

Media Engineering and Technology Faculty  
German University in Cairo



# Ju-Jutsu Training Kinect Application

Bachelor Thesis

Author: ElHassan Makled  
Supervisors: Assoc. Prof. Georg Jung  
Submission Date: 05 May, 2013



Media Engineering and Technology Faculty  
German University in Cairo



# Ju-Jutsu Training Kinect Application

Bachelor Thesis

Author: ElHassan Makled  
Supervisors: Assoc. Prof. Georg Jung  
Submission Date: 05 May, 2013

This is to certify that:

- (i) the thesis comprises only my original work toward the Bachelor Degree
- (ii) due acknowledgement has been made in the text to all other material used

---

ElHassan Makled  
05 May, 2013

# Acknowledgments

Text



# Abstract

Motion and gesture input systems are spreading in software development during the past few years. Some examples are Microsoft's Kinect, Nintendo's Wii, or Sony's Move. This spread ranged to affect video games and fitness programs. However, none of the fitness programs focus on actual contact sports techniques. In this project, we will use Microsoft Kinect to create an application that will help Ju-Jutsu (a martial arts form) practitioners to keep track of their performance.





# Contents

<b>Acknowledgments</b>	<b>V</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Background</b>	<b>3</b>
2.1 Architecture . . . . .	3
2.2 Development Tools and Technologies . . . . .	3
2.2.1 Microsoft XNA . . . . .	5
2.2.2 Kinect . . . . .	7
2.2.3 Connections and Communications . . . . .	14
2.3 Project Requirements . . . . .	14
<b>3 Application</b>	<b>17</b>
3.1 Recognition . . . . .	19
3.1.1 Glyphs Method . . . . .	19
3.1.2 Joint Position Lists . . . . .	20
3.1.3 MCS UK Solution Development Gesture Service . . . . .	24
3.2 Connecting to the Interface . . . . .	24
3.2.1 Pusher . . . . .	24
3.2.2 Socket Programming . . . . .	24
<b>4 Conclusion</b>	<b>25</b>
<b>5 Future Work</b>	<b>27</b>
<b>Appendix</b>	<b>28</b>
<b>A Lists</b>	<b>29</b>
List of Abbreviations . . . . .	29
List of Figures . . . . .	30
<b>References</b>	<b>31</b>



# Chapter 1

## Introduction

With the current rise of ubiquitous computing touchless input sensor devices have emerged to replace controllers in some video games, house hold appliances or lately, mobile phones. With these devices present, programmers take advantage of developing applications either for entertainment, such as dancing games, sword fighting games or shooting games, or fitness related applications, like yoga.

When it comes to fitness related applications, users are able to use them as support for keeping track of their performance. Some applications target to replace a coach, such as Nike+ Kinect training. This program guides the user through a series of exercises including sit ups, push ups and more. The user in the practice is able to recieve real time feedback and coaching.

All the emerging fitness applications mostly target activities that could be done at a limited space such as a living room. Only few applications target contact sports such as Ju-Jutsu, Muay Thai, or Boxing. However, there applications available using touchless input sensors that target these types of sports, but only focuses on the gaming experience and do not act as fitness applications.

Ju-Jutsu, Muay Thai, Karate, Boxing, and other contact sports have a certain practice that requires a practitioner and a coach, where the coach is holding two Thai-Pads, a thick pad that covers the arms of the coach so that the trainee would punch or kick for practice. Usually this practice is fast paced with the coach always signalling for a certain move to be executed by the practitioner. With the practice being fast paced the practitioners sometimes find it hard to recognize their mistakes or evaluate their performance.

Project Recon focuses on creating a tool for practitioners to evaluate and support in keeping track of their performance. Project Recon is part of a larger project titled Impact. Project Impact is a fitness monitor project that targets Ju-Jutsu practitioners. It has multiple plugins for applications. Those applications take input from sensors and process them. This information is later sent to Project Impact which will visualize the

information to the user. There are various types of information about the practitioner. All of which later reflect the performance and work out results of the user. The applications used as plug-ins to Project Impact include Project Recon. Which uses a Microsoft Kinect sensor to locate the practitioner and recognizes his techniques. Other possible sensors are the sensor equipped Thai-Pads and a heart rate monitor.

# Chapter 2

## Background

### 2.1 Architecture

The architectures shown below describes how communication between Project Recon and the general interface would look like. Figure 3.1 shows more details regarding the internal architecture and design of Project Recon itself.

As shown in figure 2.1, Project Recon main inputs are the Kinect sensor and Thai Pad Sensor. The Kinect sensor sends the Image stream and skeleton stream to Project Recon's Technique recognizer which in turn would take the skeleton of the user and keep track of its joints' position in time. When the practitioner moves the technique recognizer would compare the current movement with the database of reference moves. From this, it would associate the detected move with its respectful reference. The sensor equipped Thai Pads would measure the impact and send the value to Project Recon. The impact is then linked to its respectful technique and compared with reference impact. All this data is sent to the client which is responsible for the connection between the interface and Project Recon.

In figure 2.2, an overview of the project is shown. Where it shows the linking between the Main Interface and Project Recon. As previously said, the client is responsible for the connection between both by communicating and connecting to a socket specified for it by the Main Interface.

### 2.2 Development Tools and Technologies

When it came to developing with Kinect, two choices were on the table. Whether to use the official Microsoft Kinect SDK or use other unofficial SDK's. The official Kinect SDK may have slightly less capabilities, and we will be talking about them later and exploring them more. However, it is easier to use the official Microsoft SDK. The environment that

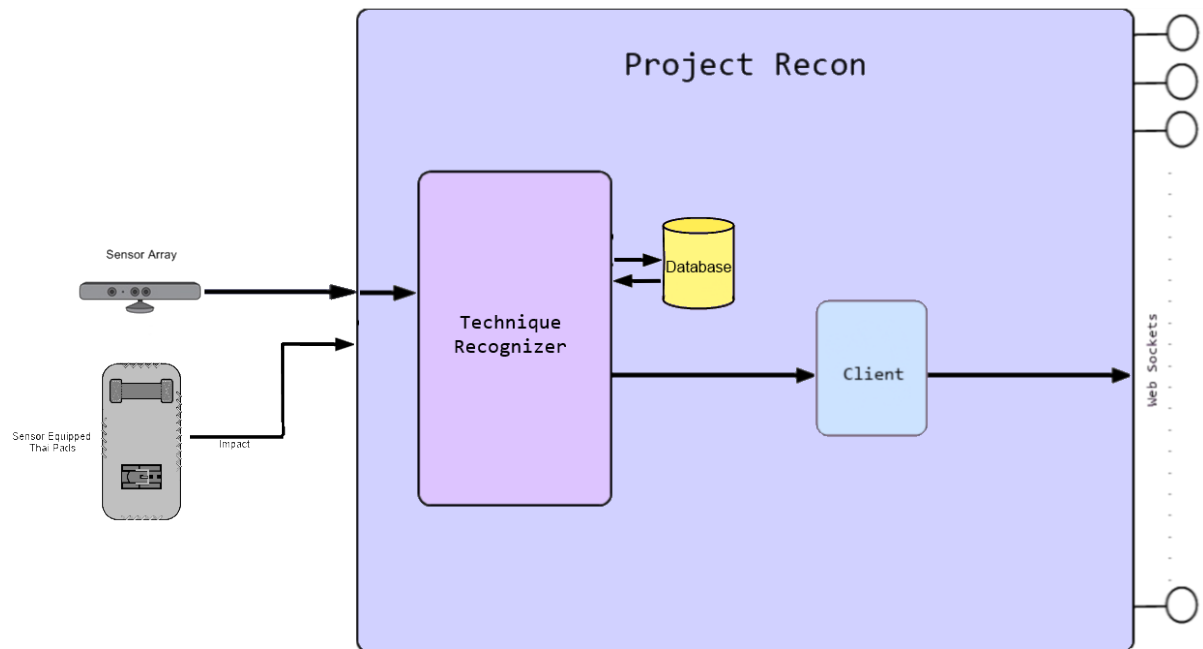


Figure 2.1: Internal architecture and design of Project Recon

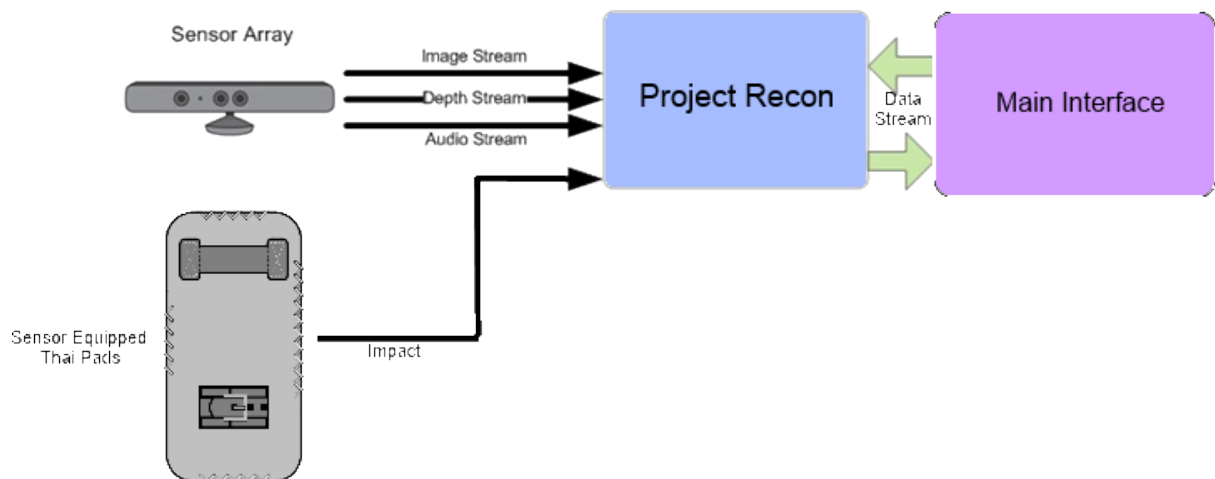


Figure 2.2: Overall architecture of Project Recon and its connection to the interface

was chosen for development (XNA) would also support our choice of using the official SDK. The programming language possible to use are Visual Basic and C#. The chosen language was C# as it was easier for me to develop for.

### 2.2.1 Microsoft XNA

XNA is an IDE (Integrated Development Environment) created by Microsoft to help game development for their systems to be easier. XNA is based on .NET Framework and it creates applications for platforms like XBox 360, Zune, Windows Phone and Microsoft Windows. Since its release there are 4 versions with the final version being XNA Game Studio 4.0. The framework was released on March of 2004 and the latest update released on September 16th 2010.

XNA is mostly used for XBox 360 video game development, which of course in turn is compatible with Kinect development, since Kinect is a Microsoft XBox 360 device. Making it easier to integrate the device within the code and collect data sent from the device and being able to read this data.

XNA makes it easier for developers to organize their code in many ways as it takes care of low level technologies related in game development. Helping developers to focus more on the gameplay and game detail itself. To make it more clear, when creating a class using XNA. Certain methods are created under a class titled *Game1* this class inherits from *Microsoft.XNA.Framework.Game*. This class includes several methods that help in organizing the code.

*Initialize()*, in this method every single variable is initialized. In our case since we use the Kinect, several initialization took place. For example, initializing the variable kinect to a connected Kinect Sensor. As well as initializing the elevation angle of the Kinect itself. Every other variable is initialized in this method.

Code:

```
1 protected override void Initialize()
2     {
3         // TODO: Add your initialization logic here
4
5         kinect = KinectSensor.KinectSensors[0];
6
7         kinect.ColorStream.Enable(ColorImageFormat.
8             RgbResolution640x480Fps30);
9
10        kinect.DepthStream.Enable(DepthImageFormat.
11            Resolution320x240Fps30);
12
13        kinect.SkeletonStream.Enable();
```

```

13         kinect.Start();
15
16         kinect.ElevationAngle = 0;
17
18         colorData = new byte[640 * 480 * 4];
19         colorTex = new Texture2D(GraphicsDevice, 640, 480);
20
21         rawSkeletons = new Skeleton[kinect.SkeletonStream.
FrameSkeletonArrayLength];
22
23         prevPositions = new SkeletonPoint[20];
24         littleGuy = new SkeletonPoint[20];
25         transGuy = new SkeletonPoint[20];
26
27         spine = new DetectGesture(30);
28         lshoulder = new DetectGesture(30);
29         rshoulder = new DetectGesture(30);
30         rwrist = new DetectGesture(30);
31         lwrist = new DetectGesture(30);
32         relbow = new DetectGesture(30);
33         lelbow = new DetectGesture(30);
34         rknee = new DetectGesture(30);
35         lknee = new DetectGesture(30);
36         lankle = new DetectGesture(30);
37         rankle = new DetectGesture(30);
38
39         TransValue = new Vector3(0, 0, 0);
40
41         lpunch = "";
42         rpunch = "";
43         lastMove = "";
44
45         keepcover = true;
46
47         base.Initialize();
48     }

```

listings/xna.cs

The *LoadContent()* method as its name specifies is responsible for loading the content required for the game. These contents range from images to audio or even video content that will be used in game as textures or in game music.

Code:

```

1     protected override void LoadContent()
2     {
3         // Create a new SpriteBatch, which can be used to draw textures
4
5         spriteBatch = new SpriteBatch(GraphicsDevice);
6
7         circleTex = Content.Load<Texture2D>("circle");
8         lineTex = Content.Load<Texture2D>("4KWjQ");
9     }

```



```
9      lineTexRed = Content.Load<Texture2D>("4KWjQR");
      rightTex = Content.Load<Texture2D>("richt");
      leftTex = Content.Load<Texture2D>("links");
11
      font = Content.Load<SpriteFont>("MainFont");
13      // TODO: use this.Content to load your game content here
    }
```

listings/xna.cs

There is also an *UnloadContent()* method that is used to unload the content once the game is done using them. This method is helpful with multi level games as loading all the game content unto the gpu will be impossible so items will be loaded when needed and unloaded when they are no longer needed during the gameplay to give space for more items to be loaded. In our project we will not need the Unload method as there aren't any high processing going on the graphics level.

The *Update()* method takes the gameTime as an argument, which is a snapshot of the game timing state. This method is called every interval. Updating all variables and values that have changed from previous frames. In our case, the positions of the joints of the skeleton. Later, the draw method would take care in rendering the images and displaying them. The two figures below show two different frames in time with the skeleton in a position and in the other time in another position.

In the following figures 2.3 and 2.4, two frames are shown where the skeleton first is standing regularly and then, later in time standing on one foot and leaning slightly to its left. As said before the update method updated the data from the previous method in terms of each joint's location in space and then the draw method rendered the skeleton with the new joints' locations.

### 2.2.2 Kinect

What makes Kinect sensor a good choice for this project is its ability to recognize and keep track of a person's movement through creating a virtual skeleton that matches the user and makes the skeleton move with the user. There are few constraints unfortunately when it comes to incorporating it with the Jiu Jitsu training. One of which would be the fact that Kinect does not differentiate between a user who is facing the Kinect from a user who is giving his back to the Kinect sensor.

The Kinect sensor consists of three major parts. A regular camera used mostly for video chat on the Xbox and the Kinect and sometimes visualization (e.g adding the user inside the scene of the game). The other two major parts are the ones that detect the user and their motion. The first is an infrared emitter and the other is an infrared sensor. The way the Kinect works is that the infrared emitter emits a mesh of infrared rays that would reflect back to the Kinect off from the user and then are detected by the sensor.

**Project Recon v1.0**  
**False**

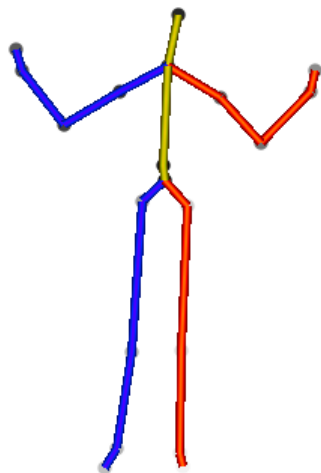


Figure 2.3: initial frame

**Project Recon v1.0**  
**False**

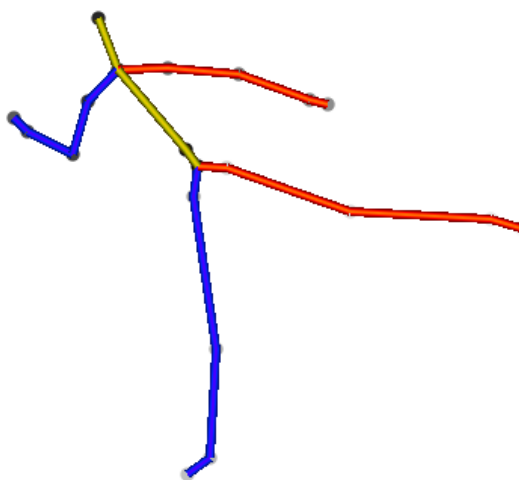


Figure 2.4: few seconds later from initial frame

The Kinect sensor has sends as an output three streams, an audio stream, a depth image stream and a color image stream. In figure 2.5 the color image stream is used to render the regular image captured by the Kinect camera.



Figure 2.5: Color stream data visualization

Other streams that the Kinect provides as output are the skeleton stream, Kinect provides an array of skeletons (having skeletons of all available users). So far the array has a maximum of four users at a time. The skeleton stream can be used to acquire information regarding the skeleton of a certain user. In figure 2.6, a circle was used to identify the places of the joints and was drawn on its respective joint. Different shades of grey were used in this image to show the system's differentiation of the joints.

There is a total of 20 joints recognized by the Kinect sensor. The head, the center shoulder, right shoulder, left shoulder, right and left elbows, right and left wrists, right and left hands, right and left hips, right and left knees, right and left ankles, right and left feet, a center hip and a spine.

After visualizing the joints, the next step was to link the joints with a line to show



Figure 2.6: Color stream data visualization with a visualization of the joints that is provided from the skeleton stream



the skeleton itself. By simple line equations and using the two points as the joints, the skeleton was drawn in the following manner:

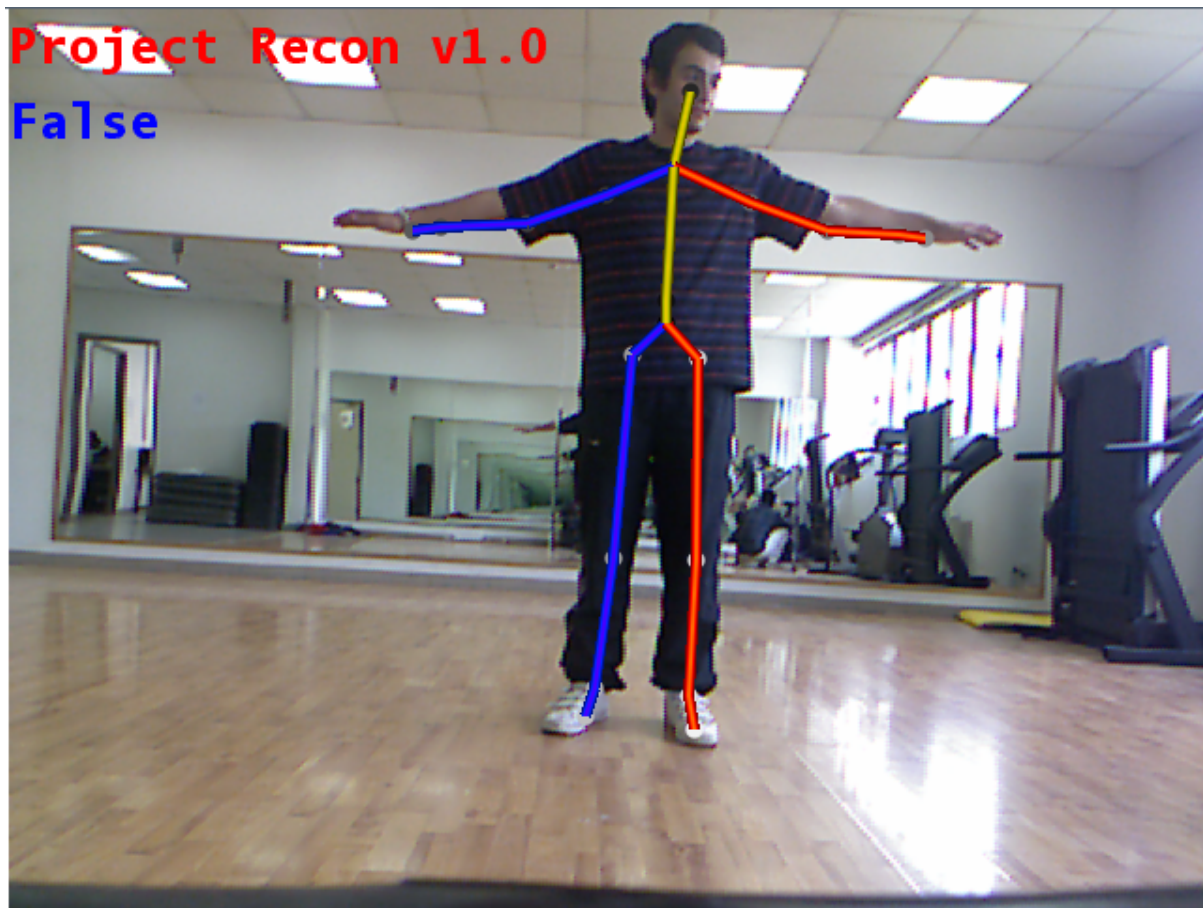


Figure 2.7: Color stream data visualization and skeleton visualization

Kinect uses three different coordinate systems. Each one is different to the other, however have relations to one another. Those spaces are Skeleton Space, Color Space and Depth Space. Kinect streams information on those three spaces during each frame.

The Color Space is a 2D space having only X and Y as its coordinates. During each frame, the color sensor (camera) captures an image of everything in the visibility box of the camera itself. The frame of the Color Space is made up of pixels, where the size is determined by the specified `NUIIMAGE_RESOLUTION`. Resolution could be either 80 x 60, 320 x 240, 640 x 480, or 1280 x 960. Every pixel contains the three values of red, green and blue at the particular coordinate. Figure 2.6 shows the data of received from the Color Space.

Similar to the Color Space, the Depth Space is a 2D space with X and Y as its coordinates. Each frame is represented as a gray scale image of the visible objects to the camera. The frame size is also determined by the specified `NUIIMAGE_RESOLUTION`, where each pixel contains the Cartesian distance between the camera plane and the

nearest body. The distance is measured in millimeters. Figure 2.8 shows how the Kinect measures the depth from the camera plane.

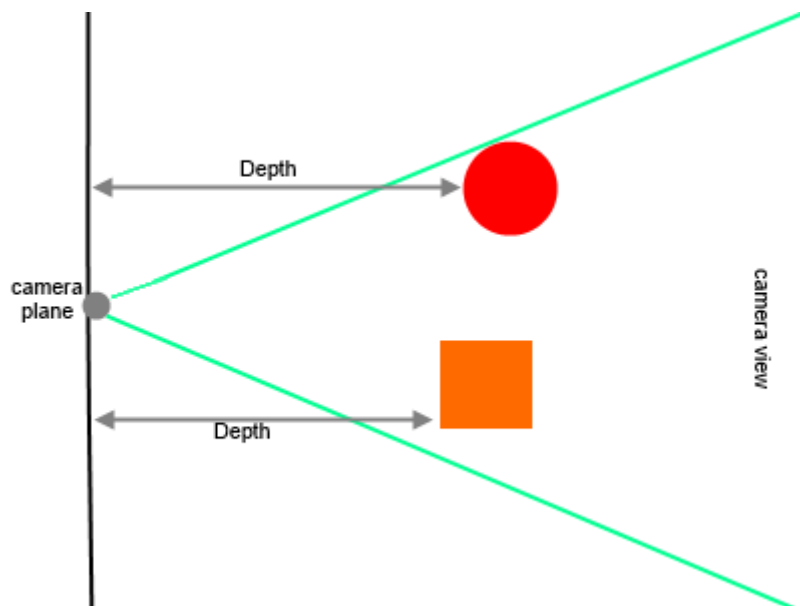


Figure 2.8: Visualization of the Kinect depth measurement

There are three different values that decide whether or not the object's location could be measured. The three values are, too near, too far, or unknown. Too near means that the object is detected, however is too near to the camera for the sensor to measure the distance accurately. Too far is similar to the too near value, however the reliability of measuring the distance is affected by the object being too far from the sensor. Finally, the unknown value means that the sensor does not detect any object. There are two different ranges for depth. Default range, which is found on Kinect for Windows sensor and the Kinect for Xbox 360 sensor. The second is the near range which is found only in the Kinect for Windows sensor. The reason for this, is that the Kinect for Windows is connected on the PC, and most of the time PC users will be sitting close to the PC and hence close to the Kinect for Windows sensor.

Skeleton Space, unlike the Color and Depth spaces is 3D (X, Y, Z). The Skeleton Space gets its data from the frames of the captured depth image. Where the Kinect processes the data and creates skeleton data that contains 3D position for the user's skeleton. This data is stored in the 3D space coordinate system which is expressed in meters.

In figure ?? the coordinate system of the Skeleton Space is visualized. The Kinect is set as the origin of the coordinate system, having its view pointed towards the positive z-axis. The y and x axis extend upwards and to the left respectively.

The skeleton data is a mirrored image from the user. The reason for this is that the visualization of the user's avatar during a game or any sort of application will be facing the user himself out from the screen. Making it very similar to look into a mirror allowing users to have an easier experience controlling their avatars. This mirroring effect can be

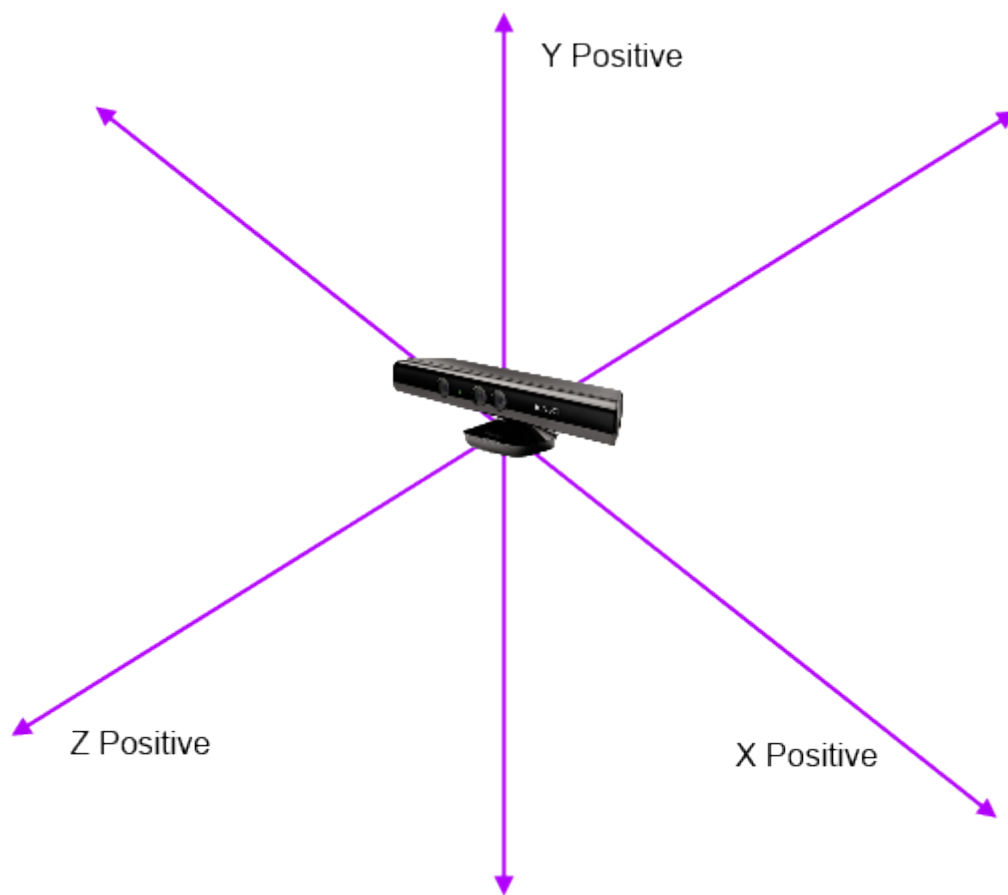


Figure 2.9: Skeleton Space coordinates

changed according by using a transformation matrix to flip the z-coordinates, but this depends on the nature of the application and its scope.

The Kinect system has different API's (Application Programming Interface) that are used to convert between the three different spaces. Some of the available conversions are from Skeleton to Depth and vic versa, and from Depth to Color.

### 2.2.3 Connections and Communications

The communication between Project Recon and the interface is seen as a client-server communication. When browsing the possible technologies to use, socket programming came at the top of the list.

Socket Programming uses the client server model of network communication, in which a server (in this case the interface) would listen on different ports for data arriving from any of the devices that are supposed to connect to it. When Project Recon starts and is ready it will take the part of the client, as it will send its data stream to the specified socket. When ready, a handshake will occur to initialize connection between starting to send the data stream. In which the interface will receive the data of gestures and delivers the proper output.

There are three different data streams, a data stream responsible for system-related gestures, like a gesture to start the session or end it. The other stream consists of technique details. Finally, the third stream is contains the impact of the technique. All of the three streams have a timestamp in order to associate each impact with the proper technique used.

## 2.3 Project Requirements

The practice that is targeted by the project is fast paced, which means the system needs to be performant enough to handle fast motion and recognize the technique in real time. Also, for it being a dynamic practice, which means rapid movement takes place during practice and the user will not necessarily be always facing the Kinect as he will be moving freely around the practice room and most of the times will be giving his back to the Kinect. We need to make the system robust enough to be able to recognize different motions.

Another basic requirement is to allow the differentiation between two different users infront of the Kinect, one being the coach and the second being the practitioner. This is easily resolved as Kinect's stream sends multiple skeletons of the multiple users. However it is required that the first user be defined as a the practitioner and the second to join in later would be the coach.



Finally, the project should be able to plugin to a different system, the fitness monitor. The fitness monitor is an interface that shows the user's current stats, technique executed, and more. A connection should be established between Project Recon and the fitness monitor, where Project Recon should be able to send information regarding the technique executed by the user for the fitness monitor display.



# Chapter 3

## Application

In order to create a robust system, two solutions are proposed. The first utilizes the fact that a person faces the same direction their waist faces. So we create a plane between three fixed points, the center hip  $\vec{c}$ , the left  $\vec{l}$  and the right hips  $\vec{r}$ . A normal vector  $\vec{N}$  was created from this plane in order to allow the program to recognize where the person is facing.

**Project Recon v1.0**

**False**

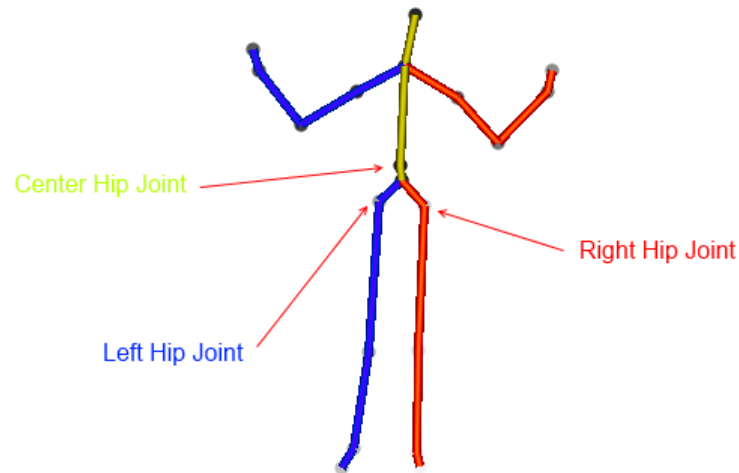


Figure 3.1: In the figure the three joints are pointed out and named

To calculate the normal vector, we subtract the right and left hip joints from the center hip joint and then we calculate the cross product of the two subtractions.

$$\vec{N} = (\vec{r} - \vec{c}) \times (\vec{l} - \vec{c})$$

We then calculate the normal by using the Normalize method.

```
var dir = Vector3.Cross(rhip - chip, lhip - chip);
var norm = Vector3.Normalize(dir);
```

As the person turns, the normal vector will be used as a reference to calculate the angle between it and the Z axis. This will allow us to rotate the skeleton so that it will be always facing the Kinect. Recognition will then take place.

Kinect has another problem, which is that it cannot differentiate between a user giving it his face or his back. In other words, a user's left and right are swapped if the user is giving the Kinect his back.

The normal vector calculation approach will reswap the left joints with the right joints when the  $\vec{N}$  is on the negative Z axis. However, here another problem emerges. When the swap takes place, an error occurs. Mainly, the programmer can not edit the value of the position of the joints. The value is read from the skeleton stream of the Kinect. In order to resolve this, an avatar skeleton needs to be created, littleGuy. littleGuy is an array of SkeletonPoints, where every SkeletonPoint represents a joint, and gets the value from these joints. This helps in allowing us to edit the SkeletonPoints as we wish and in the end draw this avatar skeleton instead.

In figure 3.2 the user is facing their right, having the normal vector (red line) erected and defining the direction the user is facing.

The translation and rotation of the skeleton occurs in the Skeleton Space. The algorithm translates littleGuy to the origin point where it calculates the  $\vec{N}$ . Rotation occurs in the origin and then the new manipulated littleGuy translates back to a fixed point in the Z axis, the Y axis and X axis however remain in the origin. After this Project Recon stores the translated littleGuy in a new avatar skeleton called transGuy. The program then renders transGuy to the 2D Color Space.

The second step is using Kinect's face detection capabilities, which may be hard and require high processing. Errors may occur when Kinect re-swaps the joints to their right place. For this we will use the face detection capabilities of Kinect to create a sanity check. That is if the swap did occur and the user's face was visible, then the swap is a mistake and it will re-swap again to the correct state.

When it came to testing the normalization of the skeleton using the Kinect, a problem emerges. When the user has a joint not facing the Kinect, then the Kinect does not have the position value of this joint. The reason for this is that no infrared rays are reflected from this joint. In turn, when the rotation occurs the some joints have false space(X, Y, and Z) values. This error will question the validity of the rendered and final translated values of the joints. This error made us discard this method and resort a different one that will also aims to maintain the robustness of the system.

**Project Recon v1.0**

**False**

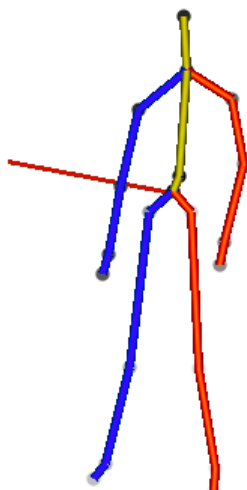


Figure 3.2: Skeleton with a normal vector defining where it is facing

## 3.1 Recognition

Recognition is the essence of this project. Recognition needs to be in real time, but most importantly how will the recognition occur. We discussed in the previous section the resolve of the normal vector. However, why is it needed? In recognition we want to be able to translate the skeleton of the person in order to always recognize from the same direction, facing the Kinect. In the coming sections we will discuss the different ways for recognition.

### 3.1.1 Glyphs Method

This is the proposed method in the case of recognition taking place after rotation by the help of  $\vec{N}$  as discussed earlier in this chapter. The idea behind it is to create a path taken by each joint from the skeleton and store it in an image. Creating a single image with the overall motion is and comparing it with the reference image, would be more optimal than taking a number of frames and comparing them with other reference frames.

As a joint moves, it creates a path shadowing its past movements. Each joint has its own exclusive color in order to differentiate if lines intersect. When the user moves the system will draw the paths taken by the joints creating some sort of a glyph image depicting the overall motion of the skeleton. An example of how a glyph would like is in both

figures below. Figure 1 shows the user as he executes a move with each joint creating a path. Figure 2 shows the final glyph as it would be stored.

After storing the glyph image, a process of image processing would take place where comparisons between the captured glyph image and the reference glyph images would begin. When a match is found, the technique would be recognized, if a match was not found the glyph image is discarded. Recognition when comparing two glyph images would of course have some sort of threshold as users are different in their executions of a technique.

Limitations that would emerge from using this method are the fact that motions on the z axis would be hard to recognize as the path would on the z axis wont be drawn. To solve this limitation, it is recommended that the rotation of  $\vec{N}$  around the y axis change. Rather than rotating the  $\vec{N}$  to the z axis it would rotate to form a  $45^\circ$  on the xz plane of the positive z and x axis. This way the glyph image would have a representation to motion detected on all three axes.

It should be clear as to when the program should record the path patterns and store the glyph. In order to know when exactly to record, we should first understand when will techniques be executed. As stated earlier, the coach gives signals to the practitioner for him to execute specific moves. Since the coach will be considered a user, it is recommended that recording of path paterns takes place as soon as a signal is initiated by the coach.

Since we will be unable to use the method of normalizing the skeleton due to the limitations of the Kinect sensor, this method would be obsolete in our application as the user is mobil and active around the room. Making it hard to have standard glyph patterns.

### 3.1.2 Joint Position Lists

This method focuses on creating a list of the last 30 positions of a joint. Where the system stores the newest position to an already full list, the last position(the earliest element on the list) is dicarded and replaced by the newest, see figure Since the application's frame-rate is 30 frames per second. The system takes every 30 positions each second and between each two frames it has inbetween lists, see figure

We create a class, StoreGesture, that describes the element of the list. StoreGesture has two attributes. Position, which is a 3D vector describing the joint's position. The second is Time, which is the specific time on which that specific position takes place. The code below shows the class StoreGesture.

```

namespace Project_Recon
2 {
    public class StoreGesture
4 {
        public DateTime Time { get; set; }

```

```

6      public Vector3 Position { get; set; }
8  }

```

listings/StoreGesture.cs

A new class DetectGesture is responsible for creating the list of StoreGesture and detecting the gestures. The way this works is that a list, posList is set to contain a list of StoreGesture and a maximum size of the list is set, which always is 30. In DetectGesture there are methods that would get the values inside the list, add an element to the list, and use the list to recognize a gesture.

Each different gesture has a series of joints that accent this gesture. Meaning that if the gesture is a punch, then the joints that are important and mostly resemble the punch are the elbow and wrist joints. Same thing for a kick, the knee's and the ankle's positions are important in order to recognize the gesture.

For each joint that would be later detected we create a DetectGesture with a size of 30 frames in the main class of Project Recon.

```

2      spine = new DetectGesture(30);
      lshoulder = new DetectGesture(30);
      rshoulder = new DetectGesture(30);
4      rwrist = new DetectGesture(30);
      lwrist = new DetectGesture(30);
6      relbow = new DetectGesture(30);
      lelbow = new DetectGesture(30);
8      rknee = new DetectGesture(30);
      lknee = new DetectGesture(30);
10     lankle = new DetectGesture(30);
      rankle = new DetectGesture(30);

```

listings/xna.cs

As the code updates, we update this list by adding the new position of the joint to its respective list by using the method in DetectGesture known as addPosition. This method takes as arguments a SkeletonPoint and a Kinect sensor. It creates a new StoreGesture entry with the given SkeletonPoint in its arguments. Then, it checks if list is full. If the list is full it would remove the earliest entry. Then in the end it would add to the list the new entry.

```

public virtual void addPosition(SkeletonPoint position , KinectSensor sensor
2  )
    {
        StoreGesture newEntry = new StoreGesture { Position = new
        Vector3(position.X, position.Y, position.Z) , Time = DateTime.Now };
4
        // Remove too old positions
6        if (posList.Count > MaxSize)
        {

```

```

8          StoreGesture entryToRemove = PosList[0];
          PosList.Remove(entryToRemove);
10      }
      // Add new position
12      posList.Add(newEntry);
  }

```

listings/DetectGesture.cs

As discussed earlier, the system needs to be robust enough to recognize gestures done at any angle the user is facing from or away of the Kinect. In order to achieve this, gesture detection needs to be exclude relations with the z and x axis as they will be dynamically changing with the time. Instead, a good way to tackle this is to detect a gesture through analyzing relations between joints and each other. For example, in a specific kick (known as the forward thrust) the knee will be higher than the hips in most cases. So taking advantage of this relation will help in achieving the robustness of the system that is required.

The Detect method in DetectGesutre, takes two arguments, reference1 and reference 2. Those arguments resemble two different position lists of reference joints that will be used in order to aquire relationship between them and the main joint. In order to recognize a gesture, we take divide a gesture into three different parts. Let us take a punch for example. In figure we divide the punch into three main motion frames. By observing the frames one can simply find a relation between the joints and each other. In this example the elbow is closer to the spine in the first and third frame in terms of the y axis, however closer to the shoulder in the second frame. Overall, another added information is that the angle between the elbow-shoulder and shoulder spine in a punch is more than 45°.

Similarly, the Detect method divides the frames into 3 different parts, each with 10 elements in them. A loop goes through them and whenever one element fits the criteria of one of the parts, a value that is responsible for calculating the probability is incremented. After the loop goes through the whole 30 frames. The probability value is checked, where 30 is 100 percent match. A threshold is given that is 25. Depending on the probability value either passing or not passing the threshold the gesture will be detected or not detected.

```

1  public String Detect(List<StoreGesture> reference1 , List<StoreGesture>
    reference2)
    {
3      if (PosList.Count == MaxSize)
        {
5          //Punch
          //r1 spine and r2 shoulder
7          for (int i = 0; i < MaxSize; i++)
            {
9              float a = PosList[i].Position.Y;
              float r1 = reference1[i].Position.Y;
11             float r2 = reference2[i].Position.Y;
            }
        }
    }

```



```

13         float diff1 = Math.Abs(a - r1);
14         float diff2 = Math.Abs(a - r2);
15
16         float magnitude1 = (float)Math.Sqrt(Math.Pow((
17 reference2[i].Position - PosList[i].Position).X, 2) +
18         Math.Pow((reference2[i].Position - PosList[i].
19 Position).Y, 2) +
20         Math.Pow((reference2[i].Position - PosList[i].
21 Position).Z, 2));
22
23         float magnitude2 = (float)Math.Sqrt(Math.Pow((
24 reference2[i].Position - reference1[i].Position).X, 2) +
25         Math.Pow((reference2[i].Position - reference1[i].
26 Position).Y, 2) +
27         Math.Pow((reference2[i].Position - reference1[i].
28 Position).Z, 2));
29
30         float angle = (float)Math.Acos((Vector3.Dot((reference2
31 [i].Position - PosList[i].Position), (reference2[i].Position -
32 reference1[i].Position))) /
33         magnitude1 * magnitude2);
34
35         if (i < 11 && diff1 < diff2)
36         {
37             punchprob++;
38         }
39
40         if (i < 21 && i > 11 && diff1 > diff2)
41         {
42             punchprob++;
43         }
44
45         if (i > 21 && i < 30 && diff1 < diff2)
46         {
47             punchprob++;
48         }
49
50         //if (angle > 0.25f * Math.PI)
51         //{
52             punchprob = 30;
53         //}
54     }
55
56     if (punchprob > 25)
57     {
58         punchprob = 0;
59         return "punch";
60     }
61     else
62     {
63         punchprob = 0;
64     }

```

```

59         }
        }
        return "No Move Detected";
    }
}

```

listings/DetectGesture.cs

Since every  $\frac{1}{3}$ rd of a second the class adds a position and removes another and then checks the latest list with the new position, then a gesture might be recognized more than once. In order to avoid this, we define a minimal period between gestures. The list becomes a completely new list every second. So the minimal period between gestures is 1 second. However, what if the person is fast enough to execute 2 gestures in less than two seconds. If this is the case and the system recognizes a gesture before the minimal period between gestures elapse. The system will compare it with the last gesture recognized. If they are the same, then it is discarded and not taken as a gesture. If not then the system considers it a gesture.

### 3.1.3 MCS UK Solution Development Gesture Service

DISCUSS HERE Microsoft Consulting Services gesture service and if it would be used how to achieve robustness

## 3.2 Connecting to the Interface

After recognition takes place and Project Recon recognizes a gesture. Project Recon needs to send the gesture and the time frame which the gesture took place to the interface so that the interface would view it to the user in real time during practice and also keep track of his performance.

Two approaches were considered. The first focuses on using socket programming to create a server and client side where communication would occur on certain ports. The second was to use a service known as Pusher.

### 3.2.1 Pusher

### 3.2.2 Socket Programming

# Chapter 4

## Conclusion

Conclusion



# Chapter 5

## Future Work

Text

# Appendix

# Appendix A

## Lists

# List of Figures

2.1	Internal architecture and design of Project Recon . . . . .	4
2.2	Overall architecture of Project Recon and its connection to the interface	4
2.3	initial frame . . . . .	8
2.4	few seconds later from initial frame . . . . .	8
2.5	Color stream data visualization . . . . .	9
2.6	Color stream data visualization with a visualization of the joints that is provided from the skeleton stream . . . . .	10
2.7	Color stream data visualization and skeleton visualization . . . . .	11
2.8	Visualization of the Kinect depth measurement . . . . .	12
2.9	Skeleton Space coordinates . . . . .	13
3.1	In the figure the three joints are pointed out and named . . . . .	17
3.2	Skeleton with a normal vector defining where it is facing . . . . .	19



# Bibliography

- [1] W.G. Campbell. *Form and style in thesis writing*. Houghton Mifflin, 1954.
- [2] S. Wenkang. An analysis of the current state of English majors' BA thesis writing [J]. *Foreign Language World*, 3, 2004.