will stay stationary.	No
WristRight	Right Hand	*See above*	No

By reviewing all of the various JointTypes, I am able to derive that I will include exercises to strengthen both arms, and both legs, by tracking the hands and feet.

2.2 Designing Rehabilitation Exercises

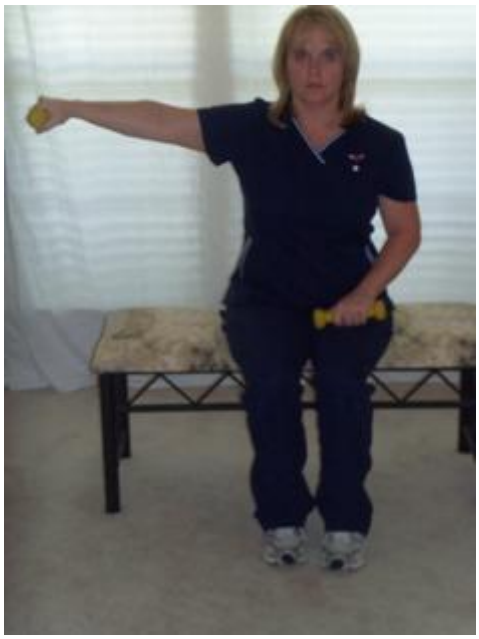
In this section I will choose some rehabilitation exercises for the arms and legs that I will implement into my Kinect application. The exercises have been taken from a website that provides stroke rehabilitation exercises to perform at home. [**17**] [**18**]

2.21 Arm Exercises

To tackle weakness on both sides of the body in the arms, I have found a particular exercise that can be used in the, **Active Assistive**, and **Active** ranges of motion. As stated before, this exercise can also be performed with dumbbells for added resistance. Exercises on the arms can also be performed sat down, if a patient needs to support their weight.

The exercise is known as the **Shoulder Abduction**, and is mirrored to allow exercises to be performed on both sides of the body.

"Instructions:
Keeping arm straight, lift arm out to side to shoulder height.
Return arm to side."



"Shoulder Abduction"

2.22 Leg Exercises

To tackle weakness on both sides of the body in the legs, another exercise was found that can be used in the, **Active Assistive**, and **Active** ranges of motion. Similar to the arm exercise, this exercise can also be performed with ankle weights for added resistance. These exercises might require a patient to help themselves balance by holding onto a stable surface.

The exercise is known as the **Hip Abduction**, and is also mirrored to allow exercise to be performed on both sides of the body.

"Instructions:

Holding to a stable surface, lift the affected leg out to the side and back down."

"Hip Abduction"

2.3 Summary

Research into Motion Tracking has shown that the Kinect is a cheap and accurate device in a rehabilitation setting, and that rehabilitative gaming in general can be a great tool to aid rehabilitation, as well as motivation. The ranges of motion were explored, identifying the active assistive, and active ranges of motion as the targets for the application. Then Kinect joints that could be compared with stroke patient movement issues, and some example physiotherapeutic exercises were chosen to use with the application.

3 Design

This section will show the design solutions I have produced for my application, derived from analysing some requirements I created as part of the waterfall model.

I used the waterfall model in my software development process whilst working on my application, and therefore listed some functional, and non-functional requirements that my application would need in order to succeed. The waterfall model was chosen because I have used it with previous projects, and feel confident in its use with smaller projects, like this one. Another advantage of using it in this instance, is that it will give me some solid requirements and deliverables that I can use in conjunction with my aims to design and implement the application's features.

3.1 Functional and Non-Functional Requirements

The functional requirements for my application are as follows:

1. The application will offer a range of exercises to perform, including sat down exercises.
2. The user will be able to choose a specific exercise to perform.
3. The user will be able to see parts of their skeleton tracked on screen.
4. The user will be able to see video, giving instructions on how to perform the exercise.
5. The exercise will not start until the user is properly tracked by the Kinect.
6. The application will show a countdown timer before the exercise, and a timer during the exercise.
7. The user will receive a score to determine how well they performed the chosen exercise.
8. The user will receive motivational feedback in the form of text depending on their score.
9. The code will be documented.

The non-functional requirements for my application are as follows:

1. The application will be fun to use.
2. The UI will be easy to use.

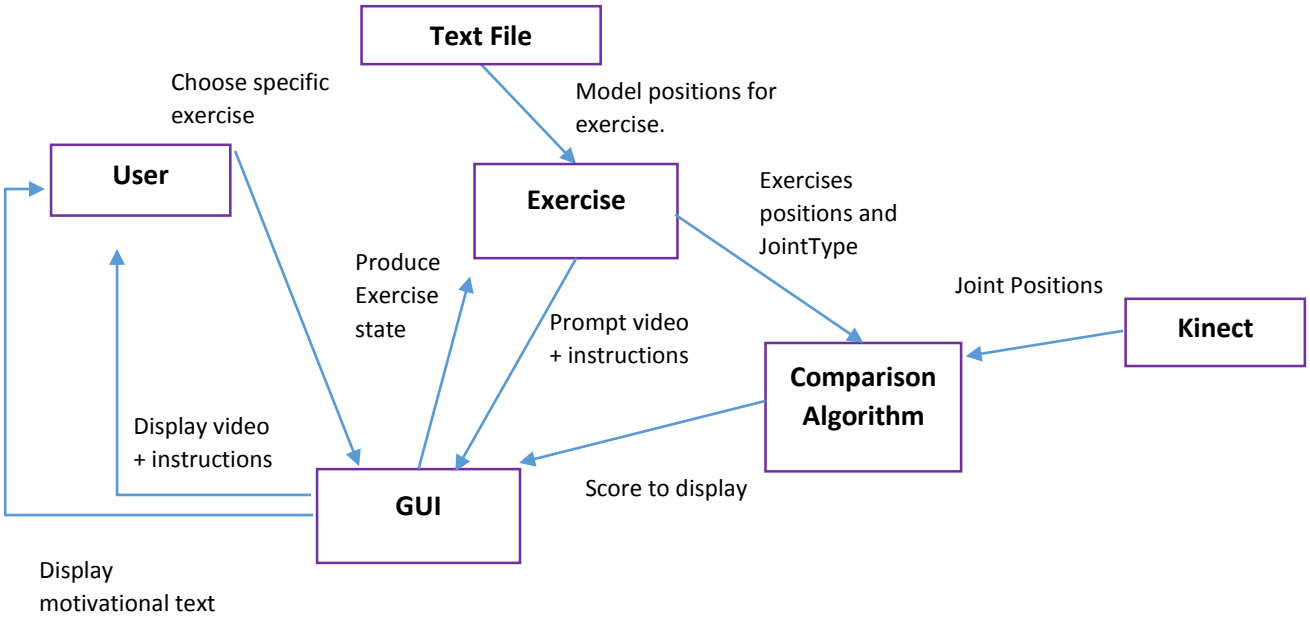
3.2 Exercise Basis

Here I will show the flow of my exercises, that is, how a user will perform one using my application.

I wanted to perform the exercise over 10 seconds so it is a relatively slow pace, and will keep the action smooth. The exercise will not start until the Kinect can track you so I can adhere to my sixth functional requirement.

3.3 High-Level Design

I designed the whole system at a high level, and so I could show how each part of the system interacts with each other.



User: Represents the user that will be using the application. The user will choose a specific exercise to perform.

GUI: The graphical user interface that allows the user to interact with the application. Will also show exercise instructions and video before performing the exercise, and afterwards will give motivational text.

Exercise: Represents a specific exercise a user can perform. Depending on the exercise, it will pass which specific joints the Comparison Algorithm should compare, as well as the model positions for a particular exercise.

Text File: Contains previously recorded model joint positions for a specific exercise, stored line by line in a text file. Exercise will take the joint positions as an input.

Comparison Algorithm: The comparison algorithm will take two sets of joint positions: the model positions for an exercise, and the current positions that the Kinect is detecting. The algorithm will then compare the positions of the two points at every frame, to see how close the user has come to performing the exercise, and pass a score on to the GUI.

Kinect: Represents the Kinect sensor itself, which will track all the current joints of the user's skeleton and provide the position data.

3.4 Tracking Skeletons and Showing Colour Image with Kinect

To show the user their body onscreen to see themselves performing exercises, I need to figure out a way to get data from the Kinect's RGB colour camera onscreen, to attempt to meet my the requirement. To display the RGB data whilst tracking skeletons, and eventually compare the positions, all three available data streams need to be synchronised. To do this, I will use an event when every frame of each different data stream is ready, to take a tracked Skeleton joint, map this onto a point on the depth map, map the depth point onto the colour image, and then use a WPF ellipse shape to show the tracked joint onscreen.

As well as showing the user where their own tracked joints are, the Ellipse Class from 'System.Windows.Shapes' [**8**], would also help me see how accurately the Kinect was tracking the joints on my own body, each time I ran the application.

3.5 Comparison Algorithm

The comparison of two positions using the Kinect is the basis for my whole application and will give the score to adhere my eighth functional requirement, so first of all I designed an algorithm that could accomplish this. The algorithm is split into three different sections:

- Positions differences
- Translating positions
- Comparison review

3.51 Position Differences

The first part of the comparison algorithm is simply calculating the differences between points. The Kinect can track over 20 joints on a user, and provide the 3D coordinates of each joint, every frame. So within a loop, the algorithm will take the current user joint position, and the current frame model exercise position, and find the absolute difference between them.

```
Input: currentPosition, modelPosition
Output: difference

START


difference =

(abs(currentPosition.X – modelPosition.X),
abs(currentPosition.Y – modelPosition.Y),
abs(currentPosition.Z – modelPosition.Z));

return difference;

END
```

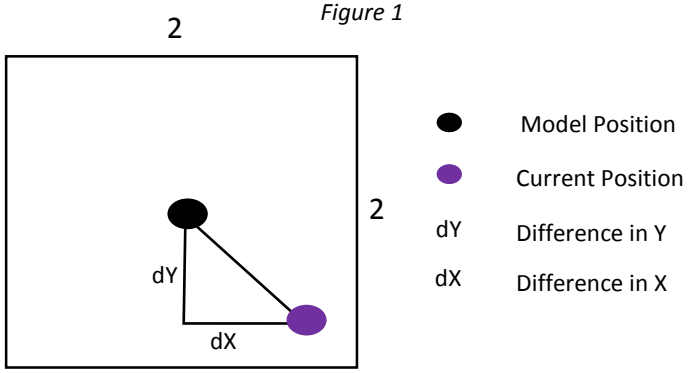
Now that the difference between positions has been found, it will be used to review how close the positions are.

3.52 Comparison Review

I wanted to design a way of using the difference between these points as a percentage, to review how close the user was to the original exercise. This was done by creating a box with the current model position at the centre, gathering whether the current position resided in that box, and if it did, how close the current position was to the centre.

This way, I could gain a percentage representation of how close the current position was to the model position. The percentages range from 100% if the current position is exactly where the model position is, and 1% if the current position is outside of the box. The dimensions of the box can also be changed, if I want to give a more lenient percentage. For example, if the box's dimensions were bigger than in *figure 1*, with the same difference in x and y, a higher relative percentage would be given. I can then fine tune the box dimensions to decide on the accuracy I want the user to be able to match.

Figure 1



```
Input: boxLength, differenceX, differenceY
Output: totalPercentX, totalPercentY
Variables: percentageX = 0, percentageY = 0

START

if (differenceX >= boxLength / 2 OR differenceY >= boxLength / 2)
    totalPercentX = 0;
    totalPercentY = 0;

else
    percentageX = (((boxLength / 2) – differenceX) /
    (boxLength / 2) * 100;
    percentageY = (((boxLength / 2) – differenceY) /
    (boxLength / 2) * 100;
    totalPercentX += percentageX;
    totalPercentY += percentageY;

END
```

Using this percentage value, I can then give the user a score out of ten, by dividing the percentage by 10, and rounding to the nearest whole number.

3.53 Translating Positions

To just use these two parts of the algorithm, an assumption has been made: the user will stand in the exact position that the person creating the model positions did, before performing the exercise. This poses a problem since, if the user were to be for example a meter to the left of where the model exercises were recorded, even if the exercise was performed exactly right, the application would give a bad score since the user's positions were not performed relative to the model's positions. To solve this, I designed a method of translating the model positions to a relevant point, so that the comparison is fair. This involves gaining the hip joint positions of the user, and calculating the difference between the user's hip positions and the model exercise hip starting position. I then used the difference to translate the model exercise positions to where they would be, if the model exercise started where the user did.

```
Input: modelHipStartingPosition, currentHipPosition,
modelPosition
Output: modelPosition
Variables: difference

START

difference = currentHipPosition.difference(modelHipPosition);

modelPosition.X = modelPosition.X - difference.X;
modelPosition.Y = modelPosition.Y – difference.Y;

END
```

Note: in this part, the function 'difference' will not find the absolute difference between points, and could output a negative value. This is because since we are dealing with some positions in a coordinate system that could be negative, subtracting the absolute value will only work for positive values.

With these three different parts, every frame the comparison algorithm will take in a current Position, and a model position, work out the percentage, which I can then output a score with.

3.6 Building Model Position Files

To have the ability to compare the position that a user is currently in, and the same position in the exercise, it requires being able to record myself performing the exercise I have designed and get the model positions for 10 seconds. Since the Kinect records in 30 fps, this equates to gathering one position every frame, for a total of 300 frames for the specific exercise. I used a timer to make sure that my exercise was completed in 10 seconds, in the same flow that my exercises are to be completed, as stated earlier.

```
Input: Joint.Position;
Output: textFile
Variables: currentPosition, frameCounter

START

if (frameCounter) < 300
     currentPosition = Joint.Position;
     textFile.saveValues(currentPosition);
     frameCounter++;

END
```

This is also my design for getting the current Position, to compare against the model exercise, however, I wouldn't need to save the position to the text file and would just output the currentPosition instead.

3.7 Correcting Inaccuracies

Another problem the might have is that once the Kinect has started tracking a user, sometimes inaccuracies can occur with where the Kinect thinks a particular joint is in relation to where it is on screen. To rectify this I had to design a way for the comparison algorithm not to use these values because they were inaccurate. To accomplish this, I saved the previous position that the Kinect had tracked, and put some checks in along with my design for getting the current position. Basically if the position that the Kinect has tracked is not within a certain offset value, the previous position will be used instead.

```
Input: currentPosition, previousPosition
Output: currentPosition;
Variables: offset

START

currentPosition = Joint.Position;

if (!currentPosition.withinPosition(previousPosition), offset)
     currentPosition = previousPosition;

previousPosition = currentPosition;

END
```

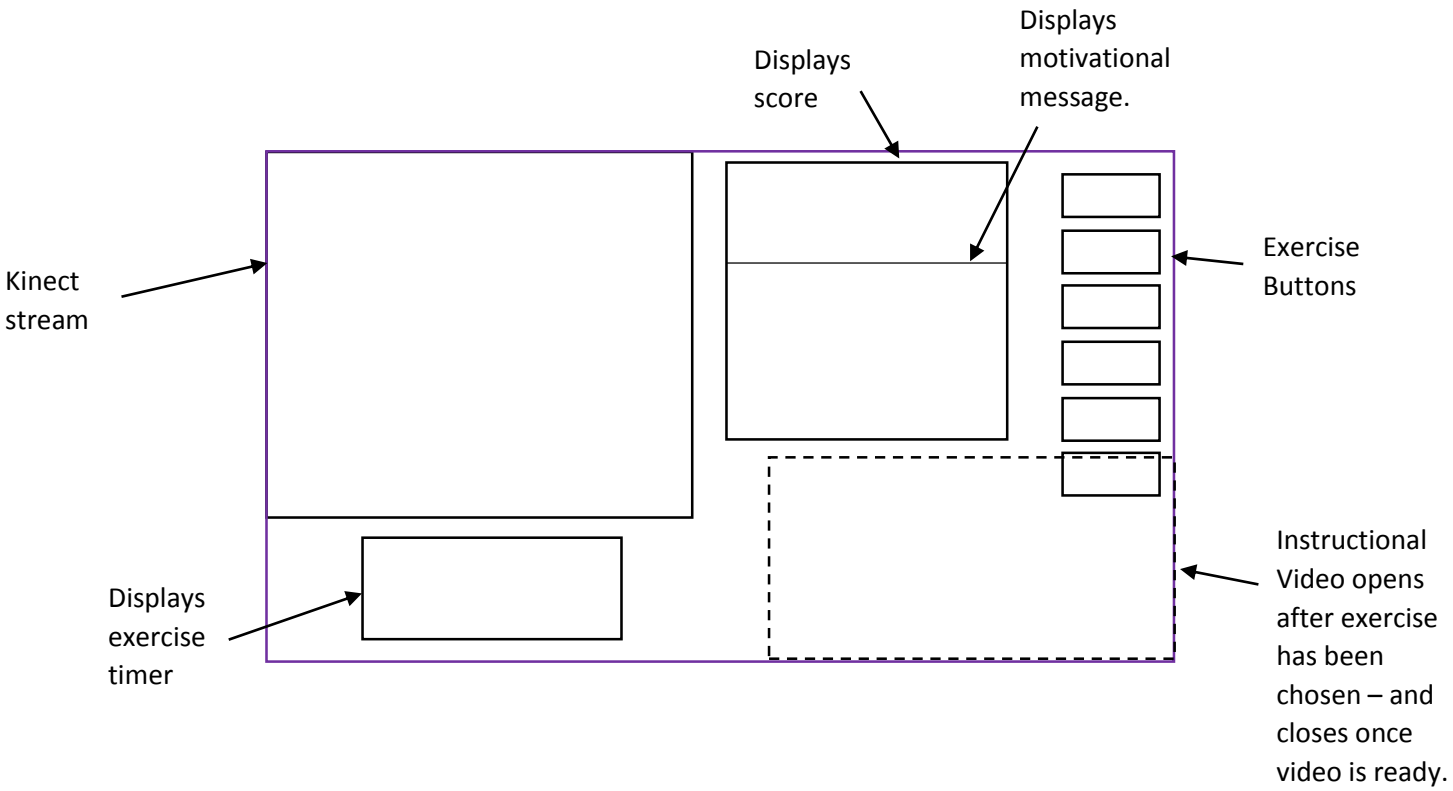
3.8 Timers

I need timers in three areas of my project. First of all a timer will be used whilst the recording of my model joints is being done. This is so that I can roughly time 10 seconds from once I start doing the exercise, so that I can get a good array of positions for my model position text files.
The second timer I will use, is the countdown before a user has to perform an exercise. This is to give the user time to prepare in-between the Kinect starting to track them, and the exercise beginning.
The third timer, will be to time the user whilst they are performing the exercise, so that they know when to start the exercise, when they should be halfway through, and when they need to be finished, to try and match the model exercise. This is to meet my third requirement

3.9 GUI

To design the GUI, I drew a sketch of how everything would be positioned. The GUI is the forefront of the direction interactions with the user, and would have to be planned out to be 'easy to use', to adhere to my second non-functional requirement. Since the resolution of the colour camera on the Kinect is 640 x 480, I chose to fit it in a window that has those dimensions, and the total window size to be 1280 x 720. I have displayed a series of buttons on the side, so that the user can select a

particular exercise to perform, adhering to my second functional requirement. A video will also start playing after an exercise has been selected for my fourth requirement. A motivational message will show, once the exercise is over once again, in a text box, this is to meet my eighth requirement.



3.10 Summary

Over the design chapter of this project, I provided a high level overview of how my system should function, and from this overview, broke down each part into how I have designed it to work, and related them to my requirements. I mainly concentrated on designing the Comparison Algorithm, which is the foundation for the whole application, and provided pseudo code to show how it should work. I also briefly looked at my design for the GUI, and the timer designs for the exercise. In the next chapter I will look into bringing my designs to life, via the implementation.

4 Implementation

Implementation is the execution of my designs, and in this chapter I will describe in detail the methods I used to implement my designs from part 3. Then show how I have integrated all the implemented parts to form a final system, which should meet my requirements and resemble the high level design I created in the last chapter. The first part to be implemented is the Comparing Algorithm, split into three sections. Also I will show the tools and technologies I used during the implementation phase.

4.1 Tools and Technologies Used

Kinect SDK v1.8

To fully utilise the Kinect for motion tracking, I needed to use the Kinect for Windows SDK provided with the hardware. The SDK includes many tools and APIs to aid my development, and since I would be working on Windows 7 OS, I would need the SDK to develop my application. The main feature that I needed was access to Skeletal Tracking which the SDK provided APIs for (mainly via the Microsoft.Kinect namespace). I also needed access to the Depth and Colour data streams to produce a video feed on screen whilst the Kinect application was running.

Developer Toolkit Browser v1.8.0

Microsoft have provided a toolkit alongside the SDK, containing examples of Kinect applications. This provides "resources to simplify developing Kinect for Windows applications." [**6**] I studied these and specifically used the 'Skeleton Basics' example to get a feel for how to implement the Skeletal Tracking feature of the Kinect.

Visual C#

Visual C# [**20**] is an object-oriented programming language from Microsoft aimed primarily at developers creating applications using the .NET framework.

I chose to use C# for the implementation of this project because:

- I will be making an application for Windows 7 and the C# .NET framework can be used to assist the development of my application.
- Since C# is an object-oriented language, abstraction can be used to help me represent parts of my high level design.
- The Kinect SDK has many code examples in C# which I can use to assist me in implementing my designs.

Visual Studio 2013

I used the IDE, Visual Studio 2013 [**20**] to design and code the WPF application in C#, since VS is one of the best programs to work on .NET applications. It contains the functionality needed to create a C# project running on Windows 7, which is what I needed to complete the project.

Windows Presentation Foundation (WPF)

I used WPF to display my game to users since you can use a markup language (XAML) to implement an application's appearance. WPF is included in the .NET Framework, so this way I could include other elements from .NET Framework libraries. [**7**] I figured this was a useful way to show timers in which a user would perform an action, and also text to provide feedback to the user all through the UI. Visual Studio 2013 also features the 'WPF Designer' which is a useful tool to use to implement my UI design. Implementing the functionality of responding to the users actions captured by the Kinect, will be done in a type of code known as 'code-behind'. This will be done in C# as stated earlier.

Windows Media Maker

To edit the videos I created of myself performing the exercises to act as a reference for the user, I used the Windows Media Maker just to edit the clips, and add captions to the video, giving some instructions on the timing of the exercises.

4.2 Tracking Skeletons with Kinect

To accomplish my objectives of "*Build an application for Kinect that can record a body movement twice, and compare each action"*, first of all I needed to utilise the Kinect's skeleton tracking to gain joint positions in 3D space. I used the Kinect Toolkit and 'Channel 9' Kinect web series shown in my research chapter to help me do this.

First of all I referenced the WpfViewers project from the Kinect Toolbox that would allow me to use certain WPF tools for my application. The first of these tools is the 'KinectSensorChooser'. This tool will listen for the event that the Kinect has just been turned on, or changed in some way, for example if a user were to unplug the Kinect from the computer. In each case the Kinect will wait for it to be useable, stop any old instances of the Kinect, and then initialise the new instance of the Kinect. In these Events I also enabled the certain streams that I needed to be able to display a colour image of the user, and track a skeleton: the depth stream, the colour stream, and the skeleton stream. The 'KinectColorViewer' tool was also used to setup the RGB camera stream.

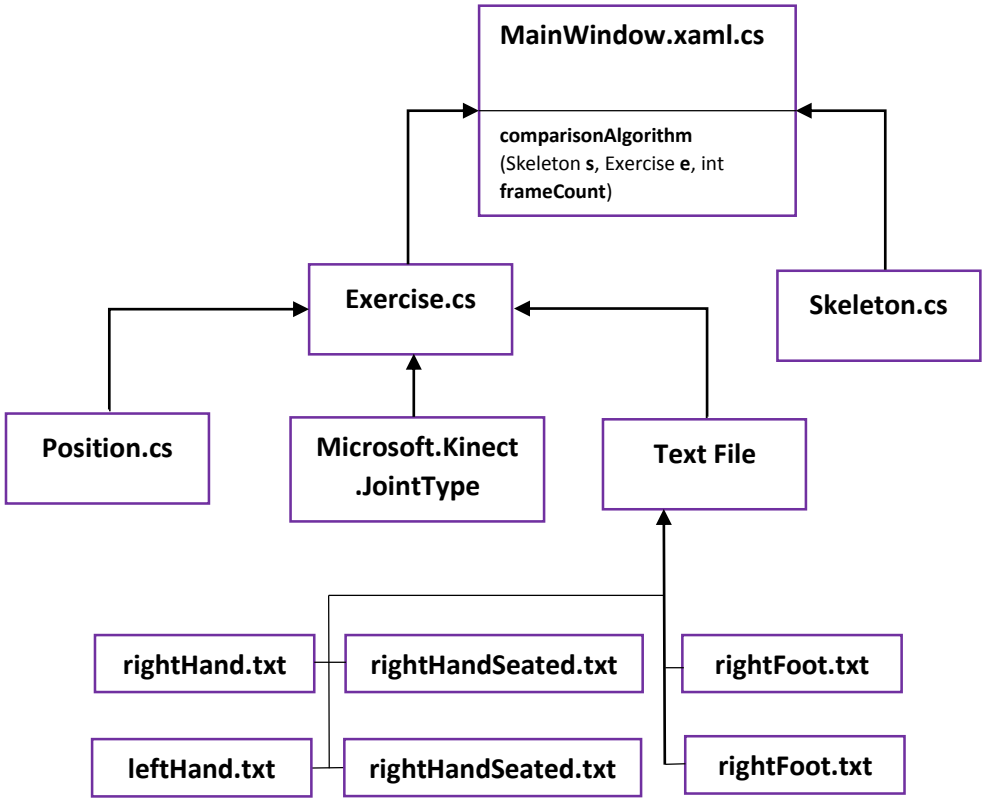
The tutorials then showed me how to get the first skeleton tracked by the Kinect, from its array of skeletons, and then I used the depth camera to map a skeleton point, to a depth point, to a point on the colour image, which then allowed me to use the WPF Ellipse shape to track a joint, every frame.

The part where I would run my Comparison Algorithm is in the AllFramesReady event, which waits until every frame is ready from each data stream, and then decides what to do with the skeleton data, if a skeleton is tracked.

4.3 Comparison Algorithm

My design for the Comparison Algorithm revolves around comparing two sets of Kinect joint positions given to the algorithm, and eventually outputting a score to the GUI by displaying text. The diagram here shows the classes `Position`, and `Exercise`, which I will develop, the enumeration `JointType` from the `Microsoft.Kinect` namespace, which gives access to the different types of joints the Kinect can track, and the `Skeleton` class also from the `Microsoft.Kinect` namespace. All these elements come together to provide the

`comparisonAlgorithm` method in the code-behind contained in `MainWindow.xaml.cs`, the input variables needed to compare two positions.



I developed two classes for my Comparison Algorithm implementation, Position, and Exercise, and I will give a brief description of the two classes here.

Position

To represent a position, I implemented the `Position` class. This class is based on a 3D Vector class which I learnt how to implement in modules during my third year at University. The class can hold an x, y and z position and has several of important methods involving distances and differences.

Exercise

To represent an exercise, I implemented the `Exercise` class. This class holds four positions to represent: the current joint position tracked; the previous joint position tracked; and the exercise's model position all in a single frame, and then also the starting position the model exercise was recorded in. It contains doubles to represent the x and y accuracy percentage values for that exercise, a string pointing to the text file where the model positions are kept, and a list to store the model Positions in once they have been converted from strings to positions. A `JointType` that is initialised in the constructor, depending on which Exercise will be performed is also included. For example, if the exercise involves the right arm, the `JointType` of the exercise will be `HandRight`. Finally the Exercise class contains a Boolean to state whether the exercise is seated or not.

In the design phase, my comparing algorithm was divided into three parts:

- **Position Differences**
- **Comparison Review**
- **Translating Positions**

Here I will pick out my classes and their methods that are directly involved with each three of the different parts of my algorithm, and then go through some vital methods from `Position`, `Exercise` and `MainWindow.xaml.cs` for the respective part.

4.31 Position Differences

Position absDifferenceBetweenPositions

```
public Position absDifferenceBetweenPositions(Position p2)
        {
            Position diff = new Position(Math.Abs(this.getX() - p2.getX()),
            Math.Abs(this.getY() - p2.getY()), Math.Abs(this.getZ() - p2.getZ()));
            return diff;
        }
```

This method will give the absolute difference between two positions in the x, y and z axis, and return the differences in a Position container. This method is essential for the first part of my Comparison Algorithm: **Position Differences**.

4.32 Comparison Review

void percentageComparedBox

```
private void percentageComparedBox(Position modelPosition, Position currentPosition)
        {
            Position difference = modelPosition.absDifferenceBetweenPositions(currentPosition);

            double percentageBoxLength = currentExercise.getLeniency();
            double dX = difference.getX();
            double dY = difference.getY();
            double percentageX;
            double percentageY;


            if (dX >= percentageBoxLength / 2 || dY >= percentageBoxLength / 2)
            {
                currentExercise.addToTotalPercentageX(1);
                currentExercise.addToTotalPercentageY(1);
            }

            else
            {
                percentageX = (((percentageBoxLength / 2) - dX) / (percentageBoxLength / 2)) * 100;
                percentageY = (((percentageBoxLength / 2) - dY) / (percentageBoxLength / 2)) * 100;

                currentExercise.addToTotalPercentageX(percentageX);
                currentExercise.addToTotalPercentageY(percentageY);
            }

        }
```

After the differences between the model position and the current position have been found, this method uses that difference and the comparison review method I designed in chapter 3, to add to the current exercise's total percentage. This method also uses a variable from `Exercise` called leniency, which can change the dimensions of the box around the current position to change the percentage values it will return. For example, because I thought the leg exercises would be harder to perform than the arm exercises, I made the leniency value higher for the legs, to make it slightly easier to get a better score.

A method called `score(Exercise e)` will then produce a score for the current exercise being performed, by finding the average of the total percentage in x and y, and then dividing by the 300 total frames recorded. This percentage is then turned into a rating out of 10, by dividing by 10 and rounding to the nearest whole number, and displayed through to the user through a text block via the GUI. Then depending on the percentage, a motivational message will be output.

Percentage Accuracy	Motivational Message	Reasoning
`if (averageTotalPercent > 80)`	`"EXCELLENT!"`	Over 80% shows that the user has performed the exercise really well, and so receives the best praise.
`else if (averageTotalPercent > 50 & averageTotalPercent < 80)`	`"GREAT!"`	In between 50% and 80% shows they have performed the exercise quite well, but still needs some improvement.
`else if (averageTotalPercent < 50)`	`"GOOD, BUT TRY AGAIN! YOU CAN DO IT!"`	Below 50% shows the user has tried, but has not been able to replicate the model exercise to a good degree, but since the message is supposed to motivate the user it says good to show the user has tried. Also encourages to have another go.
`else if (totalPercentY - totalPercentX > 30)`	`"GOOD, TRY STRETCHING TO THE SIDE MORE!"`	If the user's percentage on the y axis was more than 30% greater than the percentage on the x axis, then the message given will be to inform the user they should try to increase the accuracy on the x axis.
`else if (totalPercentX - totalPercentY > 30)`	`"GOOD, TRY STRETCHING HIGHER!"`	If the user's percentage on the x axis was more than 30% greater than the percentage on the y axis, then the message given will be to inform the user they should try to increase the accuracy on the y axis.

4.33 Translating Positions

A counterpart to the method `absDifferenceBetweenPositions` shown in the Position Differences part is `differenceBetweenPositions`, which finds the difference between two positions without the absolute values. The reason for this is so that it can be used in `positionScale`.

Position positionScale

```
private Position positionScale(Position currentPos)
        {
          return new
          Position(currentExercise.getStartingPosition().differenceBetweenPositions(currentPos));

        }
```

This will find the difference between an exercise's current hip position being tracked, and an exercise's model starting position, tracked on the hip whilst the model exercise positions are being recorded. The model positions are then translated by this difference, once all parts of the algorithm are collaborated. A problem that occurred, whilst working on translating positions, is that I thought that the hip could be tracked whilst the Kinect is in seated mode, however it cannot, meaning I would have to change the joint to compare starting positions with to the head joint, exclusively for the seated versions of my exercise.

4.34 Collaboration

void comparisonAlgorithm

```
void comparisonAlgorithm(Skeleton s, Exercise e, int frameCount)
        {
            exerciseTimer.Start();
            setJointPositions(s, e, 0.1);

            if (frameCount < 300)
            {
                e.setModelPosition(frameCount);
                e.getModelPosition().setX(e.getModelPosition().getX() - scale.getX());
                e.getModelPosition().setY(e.getModelPosition().getY() - scale.getY());

            }

            compare(e);

            frameCount++;

            if (frameCount == 300)
            {
                score(e);
            }
        }
```

This is the final collaboration of the different parts of my Comparison Algorithm. First of all, a timer is started to time the user whilst they are performing the exercise and then the `setJointPositions` method is used to gather the current position that the Kinect is tracking. Whilst the frame counter is below the max amount of frames for an exercise, the model position is translated using the global `Position` scale which receives the differences from the `positionScale` method shown earlier. This difference is then subtracted from the current model position x and y values. After the positions have been compared using `compare`, the frame counter is incremented, and then finally if the exercise has compared every available frame, the score will be produced, using the average percentage values.

4.4 Correcting Inaccuracies

My design for correcting Kinect inaccuracies is shown here, using my method of only using the previous Position's x or y values if the current position is not within a particular offset from the previous position.

bool withinPosition

```
public bool withinPositionX(Position p, double difference)
        {
            if (this.absDifferenceBetweenPositions(p).getX() > difference)
            {
                return false;
            }

            else return true;
        }
```

This method will return true if a position is within a determined distance from another position on the x axis and is used in the next method `setJointPositions`. There are two other methods: `withinPositionY`, and `withinPositionZ`. These two methods work for the y and z axis. The reason I separated the axis into three different methods, rather than returning a whole position, is that `withinPositionZ` is only used to determine if the user is an acceptable distance away from the Kinect before the exercise has begun, otherwise all other uses of the method just involve the x and y axis which happens every frame.

void setJointPositions

```
private void setJointPositions(Skeleton s, Exercise e, double offset)
        {
            Joint joint = s.Joints[e.getJointType()];
            Position jointPos = new Position(joint.Position.X, joint.Position.Y, joint.Position.Z);

            e.setCurrentPosition(joint.Position.X, joint.Position.Y, joint.Position.Z);

            if (!e.getCurrentPosition().withinPositionX(e.getPreviousPosition(), offset))
            {
              e.setCurrentPosition(e.getPreviousPosition().getX(), e.getCurrentPosition().getY(),
              e.getCurrentPosition().getZ());
            }

            if (!e.getCurrentPosition().withinPositionY(e.getPreviousPosition(), offset))
            {
              e.setCurrentPosition(e.getCurrentPosition().getX(), e.getPreviousPosition().getY(),
              e.getCurrentPosition().getZ());
            }

        }
```

The `setJointPositions` method gets a joint's position from the currently tracked `Skeleton` and a given `JointType`, and sets it to be the current position. Then proceeds to follow through with my design.

A problem I had with using previous positions to correct inaccuracies was that to begin with, a previous position would start off with coordinates (0, 0, 0), and then once the Kinect started tracking the relevant joints, my conditions would always be true, since the difference just drastically increased, and consequently would return the previous position, which is initially (0, 0, 0). To overcome this, I implemented a setJointPositions method without an offset value to be used before the Comparison Algorithm. This let the current and previous positions be set to the correct values

before they were used in the Comparison Algorithm, and before the Kinect inaccuracies were corrected.

4.5 Building Model Positions

To build my model positions, I first needed a way of inputting the specific joint positions into a text file, this was done by using a modified version of the `setJointPositions` method used for correcting inaccuracies, but instead of having an exercise, as a parameter, it used two global Positions for that particular exercise, for `example`: `rightHandCurrentPosition`, `rightHandPreviousPosition`. I then used the method `AppenedAllText` from the `System.IO` namespace to record the x, y and z values of the current position to a text file, and add a new line. Initially, timestamps were also included, in collaboration with a timer so that I could time modelling the exercises correctly. These timestamps were then removed, so I could use the raw x, y and z values for the model positions.

```
0.296552389860153, -0.155632108449936, 1.63284909725189
Time: 2015-04-21 22:46:02.092

0.295947521924973, -0.152676820755005, 1.62890982627869
Time: 2015-04-21 22:46:02.125

0.294721186161041, -0.150726154446602, 1.62533009052277
Time: 2015-04-21 22:46:02.159
```

This produced a text file similar to the one above, with 300 lines of positions. Then, using the `stringSplitToPosition` and `fillPositionList` methods I developed in `Exercise`, converted each line of x, y and z values into a position, and then filled a list of positions with the data. This list would be the data structure to hold each model position for each frame across the exercise.

4.6 Handling Seated Exercises

To allow my application to include seated exercises, I needed to take advantage of the Seated Tracking Mode that the Kinect offered. To do this I gave the `Exercise` class a Boolean value to check if the exercise was seated or not. Using this value, I put a condition to check if the exercise was seated, and if it was, turned on the seated tracking mode for the Kinect.

```
if (currentExercise.getSeated() == true)
            {
              sensor.SkeletonStream.TrackingMode = SkeletonTrackingMode.Seated;
            }
```

4.7 Calibration

Whilst implementing my application, I realised that I should make the user stand within a particular offset of the starting position's z value, so that they are a similar distance away from the Kinect to where the model exercises were recorded. To do this I created a Boolean called `calibrate`, that

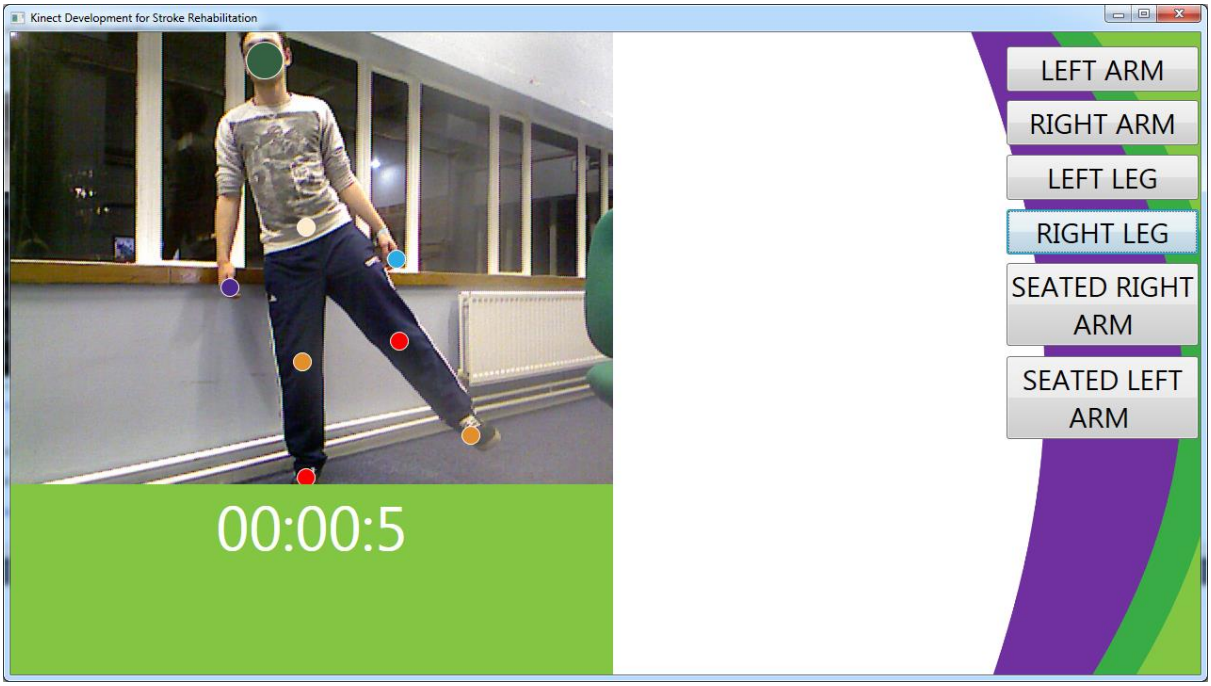
would only become true, if the user was within this z value. Once `calibrate` was true, a timer would count down until the exercise would start, and then the Comparison Algorithm would begin after that. If `calibrate` was not true, then the Kinect would still track the user, but will just wait until they step within the right distance away from the Kinect. This calibration technique relies on the user staying still after they have initially been calibrated.

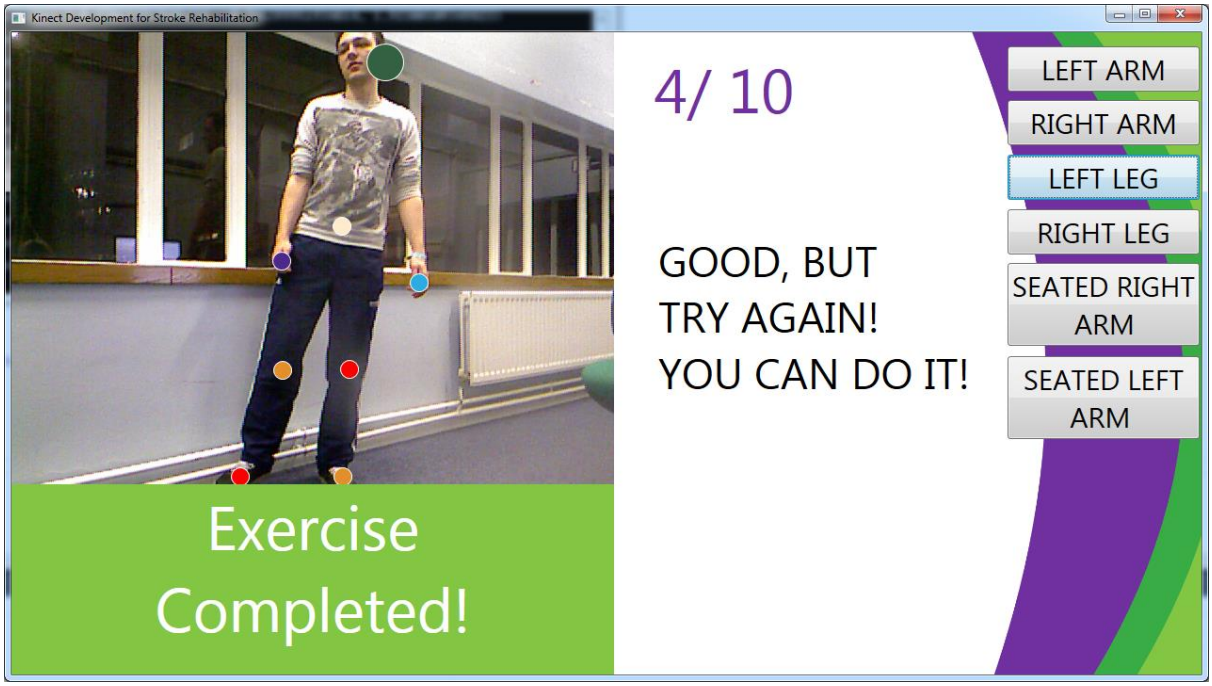
4.8 Timers

To implement each of the timers I designed in the last chapter, I used the WPF `DispatchTimer` Class. This class allowed me to do something in the event that the timers reached a certain time.

Whilst recording the model positions, all `exerciseTimer` did is count up from zero, and I would start recording the exercise at the ten second mark, then the timer would keep going until I stopped the program. For the application, I further developed this timer so that it would start once the Comparison Algorithm started, so that the user could time their exercise correctly. The second timer I implemented, `countDownTimer`, was made to start as soon as the user was calibrated. Then the Comparison Algorithm only starts once the countdown time is equal to zero.

4.9 GUI





Here are some pictures of the final UI I developed for my application. It was based on my UI design in the last chapter.

All text shown has been implemented using the WPF control TextBlock in XAML. This block of text allows me to specify the text I want to display by updating the code-behind depending on what score the user gets, as well as letting me change the foreground and background colours; for example, the background colour of the timer changes if it is counting down. I also had to use some

String formatting to allow the text to be shown on different lines. The multicolour ellipses shown are for tracking the user joints, as described in my tracking skeletons section in the last chapter.

All buttons shown were implemented using the WPF control Button, which provided a simple way to allow the user to change which type of exercise they wanted to perform. In the event of a button press, the exercise is changed to the specified exercise, as well as resetting all timers, TextBlocks, counters, and the `calibrate` Boolean. This is so that you can start another exercise without having to close the application.

Another feature of the GUI that I implemented is the use of the WPF control MediaElement. This let me display an instructional video of how to perform an exercise to the user within the app. I also added a feature to the button click so that the relevant exercise video would appear once clicked, and then once the user was tracked by the Kinect, the video would disappear.

4.10 Summary

The implementation of my Comparison Algorithm design has allowed me to compare my recorded model exercises, with an exercise performed by a user in front of the Kinect in order to meet my second objective to '*build an application for Kinect that can record a body movement twice, and compare each action'.* Then, using the GUI to provide a score to the user, and motivational messages with WPF, has allowed to meet my third objective to *'create a fun game to differ from regular physiotherapy and provide positive feedback for motivational purposes'*. The rest of my aims will be met through the evaluation of my project, and further testing.

5 Testing & Evaluation

5.1 Testing

I split my testing into three phases, unit testing, system testing and exercise testing.

5.11 Unit Testing

For unit testing, I used the white-box method of testing, where I would inspect the inner code of my project to check whether all methods were working correctly. To do this, I enabled the console in my project, and used print statements to check whether the values were in a expected range.

I tested the Comparison Algorithm method, and the methods that are used within it, since this is the bulk of my project. A method was created for the `Position` class, called `printPosition`. This method would print the x, y, and z coordinates of the position to the console and was used to see whether the positions were in the range that I expected to see.

`setJointPositions`

- Tested whether the current position was copied correctly from the Kinect's joint position, and if the previous position was used correctly to correct any inaccuracies that the Kinect may provide.

Translating Positions

- Checked whether the values for the position were the correct translated positions after subtracted the x and y differences.

`percentageComparedBox`

- Printed each percentage every time a position was compared, to check whether the percentages were around the values expected for the action I performed.

Each of the tests passed successfully, apart from one. I discovered that the value for the offset parameter as part of `setJointPositions`, was too low, and so would correct an inaccuracy by making the current position have the previous position. However, in the next frame, since the offset was too low, the current position would still not be in the right range of previous position, and would therefore just use the same previous position for the rest of the exercise. To correct this, I had to make the offset parameter larger, so the method could only correct large inaccuracies provided by the Kinect. Once this was fixed, I was confident to move on to the next testing phase.

5.12 Application Testing

The application testing differed to my unit testing, since I tested the application with the black-box testing method, as if I knew nothing about the inner workings of the code. The main way of accomplishing this, was by testing the GUI, and checking that the application performed with the expected results.

Buttons

- Test whether clicking a button changes the exercise, and starts playing the correct video.
- Test whether the button resets the timers, motivational message, and the score on screen.

Skeleton Tracking

- Test whether if in front of the Kinect's Field of View range, the ellipses will follow the tracked joints on screen.
- Test whether only the seated tracking mode ellipses are tracked, if a seated exercise has been chosen.

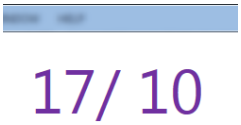
Timers

- Test whether the countdown timer will count down from 10, and start the exercise once it hits 0.
- Test whether once the exercise has started, a timer will start, counting up in seconds from 0 to 10.

Messages

- Test whether a score will be given out of 10, once the exercise has finished.
- Test whether a motivational message appears for each score given
    - 8/10 and upwards should give the message 'EXCELLENT!
    - 8/10 to 5/10 should give the message 'GREAT! TRY AGAIN!'
    - Below 5/10 should give the message 'OK, BUT TRY AGAIN! YOU CAN DO IT!'
    - If the user has performed better in the Y axis by 30% or over, the message should receive 'GOOD, TRY STRETCHING TO THE SIDE MORE!'
    - If the user has performed better in the X axis by 30% or over, the message should receive 'GOOD, TRY STRETCHING HIGHER!'
- Test whether the message 'Exercise Complete!' is shown, once the exercise timer has hit 10 seconds.

All tests were successful apart from the second button test. The criteria was for a button press to reset timers, motivational text blocks, and scores. However for both of the seated exercise buttons, the score would not reset, and would display an impossible score as shown below. This is because I had not reset the percentage totals in either of the seated button click event handlers. I added the percentage total reset method as shown below, and this fixed the problem. Now every button will reset the score.



17/ 10

```
currentExercise.resetPercentageTotals();
```

Now I have tested that my application works correctly, I can move on to test each exercise.

5.13 Exercise Testing

As well as testing my application worked correctly, I came up with a plan to perform each different exercise in different ways, and will later evaluate the results. To gather the results, I created a method that saved the final percentage values of an exercise, and wrote the results to a text file. From here, I could access the results to evaluate them later. The results will be shown to three decimal places for easier viewing, although they were recorded to thirteen decimal places.

Exercise Test Plan

The evaluation plan was split into three different parts: *Full Performance, Half Performance,* and *No Performance.*

Full Performance

- Perform the exercise 20 times with the intent of getting as high of a score as possible. The exercise will be performed from start to finish with a full extension of the limb, to try and copy the model exercise as well as possible, emulating a user with a full range of motion in their limbs.

Half Performance

- Perform the exercise 20 times, with the intent of completing half of the action required by the exercise. This is to emulate someone who has a smaller range of motion in their limbs, to test how the application deals with exercises that are not fully correct.

No Performance

- During the exercise, start in the starting position, and stand still throughout the exercise, 20 times. This will test how the application deals with no movement during the exercise.

Exercise Test Results

Full Performance

83.087	**84.508**	**81.159**	**82.143**	**82.733**	**83.066**

Average Percentages

Half Performance

53.462	**46.890**	**41.808**	**48.737**	**54.582**	**52.689**

Average Percentages

No Performance

36.077	**26.135**	**29.452**	**35.342**	**33.175**	**33.009**

Average Percentages

Now that the exercise testing has been completed, I can move on to evaluation.

5.2 Evaluation

The evaluation of my application is important to this project, because it will figure out if I have built the right system to fit my requirements. I decided to evaluate my application on the consistency of the scores that it gives to the users using my exercise tests, as well as evaluating each individual requirement to see if it has been met. I then got fellow third year students to test the application, and provide some thoughts on the application in general.

5.21 Exercise Testing Results

Using the results obtained from my exercise tests, I will discuss the usability for my application in a stroke rehabilitation setting.

Full Performance

For the full performance of my exercises, the averages of each exercise were all above 80%, meaning on average a user with a full range of motion will receive a score of 8/10 for an exercise that attempts to match the whole action over 10 seconds. The results obtained for a full performance are positive, because it shows that if the user performs the exercise well enough, it will give a good score, paired with the motivational messages 'Great' or 'Excellent' to make the user feel like they've done well in the exercise.

Half Performance

For the half performance exercises, the averages of each exercise ranged from approximately 42% to 55%. The range is about 13% for the averages of the exercises, and I think this is because whilst performing the exercise tests with half the action, I couldn't stay that consistent during testing, because my limbs weren't going to be in the exact same places each time. The hip abduction exercises especially have a higher percentage than most of the other exercises, and this could be because the leniency variables for the leg exercises I designed are higher than the arm exercises. I also think these results are quite positive because the scores shown to the user will be either 4/10 to 5/10, which is what the user should receive for a half performance.

No Performance

For the exercises where I did not move at all, the average for each exercise ranged from approximately 26% to 36%. I think these results are positive results too, since even though the user is doing nothing during the exercise, if they are in the starting position, during some frames, the starting position will be the same as, or close to the model positions at that point. The score of 2/10 or 3/10 is a good score to get for doing no performance, because even though it is a bad score for the exercise, it will never give you 0/10 which is demoralising for a patient in rehab, and the along with the score, the motivational message will show which is supposed to encourage the patient.

5.22 Fellow Student Input

I asked several students to help test my application, and get their thoughts and opinions on the GUI, the game, and idea in general. All students said that the GUI was simple to use, and that the video and timer was helpful to guide them through the exercises. Most found the leg exercises the most tiring and often provided the worst scores, however once the users supported their weight, as shown in the instructional video, the scores did improve. An issue that did appear, is that a lot of students felt a bit underwhelmed by the game aspect of the application, and wished there were more objectives than simply copying an exercise shown on screen. Most did say that using the Kinect

and seeing there joints tracked on screen was fun, and would more likely get them to perform an exercise over trying to do it on their own at home. Although they reiterated once again that another objective to the game would have provided more fun and longevity to the game.

5.23 Project Requirements

The following section lists, discusses and evaluates whether the requirements of the application have been met.

Functional Requirements

Requirement 1: *The application will offer a range of exercises to perform, including sat down exercises.*

The application provides six exercises to perform, four standing with the user's arms and legs, and two that can be performed sat down. This requirement has been met.

Requirement 2: *The user will be able to choose a specific exercise to perform.*

Buttons are provided for the user to be able to choose one of the exercises, or change the exercise before they are tracked by the Kinect. This requirement has been met.

Requirement 3: *The user will be able to see parts of their skeleton tracked on screen.*

A stream from the colour camera of the Kinect is provided for the user to see themselves on screen, along with WPF ellipse shapes following the joints tracked by the Kinect on screen whilst the user is moving. This requirement has been met.

Requirement 4: *The user will be able to see a video, giving instructions on how to perform the exercise.*

Once a button is clicked, a video appears of myself performing the specific model exercise, along with text across the video explaining the timings of the exercise. This requirement has been met.

Requirement 5: *The exercise will not start until the user is properly tracked by the Kinect.*

Once the user has clicked a button to choose an exercise, the video that appears will play until the user is standing the right distance away from the Kinect, and then the countdown timer will start. This requirement has been met.

Requirement 6: *The application will show a countdown timer before the exercise, and a timer during the exercise.*

As in requirement 5, once a user has been tracked, a countdown timer will appear below the camera stream, and start counting down from 10. Once the timer hits 0, the countdown timer disappears, and another timer will appear to time the user whilst they are performing the exercise. This requirement has been met.

Requirement 7: *The user will receive a score to determine how well they performed the chosen exercise.*

Once an exercise has been completed, a score is shown, derived from the percentage accuracy of the exercise the user has performed. This requirement has been met.

Requirement 8: *The user will receive motivational feedback in the form of text depending on their score.*

Further to requirement 7, after an exercise has been completed, a message on the right of the GUI will provide some motivational feedback dependant on the score the user has received. Specific advice will also be given if the user needs to reach higher than they are, or needs to reach further to the side. This requirement has been met.

Requirement 9: *The code will be documented.*

I have used comments within the code whilst implementing my designs, to document it.

Non-Functional Requirements

Requirement 1: *The application will be fun to use.*

This requirement has not quite been met, even though it is subjective, and will be further discussed in my conclusion.

Requirement 2: *The UI will be easy to use.*

The usage of simple buttons on the right side have made my UI easy to use. This is because all the user has to do to start an exercise is click a button. A video will automatically play and the exercise will start once tracked. Then after the exercise has finished, all the user has to do is click one of the buttons again, to perform another exercise. This will clear the UI of any scores, of messages that have previously been displayed.

5.3 Summary

Using unit testing, I managed to test whether the implementation of my designs were working as intended, through white-box testing, and discovered that all my tests passed, apart from one. After fixing the failure from the first test, I used black-box testing to test my application and GUI, which also passed all its tests except for one, which I fixed immediately after finding. The third set of tests undertaken was for the exercises, and involved testing each exercise 20 times, under different conditions. I then gathered the results from these tests, and used them to evaluate my requirements accordingly.

6 Conclusion

6.1 Overview

The aim of my project was to create a game to aid stroke rehabilitation using the Microsoft Kinect. A group of objectives and requirements were presented as the basis of my project. Background research was undertaken to gain more information about the movement issues that affect stroke survivors may face, as well as review some motion tracking devices, and the Kinect in a stroke rehabilitation situation. Next, the designs were laid out for my application, including the Comparison Algorithm made up of three sections: Position Differences, Comparison Review, and Translating Positions. Further design into how I would correct Kinect inaccuracies and record my model exercises was also undertaken. The implementations of these designs were then shown in my fourth chapter, eventually providing the application to compare two exercises and provide a score to the user. Finally, I tested my application in three different ways, and evaluated the requirements provided in my design chapter.

6.2 Aims and Objectives

In my first chapter, I defined an aim and some objectives to lay the foundation to developing this project. Here I will look at each individual objective and assess whether it has been fulfilled.

Objective 1: *Understand how the physical movements of a stroke victim differs to a non-victim.*

During my background research, I discovered the different after-effects of a stroke such as weakness, stiffness and spasticity in a survivor's body parts, as well as the fact that a large amount of patients suffer from falls during the rehabilitation period due to these weaknesses. I also researched the different ranges of motion that are associated with different severities of weakness in a survivor's limbs.

Objective 2: *Build an application for Kinect that can record a body movement twice, and compare each action.*

This part of the application was the most important part of my project, because it was the basis of my game. First of all I needed to be able to access the skeletal tracking feature of the Kinect to gain joint positions of a user. Next I focused on the design and implementation of my Comparison Algorithm that could successfully compare actions that I incorporated into exercises, by returning percentage values.

Objective 3: *Create a fun game to differ from regular physiotherapy and provide positive feedback for motivational purposes.*

I further developed my application from objective 2, identifying the requirements needed for it to be successful. The game developed into watching a colour stream of the user using a camera from the Kinect, and copying a physiotherapeutic exercise developed through research, over a certain amount of time. The game then shows the user a score out of 10, to see how well you copied the exercise, and then show a motivational message, dependant on how well they performed. To say whether the game is fun or not is subjective, however I believe that the involvement of watching yourself on screen whilst the application tracks your joints with ellipses is quite enjoyable in itself. The

involvement of a video to show exactly how to perform the exercise, as well as receiving feedback certainly differs from the way stroke patients would perform their physiotherapy exercises on their own at home. Most students who used my application also agreed that using the Kinect was fun by nature, but argued that more depth in the game would have provided wider opportunity for fun.

Objective 4: *Simulate physiotherapy in a stroke rehabilitation situation using the Kinect sensor and game.*

This objective was hard to meet, due to the ethical issues of using stroke patients with the testing of my application, and the application might have flaws in it that someone who has never suffered from a stroke will not be able to pick up on, however I do believe that I have emulated some kind of rehabilitation situation during the testing of my exercises, since I performed a range of abilities for each exercise.

Objective 5: *Evaluate my findings.*

Finally, my last objective was to evaluate my application and what I discovered during the testing phase. First of all my unit testing showed that the application worked successfully on all except two occasions, which I was able to correct after testing. The requirements for the game were then reviewed to see if the application met them all, which it did. Further testing into performing the exercises was also done, and the game provided the expected range of scores for the performances that I provided. Feedback from students confirmed that the GUI was simple to use, and that they would rather use this application than try to perform exercises on their own at home. It also presented the issue that the game should have more depth in terms of the objectives within the game, and that this could make the game more enjoyable.

6.3 Personal Development

This project has taught me many things, technical, and non-technical. My research skills have developed, and I have learnt how to efficiently research the problem domain. I have also learnt that using deadlines, even on a project that you are working on your own with, is an important part to reaching milestones in a project. On the other side, using C# now means I have knowledge in another programming language, since I had never used it before, as well as developing my programming skills in general. My problem solving skills have also developed, since I learnt how to design algorithms for the problems placed before me. Using the Kinect has also widened my knowledge on motion tracking, and the Kinect SDK with skeletal tracking, and personally, I did enjoy developing for the Kinect.

6.4 Problems Encountered

First of all, the introduction of a new programming language, C#, and the need to learn it was a slight challenge since I had never used it before. The Kinect SDK could be used with either C# or Visual Basic, but because I had been learning C++ throughout the year, I decided to use C# since they were both .NET programming languages, and would have some similarities. Learning specifically how to use events and delegates within C# was a challenge I had to overcome.

Another problem was initially figuring out how to use the skeletal tracking feature of the Kinect with the SDK, and getting some kind of output on the screen for a user. I had to use a vast amount of tutorials, and trial and error to finally get the Kinect tracking to work.

6.5 Thoughts

Here I will discuss some thoughts I have had whilst reflecting on my project.

After evaluating my project, I have realised a design flaw in the basis of my exercises. This is the fact that the score provided is influenced by the timing of a user, as well as the accuracy of how they can replicate an exercise. For example, if I were to replicate an exercise perfectly, position for position, but do it one second too late from the frames in which the model exercise was recorded, I would receive a mediocre score.

An interesting comment made about the application, was the placement of the instructional video. As touched upon very briefly in the research section, stroke survivors can suffer from mental issues, including forgetfulness. My instructional video is placed before the performance of the exercise, to use a guide of what to do, however it disappears once the user is tracked so that they can concentrate on moving their limbs and copying the actions. Because a stroke patient suffer from forgetfulness, the video might be better off being shown as the user performs the exercise, rather than before, or it could be kept on a loop, so that the user can reference the video whilst carrying out the exercise.

6.6 Future Work

6.61 Game Development

Since many people seemed to think that providing more depth in the actual game would make it more fun, a good idea would be to develop the game further. A prime example of this is the game developed within Newcastle University's culture lab, to provide physiotherapy for people suffering from Parkinson's disease [**22**]. This 3D game used the mechanic of grabbing virtual fruit as an avatar, to make the user stretch and perform actions that could help them during physiotherapy. A popular game engine such as Unity, that already has Kinect support, could be used to achieve this goal.

6.62 Correcting Inaccuracies with Direction

Instead of using the previous position of a tracked joint to correct the inaccuracies that can sometimes occur while using the Kinect, the previous direction within a frame could be used instead. This would solve the problem that I faced during testing, where the current position would always be set as the previous position, if the offset value was too low. Using directions also introduces a problem with the basis for my exercises, as when the exercise reaches the half way point, the direction of the action will change, meaning I would have to deal with the changing of directions during the performance of the exercise. I have been informed that research is being put into very subject of correcting Kinect inaccuracies at Newcastle University, and so if this problem is solved, the solution could be incorporated into my design.

6.63 Exercises with Inheritance

Instead of defining what type of exercise an exercise is, within the constructor, I could have used inheritance to abstract each different type of exercise, with the `Exercise` class being the parent of each different type. This design could be implemented in the future.

6.64 Involvement of Stroke Patients

The introduction of my application to actual stroke patients during rehabilitation could provide more insight into how it could be improved, and some of the faults that already are featured within it, that I have not discovered.

References

[**1**] Stroke rehabilitation | Guidance and guidelines | NICE. 2014. Stroke rehabilitation | Guidance and guidelines | NICE. [ONLINE] Available at:https://www.nice.org.uk/guidance/cg162/resources/nice-publishes-guideline-on-rehabilitation-for-people-who-have-had-a-stroke. [Accessed 03 December 2014].

[**2**] Stroke Association's Information Service. | 2013. Physical effects of stroke. | [Online]. | 1. Available at:http://www.stroke.org.uk/sites/default/files/F33_Physical%20effects%20of%20stroke.pdf [Accessed 03 December 2014].

[**3**] Human motion tracking for rehabilitation—A survey. 2014. Human motion tracking for rehabilitation—A survey . [ONLINE] Available at:http://www.sciencedirect.com/science/article/pii/S1746809407000778. [Accessed 08 December 2014].

[**4**] Stroke Rehab Exercises. 2014. Exercises for Stroke Patients. [ONLINE] Available at: http://www.stroke-rehab.com/stroke-rehab-exercises.html. [Accessed 11 December 2014].

[**5**] NIKE+ Kinect Training. 2015. Personalized Nike Training, NIKE+ tracks your every move. [ONLINE] Available at: http://www.nike.com/us/en_us/c/training/nike-plus-kinect-training. [Accessed 21 April 2015].

[**6**]Download Kinect for Windows Developer Toolkit v1.8 from Official Microsoft Download Center. 2015. Download Kinect for Windows Developer Toolkit v1.8 from Official Microsoft Download Center. [ONLINE] Available at:http://www.microsoft.com/en-us/download/details.aspx?id=40276. [Accessed 21 April 2015].

[**7**] Windows Presentation Foundation. 2015. Windows Presentation Foundation. [ONLINE] Available at: https://msdn.microsoft.com/en-us/library/ms754130%28v=vs.110%29.aspx. [Accessed 21 April 2015].

[**8**] Ellipse Class (System.Windows.Shapes). 2015. Ellipse Class (System.Windows.Shapes). [ONLINE] Available at:https://msdn.microsoft.com/en-us/library/system.windows.shapes.ellipse%28v=vs.110%29.aspx. [Accessed 21 April 2015].

[**9**] Gall J, Rosenhahn B, HP Seidel, ., 2010. Drift-free Tracking of Rigid and Articulated Objects. Max-Planck-Institute for Computer Science

[**10**]. 2015. . [ONLINE] Available at:http://www.blogcdn.com/www.engadget.com/media/2012/05/k4w-sensorangle.jpg. [Accessed 25 April 2015].

[**11**] Shotton, Jamie, et al. "Real-time human pose recognition in parts from single depth images."Communications of the ACM 56.1 (2013): 116-124.

[**12**]Home | Stroke Association. 2015. Home | Stroke Association. [ONLINE] Available at: http://www.stroke.org.uk/. [Accessed 25 April 2015].

[**13**] Physical Effects of Stroke | Stroke Association 2015. . [ONLINE] Available at:http://www.stroke.org.uk/sites/default/files/F33_Physical%20effects%20of%20stroke.pdf. [Accessed 25 April 2015].

[**14**] Chang, C, (2012). Towards Pervasive Physical Rehabilitation Using Microsoft Kinect | In 2012 6th International Conference on Pervasive Computing Technologies for Healthcare (PervasiveHealth) and Workshops . University of Southern California, Los Angeles, CA 3Rancho Los Amigos National Rehabilitation Hospital, Downey, CA , : Newcastle University. [Accessed 08 December 2014].

[**15**] Nyberg, Gustafson, L, Y, 1995. Patient Falls in Stroke Rehabilitation. A Challenge to Rehabilitation Strategies.

[**16**]Tracking Modes (Seated and Default). 2015. Tracking Modes (Seated and Default). [ONLINE] Available at: https://msdn.microsoft.com/en-us/library/hh973077.aspx. [Accessed 26 April 2015].

[**17**] Arm Exercises for Stroke Patients. 2014. Arm Exercises for Stroke Patients. [ONLINE] Available at: http://www.stroke-rehab.com/Arm-Exercises.html. [Accessed 11 December 2014].

[**18**] Leg Exercises for Stroke Patients. 2014. Leg Exercises for Stroke Patients. [ONLINE] Available at: http://www.stroke-rehab.com/Leg-Exercises.html. [Accessed 08 December 2014].

[**19**]Kinect for Windows Sensor Components and Specifications. 2015. Kinect for Windows Sensor Components and Specifications. [ONLINE] Available at: https://msdn.microsoft.com/en-us/library/jj131033.aspx. [Accessed 26 April 2015].

[**20**] John Sharp (2013). Developer Step by Step Microsoft C# 2013. .: Octal Publishing. ..

[**21**]Kinect for Windows Quickstart Series | Channel 9. 2015. Kinect for Windows Quickstart Series | Channel 9. [ONLINE] Available at:http://channel9.msdn.com/Series/KinectQuickstart. [Accessed 05 May 2015].

[**22**] Galna B, Jackson D, Schofield G, McNaney R, Webster M, Barry G, Mhiripiri D, Balaam M, Olivier P, Rochester L. 2014. Retraining function in people with Parkinson's disease using the Microsoft kinect: game design and pilot testing. Journal of NeuroEngineering and Rehabilitation, 11 - 60.

[**23**] Cameirão, M. S., Bermúdez, I. B. S., Duarte Oller, E., & Verschure, P. F. (2009). The rehabilitation gaming system: a review. *Stud Health Technol Inform*, *145*(6).

[**24**] Using the Wii Remote | Wii | Support | Nintendo. 2015. Using the Wii Remote | Wii | Support | Nintendo. [ONLINE] Available at:https://www.nintendo.co.uk/Support/Wii/Usage/Wii-Remote/Using-the-Wii-Remote/Using-the-Wii-Remote-243981.html. [Accessed 08 May 2015].

[**25**] Huang, M. C., Chen, E., Xu, W., & Sarrafzadeh, M. (2011, October). Gaming for upper extremities rehabilitation. In Proceedings of the 2nd Conference on Wireless Health (p. 27). ACM.

[**26**] Harley, L., Robertson, S., Gandy, M., Harbert, S., & Britton, D. (2011). The design of an interactive stroke rehabilitation gaming system. In Human-Computer Interaction. Users and Applications (pp. 167-173). Springer Berlin Heidelberg.

[**27**] Q & A - Jintronix. 2015. Jintronix. [ONLINE] Available at: http://www.jintronix.com/faqs/. [Accessed 08 May 2015].

[**28**] Sugarman, H., Weisel-Eichler, A., Burstin, A., & Brown, R. (2011, June). Use of novel virtual reality system for the assessment and treatment of unilateral spatial neglect: A feasibility study. InVirtual Rehabilitation (ICVR), 2011 International Conference on (pp. 1-2). IEEE.

Appendices

Appendix A: WPF Code-behind

```
// (c) Copyright Microsoft Corporation.
// This source is subject to the Microsoft Public License (Ms-PL).
// Please see http://go.microsoft.com/fwlink/?LinkID=131993 for details.
// All other rights reserved.

//Code-behind
//Edited by: Edward Jack Chandler
//Student Number: 120232420

using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using Microsoft.Kinect;
using System.IO;
using System.Reflection;
using Kinect12_02_15;
using System.Windows.Threading;
using Microsoft.Win32;

namespace SkeletalTracking
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        //Instances of Kinect
        KinectSensor old;
        KinectSensor sensor;

        //Timers used
        DispatcherTimer exerciseTimer;
        DispatcherTimer countDownTimer;

        //timer integers
        int time = 0;
        int countDownTime = 10;

        public MainWindow()
        {
            //Initialise timers for WPF
            InitializeComponent();
```

```
            exerciseTimer = new DispatcherTimer();
            exerciseTimer.Interval = TimeSpan.FromSeconds(1);
            exerciseTimer.Tick += exerciseTimerTick;

            countDownTimer = new DispatcherTimer();
            countDownTimer.Interval = TimeSpan.FromSeconds(1);
            countDownTimer.Tick += countDownTick;

        }


        bool closing = false;
        const int skeletonCount = 6;
        Skeleton[] allSkeletons = new Skeleton[skeletonCount];

        bool calibrate = false;
        Position scale;
        Joint joint;

        //current Position for head and hip
        Position hip = new Position();
        Position head = new Position();

        //Every different type of Exercise
        Exercise currentExercise = new Exercise("left arm");
        Exercise leftArm = new Exercise("left arm");
        Exercise rightArm = new Exercise("right arm");
        Exercise leftLeg = new Exercise("left leg");
        Exercise rightLeg = new Exercise("right leg");
        Exercise seatedLeftArm = new Exercise("left arm seated");
        Exercise seatedRightArm = new Exercise("right arm seated");

        //counter for the number of frames
        int frameCounter = 0;
        //when the count down should stop
        int countDownStop = 0;
        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            kinectSensorChooser1.KinectSensorChanged += new
DependencyPropertyChangedEventHandler(kinectSensorChooser1_KinectSensorChanged);
        }

        //Checks if the instance of the Kinect has changed
        void kinectSensorChooser1_KinectSensorChanged(object sender,
DependencyPropertyChangedEventArgs e)
        {
            old = (KinectSensor)e.OldValue;

            StopKinect(old);

            sensor = (KinectSensor)e.NewValue;

            if (sensor == null)
            {
                return;
            }


            var parameters = new TransformSmoothParameters
            {
                Smoothing = 0.3f,
                Correction = 0.0f,
```

```
                Prediction = 0.0f,
                JitterRadius = 1.0f,
                MaxDeviationRadius = 0.5f
            };


            //enable the skeleton stream
            sensor.SkeletonStream.Enable(parameters);

            sensor.SkeletonStream.Enable();

            // subscribe to all frames ready
            sensor.AllFramesReady += new
EventHandler<AllFramesReadyEventArgs>(sensor_AllFramesReady);

            //enable depth and colour stream
            sensor.DepthStream.Enable(DepthImageFormat.Resolution640x480Fps30);
            sensor.ColorStream.Enable(ColorImageFormat.RgbResolution640x480Fps30);

            try
            {
                sensor.Start();
            }
            catch (System.IO.IOException)
            {
                kinectSensorChooser1.AppConflictOccurred();
            }


        }

        //Check whether all frames from each data stream are ready - main loop
        void sensor_AllFramesReady(object sender, AllFramesReadyEventArgs e)
        {
            if (closing)
            {
                return;
            }

            //Get a skeleton
            Skeleton s = GetFirstSkeleton(e);

            if (s == null)
            {
                return;
            }

            //If it is a seated exercise, enable seated tracking mode and make the hip
ellipse disappear
            if (currentExercise.getSeated() == true)
            {
                sensor.SkeletonStream.TrackingMode = SkeletonTrackingMode.Seated;
                hipEllipse.Opacity = 0;
            }

            //else make all other ellipses appear, and turn skeleton tracking mode to
default
            else
            {
                sensor.SkeletonStream.TrackingMode = SkeletonTrackingMode.Default;
                leftKneeEllipse.Opacity = 1;
                rightKneeEllipse.Opacity = 1;
```

```
                hipEllipse.Opacity = 1;
                leftFootEllipse.Opacity = 1;
                rightFootEllipse.Opacity = 1;
            }

            //If a skeleton is tracked
            if (s.TrackingState == SkeletonTrackingState.Tracked)
            {
                //set initial joints for head and hip
                setJointPositions(s, JointType.HipCenter, hip);
                setJointPositions(s, JointType.Head, head);

                //If standing exercise and not calibrated
                if (currentExercise.getSeated() == false & calibrate == false)
                {
                    //Check whether you right distance away to start exerise
                    if (hip.withinPositionZ(currentExercise.getStartingPosition(),
0.05))
                    {
                        calibrate = true;
                        // start countdown timer, and make video disappear
                        countDownTimer.Start();
                        Instructional_Video_Player.Stop();
                        Instructional_Video_Player.Opacity = 0;
                    }
                }

                //else if seated
                else if (currentExercise.getSeated() == true & calibrate == false)
                {
                    //Check whether you right distance away to start exerise
                    if (head.withinPositionZ(currentExercise.getStartingPosition(),
0.05))
                    {
                        calibrate = true;
                        // start countdown timer, and make video disappear
                        countDownTimer.Start();
                        Instructional_Video_Player.Stop();
                        Instructional_Video_Player.Opacity = 0;
                    }
                }

                //If stood right distance away from Kinect
                if (calibrate == true)
                {
                    //and still first frame
                    if (frameCounter == 0)
                    {
                        //if seated exercise
                        if (currentExercise.getSeated() == false)
                        {
                            //Set joints on the hip and work out the distances needed
to translate positions. Hold values in Position scale.
                            setJointPositions(s, JointType.HipCenter, hip);
                            setJointPositions(s, currentExercise.getJointType(),
currentExercise.getCurrentPosition());
                            scale = positionScale(hip);
                        }

                            //if seated
                        else if (currentExercise.getSeated() == true)
                        {
```

```
                            //Set joints on the head and work out the distances needed
to translate positions. Hold values in Position scale.
                            setJointPositions(s, JointType.Head, head);
                            setJointPositions(s, currentExercise.getJointType(),
currentExercise.getCurrentPosition());
                            scale = positionScale(head);
                        }

                        //Set previous position as current position

currentExercise.setPreviousPosition(currentExercise.getCurrentPosition());
                    }

                    //if count down should stop
                    if (countDownTime == countDownStop)
                    {
                        //start comparison algorithm
                        comparisonAlgorithm(s, currentExercise);

                    }

                }
            }
            //get points on camera to set ellipses to
            GetCameraPoint(s, e);

        }


        void comparisonAlgorithm(Skeleton s, Exercise e)
        {
            //stop count down, start exercise timer
            countDownTimer.Stop();
            exerciseTimer.Start();
            //set positions and correct any inaccuracies above the offset  of 0.2
            setJointPositions(s, e, 0.2);

            //if exercise not finished
            if (frameCounter < 300)
            {
                //translate the model positions
                e.setModelPosition(frameCounter);
                e.getModelPosition().setX(e.getModelPosition().getX() - scale.getX());
                e.getModelPosition().setY(e.getModelPosition().getY() - scale.getY());
                //set percentage values for the current frame
                percentageComparedBox(currentExercise);
                //set previous position as current position

currentExercise.setPreviousPosition(currentExercise.getCurrentPosition());
                //next frame
                frameCounter++;
            }


            //if exercise finished
            if (frameCounter == 300)
            {
                //display score
                score(e);
                //increment frame counter by one to finish exercise
                frameCounter++;
            }
```

```
        }

        //Converts joint location and gets the points to display on the colour stream,
then sets the ellipses to track a user
        void GetCameraPoint(Skeleton first, AllFramesReadyEventArgs e)
        {

            using (DepthImageFrame depth = e.OpenDepthImageFrame())
            {
                if (depth == null ||
                    kinectSensorChooser1.Kinect == null)
                {
                    return;
                }


                //Map a joint location to a point on the depth map
                //head
                DepthImagePoint headDepthPoint =
                    depth.MapFromSkeletonPoint(first.Joints[JointType.Head].Position);
                //left hand
                DepthImagePoint leftDepthPoint =

depth.MapFromSkeletonPoint(first.Joints[JointType.HandLeft].Position);
                //right hand
                DepthImagePoint rightDepthPoint =

depth.MapFromSkeletonPoint(first.Joints[JointType.HandRight].Position);
                //left knee
                DepthImagePoint leftKneeDepthPoint =

depth.MapFromSkeletonPoint(first.Joints[JointType.KneeLeft].Position);
                //right knee
                DepthImagePoint rightKneeDepthPoint =

depth.MapFromSkeletonPoint(first.Joints[JointType.KneeRight].Position);
                //hip
                DepthImagePoint hipDepthPoint =

depth.MapFromSkeletonPoint(first.Joints[JointType.HipCenter].Position);
                //left foot
                DepthImagePoint leftFootDepthPoint =

depth.MapFromSkeletonPoint(first.Joints[JointType.FootLeft].Position);
                DepthImagePoint rightFootDepthPoint =

depth.MapFromSkeletonPoint(first.Joints[JointType.FootRight].Position);

                //Map a depth point to a point on the color image
                //head
                ColorImagePoint headColorPoint =
                    depth.MapToColorImagePoint(headDepthPoint.X, headDepthPoint.Y,
                    ColorImageFormat.RgbResolution640x480Fps30);
                //left hand
                ColorImagePoint leftColorPoint =
                    depth.MapToColorImagePoint(leftDepthPoint.X, leftDepthPoint.Y,
                    ColorImageFormat.RgbResolution640x480Fps30);
                //right hand
                ColorImagePoint rightColorPoint =
                    depth.MapToColorImagePoint(rightDepthPoint.X, rightDepthPoint.Y,
                    ColorImageFormat.RgbResolution640x480Fps30);
                //left hand
```

```
                ColorImagePoint leftKneeColorPoint =
                    depth.MapToColorImagePoint(leftKneeDepthPoint.X,
leftKneeDepthPoint.Y,
                    ColorImageFormat.RgbResolution640x480Fps30);
                //right hand
                ColorImagePoint rightKneeColorPoint =
                    depth.MapToColorImagePoint(rightKneeDepthPoint.X,
rightKneeDepthPoint.Y,
                    ColorImageFormat.RgbResolution640x480Fps30);
                //hip
                ColorImagePoint hipColorPoint =
                    depth.MapToColorImagePoint(hipDepthPoint.X, hipDepthPoint.Y,
                    ColorImageFormat.RgbResolution640x480Fps30);
                //left foot
                ColorImagePoint leftFootColorPoint =
                    depth.MapToColorImagePoint(leftFootDepthPoint.X,
leftFootDepthPoint.Y,
                    ColorImageFormat.RgbResolution640x480Fps30);
                //right foot
                ColorImagePoint rightFootColorPoint =
                    depth.MapToColorImagePoint(rightFootDepthPoint.X,
rightFootDepthPoint.Y,
                    ColorImageFormat.RgbResolution640x480Fps30);

                //Set location
                CameraPosition(headImage, headColorPoint);
                CameraPosition(leftEllipse, leftColorPoint);
                CameraPosition(rightEllipse, rightColorPoint);
                CameraPosition(leftKneeEllipse, leftKneeColorPoint);
                CameraPosition(rightKneeEllipse, rightKneeColorPoint);
                CameraPosition(hipEllipse, hipColorPoint);
                CameraPosition(leftFootEllipse, leftFootColorPoint);
                CameraPosition(rightFootEllipse, rightFootColorPoint);

            }
        }

        //Gets the skeleton from the skeleton array (API always returns 6)
        Skeleton GetFirstSkeleton(AllFramesReadyEventArgs e)
        {
            using (SkeletonFrame skeletonFrameData = e.OpenSkeletonFrame())
            {
                if (skeletonFrameData == null)
                {
                    return null;
                }


                skeletonFrameData.CopySkeletonDataTo(allSkeletons);

                //get the first tracked skeleton
                Skeleton first = (from s in allSkeletons
                                  where s.TrackingState ==
SkeletonTrackingState.Tracked
                                  select s).FirstOrDefault();

                return first;

            }
        }

        //Used to stop the Kinect sensor
```

```
        private void StopKinect(KinectSensor sensor)
        {
            if (sensor != null)
            {
                if (sensor.IsRunning)
                {
                    //stop sensor
                    sensor.Stop();

                    //stop audio if not null
                    if (sensor.AudioSource != null)
                    {
                        sensor.AudioSource.Stop();
                    }


                }
            }
        }

        //Used to set the ellipse to a point on the colour camera
        private void CameraPosition(FrameworkElement element, ColorImagePoint point)
        {
            //Divide by 2 for width and height so point is right in the middle
            // instead of in top/left corner
            Canvas.SetLeft(element, point.X - element.Width / 2);
            Canvas.SetTop(element, point.Y - element.Height / 2);

        }

        //if the window closes stop the kinect
        private void Window_Closing(object sender,
System.ComponentModel.CancelEventArgs e)
        {
            closing = true;
            StopKinect(kinectSensorChooser1.Kinect);
        }

        //correcting inaccuracies and sets positions
        private void setJointPositions(Skeleton s, Exercise e, double offset)
        {
            joint = s.Joints[e.getJointType()];
            Position jointPos = new Position(joint.Position.X, joint.Position.Y,
joint.Position.Z);
            //Takes joint tracked by Kinect and makes a copy of the xyz positions
            e.setCurrentPosition(joint.Position.X, joint.Position.Y,
joint.Position.Z);

            //If X is out of position use previous X
            if (!e.getCurrentPosition().withinPositionX(e.getPreviousPosition(),
offset))
            {
                e.setCurrentPosition(e.getPreviousPosition().getX(),
e.getCurrentPosition().getY(), e.getCurrentPosition().getZ());
            }

            //If Y is out of position use previous Y
            if (!e.getCurrentPosition().withinPositionY(e.getPreviousPosition(),
offset))
            {
                e.setCurrentPosition(e.getCurrentPosition().getX(),
e.getPreviousPosition().getY(), e.getCurrentPosition().getZ());
```

```
            }

            e.getCurrentPosition().printPosition();

        }

        //sets joint positions without correcting inaccuracies
        private void setJointPositions(Skeleton s, JointType t, Position currentPos)
        {
            Joint joint = s.Joints[t];
            Position jointPos = new Position(joint.Position.X, joint.Position.Y,
joint.Position.Z);

            currentPos.setPosition(joint.Position.X, joint.Position.Y,
joint.Position.Z);

        }

        //used to set the joint positions whilst I recorded of model exercises,
without the use of an exercise
        private void setJointPositionsRecord(Skeleton s, JointType t, Position
currentPos, Position previousPos, double offset)
        {
            Joint joint = s.Joints[t];
            Position jointPos = new Position(joint.Position.X, joint.Position.Y,
joint.Position.Z);

            currentPos.setPosition(joint.Position.X, joint.Position.Y,
joint.Position.Z);

            //If X is out of position use previous X
            if (!jointPos.withinPositionX(previousPos, offset))
            {
                currentPos.setPosition(previousPos.getX(), currentPos.getY(),
currentPos.getZ());
            }

            //If Y is out of position use previous Y
            if (!jointPos.withinPositionY(previousPos, offset))
            {
                currentPos.setPosition(currentPos.getX(), previousPos.getY(),
currentPos.getZ());
            }

        }

        //save model exercises to a text file
        private void saveValues(String file, Position p)
        {
            File.AppendAllText(file, p.getX() + ", " + p.getY() + ", " + p.getZ() +
Environment.NewLine);
        }

        //save percentages for evaluation
        private void savePercentage(String file, double percentage)
        {
            File.AppendAllText(file, percentage + Environment.NewLine);
        }

        //create the box shown in designs to turn the comparison of two points into a
percentage
        private void percentageComparedBox(Exercise e)
```

```
        {
            Position difference =
e.getModelPosition().absDifferenceBetweenPositions(e.getCurrentPosition());

            double percentageBoxLength = currentExercise.getLeniency();
            double dX = difference.getX();
            double dY = difference.getY();
            double percentageX;
            double percentageY;

            //if the current position not within percentage box, add 1% to the total
            if (dX >= percentageBoxLength / 2 || dY >= percentageBoxLength / 2)
            {
                currentExercise.addToTotalPercentageX(1);
                currentExercise.addToTotalPercentageY(1);
            }

            //else
            else
            {
                percentageX = (((percentageBoxLength / 2) - dX) / (percentageBoxLength
/ 2)) * 100;
                percentageY = (((percentageBoxLength / 2) - dY) / (percentageBoxLength
/ 2)) * 100;

                //add each x and y percentage to the current exercise's totals
                currentExercise.addToTotalPercentageX(percentageX);
                currentExercise.addToTotalPercentageY(percentageY);
            }

        }

        //returns the non absolute differences between two point for use with
translating positions
        private Position positionScale(Position currentPos)
        {
            return new
Position(currentExercise.getStartingPosition().differenceBetweenPositions(currentPos))
;


        }

        //works out the score of a given exercise and outputs to a text block, and
shows a motivational message to the user
        private void score(Exercise e)
        {
            // divide by 300 frames since that is the total for one 10 second exercise
at 30fps
            double percentageX = e.getTotalPercentageX() / 300;
            double percentageY = e.getTotalPercentageY() / 300;

            //average of the X and Y percentages
            double averageTotalPercent = (percentageX + percentageY) / 2;
            //divide percentage by 10 and round up to the nearest whole number to get
score. Then output to 'percent' text block
            percent.Text = Math.Round(averageTotalPercent / 10, 0).ToString() + "/
10";

            //Excellent score
            if (averageTotalPercent > 80)
            {
```

```
                //reset percentages
                averageTotalPercent = 0;
                percentageX = 0;
                percentageY = 0;

                //output motivation to text block
                motivation.Text = "EXCELLENT!";
            }

            //Great score
            else if (averageTotalPercent > 50 & averageTotalPercent < 80)
            {
                averageTotalPercent = 0;
                percentageX = 0;
                percentageY = 0;

                motivation.Text = string.Format("GREAT!,{0}TRY AGAIN!",
Environment.NewLine);
            }

            // < 50
            else if (averageTotalPercent < 50)
            {
                averageTotalPercent = 0;
                percentageX = 0;
                percentageY = 0;

                motivation.Text = string.Format("OK, BUT{0}TRY AGAIN!{0}YOU CAN DO
IT!", Environment.NewLine);
            }

            //exercise better in the y axis
            else if (percentageY - percentageX > 30)
            {
                averageTotalPercent = 0;
                percentageX = 0;
                percentageY = 0;

                motivation.Text = string.Format("GOOD,{0}TRY STRETCHING{0}TO
THE{0}SIDE MORE!", Environment.NewLine);
            }

            //exercise better in the x axis
            else if (percentageX - percentageY > 30)
            {
                averageTotalPercent = 0;
                percentageX = 0;
                percentageY = 0;

                motivation.Text = string.Format("GOOD,{0}TRY STRETCHING{0}HIGHER!",
Environment.NewLine);
            }

        }

        private void kinectColorViewer1_Loaded(object sender, RoutedEventArgs e)
        {

        }

        //Count down event handler
        void countDownTick(object sender, EventArgs e)
```

```
        {
            if (tbTime != null)
            {
                //if countdown still counting down
                if (countDownTime > countDownStop)
                {
                    //subtract 1 second from the time
                    countDownTime--;
                    //Make text background purple and output time to text
                    tbTime.Background = (SolidColorBrush)(new
BrushConverter().ConvertFrom("#563D90"));
                    tbTime.Text = string.Format("00:0{0}:{1}", countDownTime / 60,
countDownTime % 60);
                }

            }
        }

        void exerciseTimerTick(object sender, EventArgs e)
        {
            //whilst counting up
            if (time < 10)
            {
                //add 1 second to the time
                time++;
                //Make text background green and output time to text
                tbTime.Background = (SolidColorBrush)(new
BrushConverter().ConvertFrom("#83C643"));
                tbTime.Text = string.Format("00:0{0}:{1}", time / 60, time % 60);

            }

            //once exercise has finished, output "Exercise Complete!"
            else
            {
                tbTime.Text = string.Format("Exercise{0}Complete!",
Environment.NewLine);
            }

        }

        //left arm event handler
        private void leftArmButtonClick(object sender, RoutedEventArgs e)
        {
            //Reset all UI elements, the percentage totals, the timers, the
calibration boolean, and play the relevant video
            currentExercise.resetPercentageTotals();
            currentExercise = leftArm;
            exerciseTimer.Stop();
            time = 0;
            countDownTime = 10;
            tbTime.Text = "";
            tbTime.Background = new SolidColorBrush(Colors.White);
            motivation.Text = "";
            percent.Text = "";
            calibrate = false;
            frameCounter = 0;
            Instructional_Video_Player.Opacity = 1;
            Instructional_Video_Player.Source = new
Uri(System.IO.Path.Combine(System.IO.Path.GetDirectoryName(Assembly.GetExecutingAssemb
ly().Location), @"Instructional Videos/leftArmMov.mp4"));
            Instructional_Video_Player.Play();
```

```
        }

        //right arm event handler
        private void rightArmButtonClick(object sender, RoutedEventArgs e)
        {
            //Reset all UI elements, the percentage totals, the timers, the
calibration boolean, and play the relevant video
            currentExercise.resetPercentageTotals();
            currentExercise = rightArm;
            exerciseTimer.Stop();
            time = 0;
            countDownTime = 10;
            tbTime.Text = "";
            tbTime.Background = new SolidColorBrush(Colors.White);
            motivation.Text = "";
            percent.Text = "";
            calibrate = false;
            frameCounter = 0;
            Instructional_Video_Player.Opacity = 1;
            Instructional_Video_Player.Source = new
Uri(System.IO.Path.Combine(System.IO.Path.GetDirectoryName(Assembly.GetExecutingAssemb
ly().Location), @"Instructional Videos/rightArmMov.mp4"));
            Instructional_Video_Player.Play();
        }

        //left leg event handler
        private void leftLegButtonClick(object sender, RoutedEventArgs e)
        {
            //Reset all UI elements, the percentage totals, the timers, the
calibration boolean, and play the relevant video
            currentExercise.resetPercentageTotals();
            currentExercise = leftLeg;
            exerciseTimer.Stop();
            time = 0;
            countDownTime = 10;
            tbTime.Text = "";
            tbTime.Background = new SolidColorBrush(Colors.White);
            motivation.Text = "";
            percent.Text = "";
            calibrate = false;
            frameCounter = 0;
            Instructional_Video_Player.Opacity = 1;
            Instructional_Video_Player.Source = new
Uri(System.IO.Path.Combine(System.IO.Path.GetDirectoryName(Assembly.GetExecutingAssemb
ly().Location), @"Instructional Videos/leftLegMov.mp4"));
            Instructional_Video_Player.Play();
        }

        //right leg event handler
        private void rightLegButtonClick(object sender, RoutedEventArgs e)
        {
            //Reset all UI elements, the percentage totals, the timers, the
calibration boolean, and play the relevant video
            currentExercise.resetPercentageTotals();
            currentExercise = rightLeg;
            exerciseTimer.Stop();
            time = 0;
            countDownTime = 10;
            tbTime.Text = "";
            tbTime.Background = new SolidColorBrush(Colors.White);
            motivation.Text = "";
            percent.Text = "";
```

```
            calibrate = false;
            frameCounter = 0;
            Instructional_Video_Player.Opacity = 1;
            Instructional_Video_Player.Source = new
Uri(System.IO.Path.Combine(System.IO.Path.GetDirectoryName(Assembly.GetExecutingAssemb
ly().Location), @"Instructional Videos/rightLegMov.mp4"));
            Instructional_Video_Player.Play();
        }

        //seated right arm event handler
        private void seatedRightArmButtonClick(object sender, RoutedEventArgs e)
        {
            //Reset all UI elements, the percentage totals, the timers, the
calibration boolean, and play the relevant video
            currentExercise.resetPercentageTotals();
            currentExercise = seatedRightArm;
            exerciseTimer.Stop();
            time = 0;
            countDownTime = 10;
            tbTime.Text = "";
            tbTime.Background = new SolidColorBrush(Colors.White);
            motivation.Text = "";
            percent.Text = "";
            calibrate = false;
            frameCounter = 0;
            Instructional_Video_Player.Opacity = 1;
            Instructional_Video_Player.Source = new
Uri(System.IO.Path.Combine(System.IO.Path.GetDirectoryName(Assembly.GetExecutingAssemb
ly().Location), @"Instructional Videos/seatedRightArmMov.mp4"));
            Instructional_Video_Player.Play();
        }

        //seated left arm event handler
        private void seatedLeftArmButtonClick(object sender, RoutedEventArgs e)
        {
            //Reset all UI elements, the percentage totals, the timers, the
calibration boolean, and play the relevant video
            currentExercise.resetPercentageTotals();
            currentExercise = seatedLeftArm;
            exerciseTimer.Stop();
            time = 0;
            countDownTime = 10;
            tbTime.Text = "";
            tbTime.Background = new SolidColorBrush(Colors.White);
            motivation.Text = "";
            percent.Text = "";
            calibrate = false;
            frameCounter = 0;
            Instructional_Video_Player.Opacity = 1;
            Instructional_Video_Player.Source = new
Uri(System.IO.Path.Combine(System.IO.Path.GetDirectoryName(Assembly.GetExecutingAssemb
ly().Location), @"Instructional Videos/seatedLeftArmMov.mp4"));
            Instructional_Video_Player.Play();
        }

        private void kinectSensorChooser1_Loaded(object sender, RoutedEventArgs e)
        {

        }
    }
}
```

Appendix B: Class Implementations

B1: The `Exercise` Class

```
//Exercise Class
//Author: Edward Jack Chandler
//Student Number: 120232420

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO;
using System.Reflection;
using SkeletalTracking;
using Microsoft.Kinect;
using Microsoft.Win32;

namespace Kinect12_02_15
{
    class Exercise
    {
        //Exercies holds the current position, previous position, and model position
for the current frame, and also the starting position of the exercise
        private Position currentPos = new Position();
        private Position previousPos = new Position();
        private Position modelPos = new Position();
        private Position startingPos = new Position();

        //total percentages of the exercise
        private double totalPercentageX = 0;
        private double totalPercentageY = 0;

        //text file location for the model exercises
        private String modelTextFile;

        //List of model positions, and a list of strings,
        private List<Position> posList = new List<Position>();

        //joint type for the  current exercise, initialised as the hip
        JointType t = JointType.HipCenter;
        //boolean to check if the exercise is seated
        bool seated = false;

        //leniency value to alter the dimensions of the percentage box
        private double leniency;

        //Constructor takes the exercise name as a parameter, then depending on the
name, will set the text file location, fill the list of positions with model
positions, set the relevant joint type,
        //set the correct starting position for those model exercises, and set the
leniency
        public Exercise(String exercise)
        {
            if (exercise == "right arm")
            {
                modelTextFile =
Path.Combine(Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location), @"Joint
Samples/rightHand.txt");
                fillPositionList(modelTextFile);
```

```
                t = JointType.HandRight;
                startingPos.setPosition(-0.0926099568605423, 0.324595719575882,
2.42309355735779);
                leniency = 0.75;
            }

            if (exercise == "left arm")
            {
                modelTextFile =
Path.Combine(Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location), @"Joint
Samples/leftHand.txt");
                fillPositionList(modelTextFile);
                t = JointType.HandLeft;
                startingPos.setPosition(0.0216575562953949, 0.329336374998093,
2.34459733963013);
                leniency = 0.75;
            }

            if (exercise == "right arm seated")
            {
                modelTextFile =
Path.Combine(Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location), @"Joint
Samples/seatedRightHand.txt");
                fillPositionList(modelTextFile);
                t = JointType.HandRight;
                startingPos.setPosition(0.0509357675909996, 0.549594581127167,
1.84863603115082);
                seated = true;
                leniency = 0.75;
            }

            if (exercise == "left arm seated")
            {
                modelTextFile =
Path.Combine(Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location), @"Joint
Samples/seatedLeftHand.txt");
                fillPositionList(modelTextFile);
                t = JointType.HandLeft;
                seated = true;
                startingPos.setPosition(0.102827176451683, 0.570048153400421,
2.06467723846436);
                leniency = 0.75;
            }

            if (exercise == "right leg")
            {
                modelTextFile =
Path.Combine(Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location), @"Joint
Samples/rightFoot.txt");
                fillPositionList(modelTextFile);
                t = JointType.FootRight;
                startingPos.setPosition(0.0411089211702347, 0.320578664541245,
2.37570142745972);
                leniency = 0.8;

            }

            if (exercise == "left leg")
            {
                modelTextFile =
Path.Combine(Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location), @"Joint
Samples/leftFoot.txt");
```

```
                fillPositionList(modelTextFile);
                t = JointType.FootLeft;
                startingPos.setPosition(0.0647142082452774, 0.315704494714737,
2.31226325035095);
                leniency = 0.8;

            }

        }

        //GET METHODS
        public Position getCurrentPosition()
        {
            return currentPos;
        }

        public Position getPreviousPosition()
        {
            return previousPos;
        }

        public Position getModelPosition()
        {
            return modelPos;
        }

        public List<Position> getPositionList()
        {
            return posList;
        }

        public double getTotalDiffX()
        {
            return totalPercentageX;
        }

        public double getTotalDiffY()
        {
            return totalPercentageY;
        }

        public JointType getJointType()
        {
            return t;
        }

        public Position getStartingPosition()
        {
            return startingPos;
        }

        public double getLeniency()
        {
            return leniency;
        }

        public double getTotalPercentageX()
        {
            return totalPercentageX;
        }

        public double getTotalPercentageY()
```

```
        {
            return totalPercentageY;
        }

        public bool getSeated()
        {
            return seated;
        }

        //Adds to the total percentageX
        public void addToTotalPercentageX(double percentage)
        {
            totalPercentageX += percentage;
        }
        //Adds to the total percentageY
        public void addToTotalPercentageY(double percentage)
        {
            totalPercentageY += percentage;
        }


        //makes the total percentage x and y equal to zero
        public void resetPercentageTotals()
        {
            totalPercentageX = 0;
            totalPercentageY = 0;

        }

        //SET METHODS
        public void setCurrentPosition(Position p)
        {
            currentPos.setPosition(p);
        }

        public void setCurrentPosition(double x, double y, double z)
        {
            currentPos.setPosition(x, y, z);
        }

        public void setPreviousPosition(Position p)
        {
            previousPos.setPosition(p);
        }

        public void setModelPosition(int counter)
        {
            modelPos.setPosition(posList[counter]);
        }

        //Takes a line of coordinates eg. 0.41251325, 1.523525, 2.523525, split by the
delimiter ", "
        //Then adds each string value to a list of strings. Once in the list of
strings
        //the string values are converted to doubles, and a new position is returned
with
        // the 3 values as x, y and z
        public Position stringSplitToPosition(String line)
        {
            String[] stringSeperator = new String[] { ", " };
            String[] result = line.Split(stringSeperator, StringSplitOptions.None);
            List<String> stringList = new List<String>();
```

```
            foreach (String s in result)
            {
                stringList.Add(s);

            }

            double x = Convert.ToDouble(stringList[0]);
            double y = Convert.ToDouble(stringList[1]);
            double z = Convert.ToDouble(stringList[2]);

            return new Position(x, y, z);
        }

        //Takes in the the model exercise file, and creates a position out of every
line, and
        //then puts every position into a list of positions
        public void fillPositionList(String file)
        {
            String line;
            using (StreamReader sr = new StreamReader(file))
            {
                while ((line = sr.ReadLine()) != null)
                {
                    posList.Add(stringSplitToPosition(line));

                }

            }
        }


    }
}
```

B2: The `Position` Class

```
//Position Class
//Author: Edward Jack Chandler
//Student Number: 120232420

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Kinect12_02_15
{

    class Position
    {
        private double x;
        private double y;
        private double z;

        //Default constructor
        public Position()
        {
            x = 0;
            y = 0;
            z = 0;
        }

        //3D Constructor
        public Position(double xPos, double yPos, double zPos)
        {
            x = xPos;
            y = yPos;
            z = zPos;
        }

        //2D Constructor
        public Position(double xPos, double yPos)
        {
            x = xPos;
            y = yPos;
            z = 0;
        }

        //copy constructor
        public Position(Position p)
        {
            x = p.x;
            y = p.y;
            z = p.z;
        }

        //GET METHODS
        public double getX()
        {
            return x;
        }

        public double getY()
        {
            return y;
```

```
        }


        public double getZ()
        {
            return z;
        }

        //SET VALUES
        public void setX(double x)
        {
            this.x = x;
        }

        public void setY(double y)
        {
            this.y = y;
        }

        public void setZ(double z)
        {
            this.z = z;
        }

        public void setPosition(double x, double y, double z)
        {
            setX(x);
            setY(y);
            setZ(z);
        }

        public void setPosition(Position p)
        {
            setX(p.getX());
            setY(p.getY());
            setZ(p.getZ());
        }

        //Works out the absolute difference between two positions.
        public Position absDifferenceBetweenPositions(Position p2)
        {
            Position diff = new Position(Math.Abs(this.getX() - p2.getX()),
                Math.Abs(this.getY() - p2.getY()), Math.Abs(this.getZ() - p2.getZ()));
            return diff;
        }

        //Works out the difference between two positions without absolute values, for
translating positions
        public Position differenceBetweenPositions(Position p2)
        {
            Position diff = new Position(this.getX() - p2.getX(), this.getY() -
p2.getY(), this.getZ() - p2.getZ());
            return diff;
        }

        //Prints the position to the console for testing purposes
        public void printPosition()
        {
            Console.WriteLine("Position: (" + x + ", " + y + ", " + z + ")" + "\n");
        }
```

```
        //Checks whether an X position is within specific distance from another X
position
        public bool withinPositionX(Position p, double difference)
        {
            if (this.absDifferenceBetweenPositions(p).getX() > difference)
            {
                return false;
            }

            else return true;
        }

        //Checks whether a Y position is within specific distance from another X
position
        public bool withinPositionY(Position p, double difference)
        {
            if (this.absDifferenceBetweenPositions(p).getY() > difference)
            {
                return false;
            }

            else return true;
        }

        //Checks whether a Z position is within specific distance from another X
position
        public bool withinPositionZ(Position p, double difference)
        {
            if (this.differenceBetweenPositions(p).getZ() > difference)
            {
                return false;
            }

            else return true;
        }

    }
}
```

Appendix C: Full Exercise Testing Results

C1: Full Performance Results

Left Shoulder Abduction	Right Shoulder Abduction	Seated Left Shoulder Abduction	Seated Right Shoulder Abduction	Left Hip Abduction	Right Hip Abduction
91.161	86.579	83.633	77.250	83.863	84.686
82.647	79.273	83.396	82.950	70.835	69.807
90.036	91.474	77.462	82.870	74.186	83.453
79.473	85.219	75.833	87.015	85.625	85.202
80.768	74.248	85.407	79.309	92.072	89.634
81.419	74.945	84.097	84.784	71.722	86.948
85.366	79.899	86.397	78.160	83.438	90.025
85.455	87.222	69.081	85.472	86.665	89.381
87.306	89.458	84.092	79.209	92.338	84.972
78.141	84.272	88.826	87.259	86.169	85.430
83.238	85.163	84.867	80.567	82.166	88.566
79.494	87.436	77.596	83.627	79.501	60.879
76.706	82.183	84.815	82.022	77.419	84.742
83.119	92.600	86.947	89.756	84.352	72.399
82.239	85.694	68.172	80.036	83.958	78.548
83.398	87.574	81.129	78.185	85.359	88.907
87.031	84.975	77.857	79.759	79.851	74.295
79.759	85.237	81.909	80.576	79.240	85.742
84.902	79.123	84.988	82.578	87.784	89.129
80.085	87.578	76.685	81.482	88.120	88.578

C2: Half Performance Results

Left Shoulder Abduction	Right Shoulder Abduction	Seated Left Shoulder Abduction	Seated Right Shoulder Abduction	Left Hip Abduction	Right Hip Abduction
59.969	46.953	54.128	50.333	61.770	53.409
51.277	51.236	39.097	47.819	62.747	55.567
53.679	43.656	35.154	45.453	52.593	58.107
58.912	43.963	39.563	47.096	63.577	57.190
60.120	47.718	39.060	50.314	47.584	59.664
54.892	45.538	42.491	46.506	59.363	51.031
48.247	45.823	44.058	47.375	47.023	52.718
54.958	47.522	48.160	51.106	48.716	53.272
54.515	43.415	39.811	49.267	57.376	53.263
54.604	48.981	47.525	51.157	57.349	48.908
51.019	49.752	42.937	45.558	52.001	53.049
52.540	44.913	42.392	51.399	52.494	51.801
51.893	48.969	39.501	48.048	58.715	48.028
54.911	44.546	39.718	48.942	55.095	52.149
53.910	47.887	35.667	46.128	48.897	49.368
48.191	46.040	43.077	51.337	58.402	53.962
53.829	47.310	39.261	51.222	53.011	52.895
50.005	48.614	44.414	51.136	48.339	50.617
53.367	44.280	43.449	46.188	47.024	49.384
48.403	50.692	36.686	48.356	59.562	49.400

C3: No Performance Results

Left Shoulder Abduction	Right Shoulder Abduction	Seated Left Shoulder Abduction	Seated Right Shoulder Abduction	Left Hip Abduction	Right Hip Abduction
37.194	26.660	28.299	35.381	28.321	32.708
39.431	26.540	32.750	36.884	34.927	30.755
38.636	26.725	25.462	35.461	33.590	31.517
33.096	27.683	30.217	37.420	30.990	36.817
39.567	26.945	34.499	34.968	31.272	37.410
33.494	26.531	27.242	35.274	34.751	30.846
37.262	26.144	29.609	34.909	33.835	31.939
35.250	26.019	31.956	35.662	32.954	34.599
36.479	25.425	29.040	34.478	31.507	30.304
37.936	25.461	27.870	34.054	37.507	31.060
36.853	26.254	31.566	34.283	34.379	33.677
35.040	26.897	30.777	35.320	32.132	34.999
33.046	25.200	28.500	35.259	30.422	31.192
37.816	26.485	31.976	35.243	34.049	31.642
34.510	25.256	26.741	34.237	32.268	32.141
33.519	25.906	27.849	35.954	34.338	30.005
33.512	26.488	27.680	35.972	33.816	34.468
37.850	25.133	31.391	35.256	33.622	34.743
33.980	25.698	26.443	35.524	31.552	34.881
37.066	25.240	29.167	35.308	35.173	34.472