Gesture-Based Human-Computer-Interaction Using Kinect for Windows Mouse Control and PowerPoint Presentation

Toyin Osunkoya[1], and Johng-Chern Chern[2]
Department of Mathematics and Computer Science
Chicago State University
Chicago, IL 60628
[1]tosunkoy@csu.edu, and [2]jchern@csu.edu

Abstract

One of the most important research areas in the field of Human-Computer-Interaction (HCI) is gesture recognition as it provides a natural and intuitive way to communicate between people and machines. Gesture-based HCI applications range from computer games to virtual/augmented reality and is recently being explored in other fields.

The idea behind this work is to develop and implement a gesture-based HCI system using the recently developed Microsoft Kinect depth sensor to control the Windows Mouse Cursor as well as PowerPoint presentations.

The paper can be divided into two major modules namely, hand detection and gesture recognition. For hand detection, the application uses the Kinect for Windows Software Development Kit (SDK) and its skeletal tracking features to detect a user's hand which enables the user to control the Windows mouse cursor. Gesture recognition involves capturing user gestures and interpreting motions or signs that the user performs to simulate different mouse events.

1 Introduction

The development of ubiquitous computing and the need to communicate in a more natural, flexible, efficient but powerful way has rendered most current user interaction approaches which utilizes keyboard, mouse and pen insufficient.

Human-Computer-Interaction technology has seen significant changes over the years which range from text-based UI which relies heavily on the keyboard as an input device to 2-D graphical-based interfaces based on mice, to multimedia-supported interfaces, to fully-fledged multi-participant virtual environment (VE) systems. Although inconvenient, unnatural and cumbersome, these devices (keyboard and mouse) have dominated basic Human-Computer-Interaction (HCI) within different applications and their limitations have also limits the useable command set.

The desire to therefore provide a more natural interaction between humans and machines has brought about the wide focus on gesture recognition. Gestures have long been considered the most natural form of interaction among humans and it is defined as a motion of the body that contains information [1]. It involves the physical movement of the head, hands, arms, face or body with the aim of conveying semantic information.

This paper aims to provide a basic understanding on gesture recognition and how to develop an application which is able to recognize users and understand their intentions using a “natural” user interface consisting of gestures. To implement this, the skeletal tracking ability of the Kinect sensor is utilized as well as both the depth map and color image obtained by the sensor which enables the user to operate Windows 7 Operating System and explore its functionality with no physical contact to a peripheral device such as mouse. The predefined gestures recognized by the device allow the simulation of different commands and mouse behaviors.

This paper is organized as follows: Section 2 gives a brief overview of gesture recognition. Section 3: The Microsoft Kinect Sensor. Section 4: Implementing a gesture-based HCI with Kinect. Conclusions and further research areas are given in Section 5.

2 Gesture Recognition: Overview

Gesture Recognition is a technology that achieves dynamic human-machine interactions without requiring physical, touch, or contact based input mechanisms. The main goal of gesture recognition involves creating a system capable of interpreting specific human gestures via mathematical algorithms and using them to convey meaningful information or for device control [2].

2.1 Tracking Technologies

The main requirement for supporting gesture recognition is the tracking technology used for obtaining the input data. The approach used generally fall into two major categories: *Glove-based* approaches and *Vision-based* approaches.

In glove-based systems (see Figure 1), the common technique for hand pose tracking is to instrument the hand with a glove which is equipped with a number of sensors to provide input to the computer about hand position, orientation, and flex of the fingers using magnetic or inertial tracking devices. An example of such device is the Data-glove which was the first commercially available hand tracker [3].While it is however easier to collect hand configuration and movement with this approach, the major drawback is that the required devices are quite expensive and cumbersome. Also, the ease and naturalness with which the user can interact with the computer controlled environment is hampered by the load of cables attached to the user. More details about data-glove approaches are available in a survey on data glove by Dipietro et al. [4].

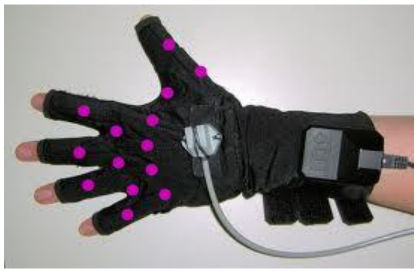


Figure 1: Glove-based system [5]

On the other hand, vision-based approach (see Figure 2) offers a more natural way of HCI as it requires no physical contact with any devices. Camera(s) placed at a fixed location or on a mobile platform in the environment are used for capturing the input image usually at a frame rate of 30 HZ or more. To recognize human gestures, these images are interpreted to produce visual features which can then be used to interpret human activity [6]. Major drawback of this approach is the problem of occlusion. The camera view is always limited as there are always parts of the user’s body that are not visible. More details about vision-based approaches are mentioned by Porta [7].

Figure 2: Vision-based system (From Google image gallery)

2.2 Gesture Recognition System

Gesture recognition is a complex task which involves many aspects and while there may be variations on the methodology used and recognition phases depending on the application areas, a typical gesture recognition system involves such process as data acquisition, gesture modeling, feature extraction and gesture recognition. The detail description of each process is given in [2] and [8].

3 Microsoft's Kinect

Of recent, gesture recognition has been integrated in various consumer devices for the purpose of entertainment. An example of such device is Microsoft's Kinect, which allows a user to use gestures that are typically intuitive and relatively simple to perform various tasks such as controlling games, starting a movie etc.

3.1 The Kinect Sensor

The Kinect sensor is a motion-sensing input device that was originally developed in November 2010 for use with the Xbox 360 but has recently been opened up for use with Windows PCs for commercial purposes.

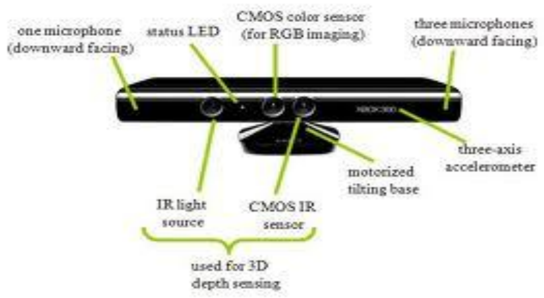

one microphone (downward facing)
status LED
CMOS color sensor (for RGB imaging)
three microphones (downward facing)
three-axis accelerometer
motorized tilting base
IR light source
CMOS IR sensor
used for 3D depth sensing

Figure 3: The Kinect Sensor Components [9]

3.1.1 Architecture

The Kinect works as a 3D camera by capturing a stream of colored pixels with data about the depth of each pixel. Each pixel in the picture contains a value that represents the distance from the sensor to an object in that direction [10]. This hardware feature provide developers the means for creating a touch-less and immersive user experience through voice, movement and gesture control although it does not inherently perform any tracking or recognition operations, leaving all such processing to software.

Skeleton tracking is generally handled by the SDK with gesture recognition left to the developer, though multiple libraries exist to aid in recognition of gestures. In addition, speech recognition is done by external SDKs such as the Microsoft Speech Platform [11].

The Kinect sensor as shown in Figure 3 has the following properties and functions:

- An ***RGB Camera*** that stores three channel data in a 1280x960 resolution at 30Hz. The camera's field of view as specified by Microsoft is 43° vertical by 57° horizontal [10]. The system can measure distance with a 1cm accuracy at 2 meters distance
- An ***infrared (IR) emitter*** and an ***IR depth sensor*** used for capturing depth image.
- An array of ***four microphones*** to capture positioned sounds.
- A ***tilt motor*** which allows the camera angle to be changed without physical interaction and a ***three-axis accelerometer*** which can be used to determine the current orientation of the Kinect.

3.1.2 Hardware Interface

The sensor interface with the PC via a standard USB 2.0 port; however an additional power supply is needed because the USB port cannot directly support the sensor's power consumption [12].

3.2 Hardware and Software Requirements

According to Microsoft, the PC that is to be used with the Kinect sensor must have the following minimum capabilities [13]: (a) 32-bit (x86) or 64-bit (x64) processors,(b) Dual-core, 2.66-GHz or faster processor,(c) USB 2.0 bus dedicated to the Kinect, and (d) 2 GB of RAM.

To access Kinect's capabilities, the following software is also required to be installed on the developer's PC: Microsoft Visual Studio 2010/2012 Express or other Visual Studio edition. The development programming languages that can be used include C++, C# (C-Sharp), and Visual Basic.

3.2.1 Kinect for Windows SDK

Installing Kinect for Windows SDK is necessary to develop any Kinect-enabled application. Figure 4 shows how Kinect communicates with an application. The SDK in conjunction with the Natural User Interface (NUI) library provides the tools and the Application Programming Interface (APIs) needed such as high-level access to color and calibrated depth images, the tilt motor, advanced audio capabilities, and skeletal tracking but requires Windows 7 (or newer) and the .NET Framework 4.0 [10].

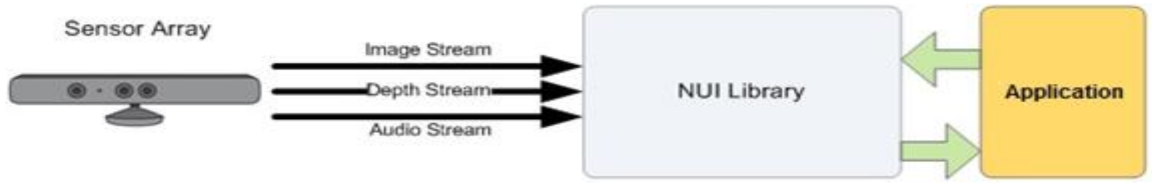


Figure 4**:** Kinect Interaction with an Application [13]

3.3 Limitations

Kinect has the following limitations which are primarily based on the optical lenses:

- For a smooth skeleton tracking, the user distance from the sensor must be between 1m and 3m.
- Kinect cannot perform Finger Detection. This is due to the low-resolution of the Kinect depth map which, although works well when tracking a large object such as the human body; it is however difficult to track human hand (which occupies a very small portion of the image) especially gestures made with fingers [14].
- Joint confusion may also occur when certain human joints are in the same position.

4 Implementing a Gesture-Based HCI System Using Kinect

Three key stages are involved in the development of the proposed HCI system. Firstly, the device must be able to track the skeletal features of a user to detect the hand before any gestures can be recognized and processed. Secondly, each gesture must be properly recognized. Thirdly, gestures recognized must be interpreted to carry out the actions related to it. In general, these three key stages spans around skeleton tracking and gesture recognition functions. This section discusses how this is achieved.

4.1 Choice of Programming Language

Most Kinect-enabled applications are usually developed with C#, C++, or Visual Basic. This interface is developed using Windows Presentation Foundation (WPF) in C# which provides a more human-readable code and a more simplified syntax.

4.2 Modeling Mouse Behaviors

The basic HCI input device used when accessing the desktop features such as icon, menu bars, windows etc. is the keyboard and mouse. Cursor movement on the screen is normally controlled with the movement of the mouse by a user. Mouse events allow for object selection, menu display, dragging an object, opening and closing an application etc. These mouse behaviors and the actions they trigger on a typical desktop can be summarized as follows:

- *Cursor Movement:* triggered with the movement of the mouse
- *Object Selection:* triggered by the pressing of the mouse left button (left click)
- *Opening an Application:* triggered by the pressing of the mouse left button twice in quick succession (double click)
- *Drop Down Menu:* triggered by the pressing of the mouse right button (right click)
- *Dragging Operation:* triggered by holding the left mouse down while moving the mouse from one position to the other.

To successfully implement the proposed HCI interface, modeling of these mouse events is required which can be quite a complex task. The best solution to achieve this is through finger gesture detection however finger detection is not possible with the Kinect due to its low resolution, although several algorithms [14] and [15] exist to serve as a workaround [16].

The alternate solution used in this paper is to assign the function of cursor movement to one hand while the user controls the other actions with the second hand by means of predefined gestures (See Section 4.4.6).

4.3 Hand Detection

The most important aspect of developing a gesture-based HCI system is to first track the user before any hand detection can occur. The skeletal tracking features of Kinect combined with the NUI library allow users and their actions to be recognized.

According to Microsoft, the infrared (IR) camera can recognize up to six users in its field of view while only two users can be tracked in detail. It should be noted that the Kinect sensor itself does not recognize people; it simply sends the depth image to the host device, such as an Xbox or computer. Software running on the host device contains logic to decode the information and recognize elements in the image with characteristic human shapes. The software has been "trained" with a wide variety of body shapes. It uses the alignment of the various body parts, along with the way that they move, to identify and track them [17].

With skeleton tracking, an application can locate twenty (20) skeletal joints of a user standing and ten upper-body joints (shoulders, elbows, wrists, arms and head) of a user sitting directly in front of the Kinect Sensor. Figure 5 shows the various joints relative to the human body.

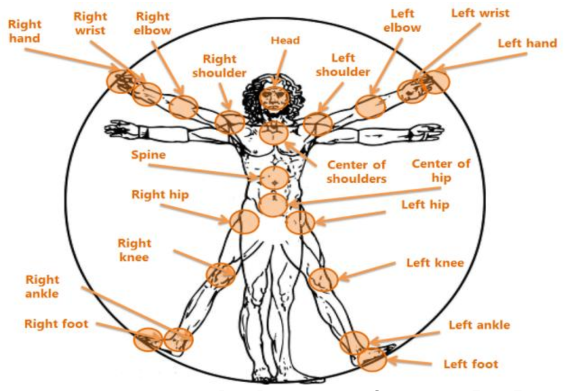


Figure 5: Tracked joints of a user [18]

After skeleton tracking is done, the position of each joint in 3D space is returned by the NUI library in the format of X, Y and Z coordinates expressed in meters according to the skeleton space coordinate system.

4.4 Developing the Application

4.4.1 Setting up the Kinect Sensor

Once the installation of the required software and hardware (Section 3) is completed, the first step in developing this interface involves referencing Microsoft.Kinect.dll assembly in the C# application and including the using directive for the namespace *Microsoft.Kinect*. This assembly calls unmanaged functions from managed code and is used to control the Kinect sensor.

Also a connection must be made to the sensor bar. This is done by obtaining an instance of the *KinectSensor* class which is part of the Kinect SDK. The *KinectSensor* instance acts like an "engine room" for the Kinect and its function involves processing commands specified in the developer's program while also sending out error messages [17].

4.4.2 Obtaining Data from the Sensor

Having made reference to the Kinect unit, the application must also be able to acquire the capabilities (such as the depth stream, the color stream, skeleton stream) needed from the Kinect sensor which is essential for the successful implementation of the proposed system.

This is done by enabling through coding the *ColorStream* component of the Kinect which provides the RGB video stream; the *DepthStream* which provides the 3D representation of the image in front of the sensor as shown in Figure 6, and enabling the *SkeletonStream* which is used for acquiring the skeleton data.

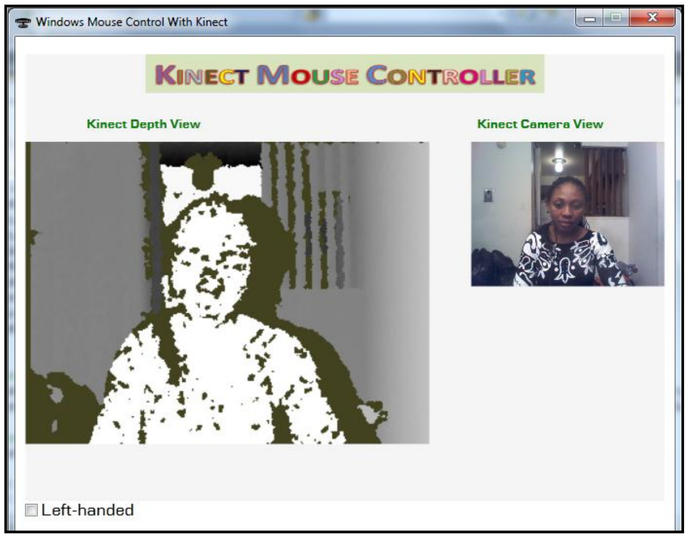
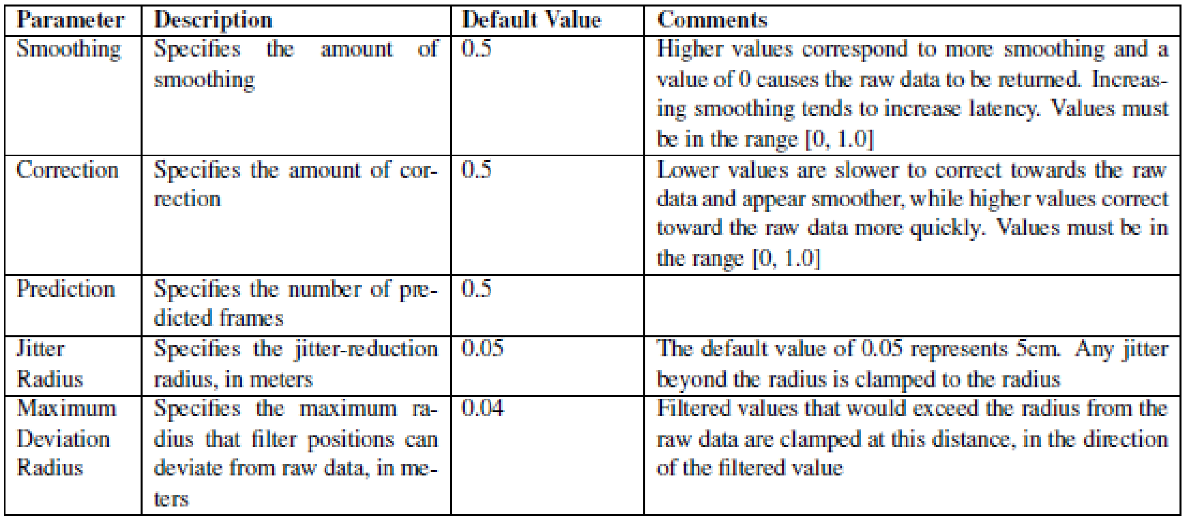


Figure 6: Depth and Camera Image Obtained From The Sensor.

4.4.3 Setting the Skeletal Tracking Parameters

To initiate the skeletal tracking features of the Kinect, some algorithm parameters (Table 1) must be set. The reason for this is that when the sensor *SkeletonStream* is enabled, skeleton data are sent out without smoothing or filtering and with a lot of noise. This is caused by the low resolution of the Kinect depth sensor which is unable to ensure consistent accuracy of the skeleton data over time [19].

It is therefore the responsibility of the developer to perform smooth skeleton tracking, prediction of skeleton, correction of skeletal tracking using the algorithm provided by the NUI library for filtering and smoothing incoming data from the sensor in order to reduce noise. It should however be noted that no set of best values exist for each parameter, the level of filtering and smoothing required depends on the application being developed.



Parameter	Description	Default Value	Comments
Smoothing	Specifies the amount of smoothing	0.5	Higher values correspond to more smoothing and a value of 0 causes the raw data to be returned. Increasing smoothing tends to increase latency. Values must be in the range [0, 1.0]
Correction	Specifies the amount of correction	0.5	Lower values are slower to correct towards the raw data and appear smoother, while higher values correct toward the raw data more quickly. Values must be in the range [0, 1.0]
Prediction	Specifies the number of predicted frames	0.5	
Jitter Radius	Specifies the jitter-reduction radius, in meters	0.05	The default value of 0.05 represents 5cm. Any jitter beyond the radius is clamped to the radius
Maximum Deviation Radius	Specifies the maximum radius that filter positions can deviate from raw data, in meters	0.04	Filtered values that would exceed the radius from the raw data are clamped at this distance, in the direction of the filtered value

Table 1: Parameters for Skeletal Smoothness [19]

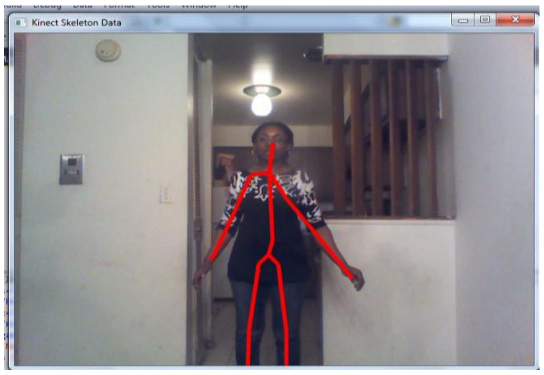
Besides setting the parameters for skeletal tracking, consistent data about the position of the hand that will be used to move the cursor must be obtained and the hand must be scaled to the screen size to ensure smooth cursor movement. This involves setting the skeleton frame X (*SkeletonMaxX*) position value which determines the distance that the cursor travels between the screen border when the hand is moved on X axis and setting the skeleton frame Y (*SkeletonMaxY*) position value which determines the distance the cursor travels once the hand is moved on Y axis between the screen border.

4.4.3 Initiating skeletal tracking for the joints and Obtaining Skeleton Information

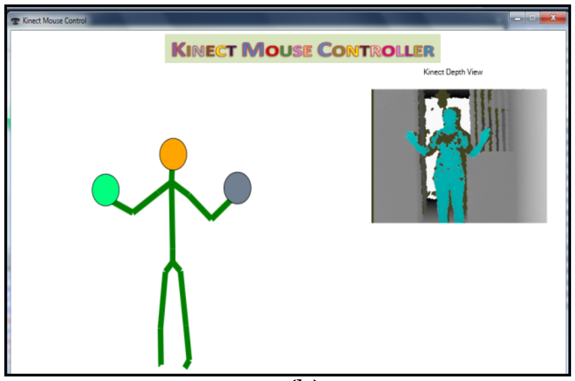
Once the skeleton stream is enabled, it can generate an event each time it has new skeleton data for the application. In the application, an event handler is connected to the following method used for obtaining the skeleton frame data:

```
sensor.AllFramesReady +=
newEventHandler<AllFramesReadyEventArgs>(sensor_AllFramesReady);
```

The method *sensor_AllFramesReady* is connected to the *AllFramesReady* event and will run each time that the skeleton tracking code in the Kinect SDK has completed the analysis of the scene in front of the sensor. When this method is called, it is given an argument of type *AllFramesReadyEventArgs* [17]. This contains a method called *OpenSkeletonFrame* that provides access to the frame of skeleton data (Figure 7).



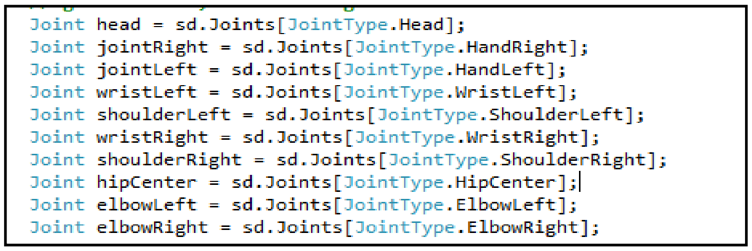
(a)



(b)

Figure 7: (a) A Kinect Skeleton (b) A Kinect Skeleton showing positions of the hand

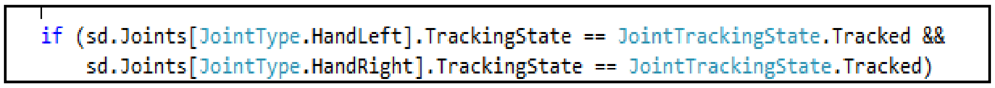
The application uses several joints data for (1) cursor movement, (2) gestures to simulate mouse events, and (3) for presentation control. Therefore, tracking these joints is of paramount importance, and unless they are initiated, Kinect cannot begin to track them. The code to initiate the tracking of these joints is given in Figure 8 below.



```
Joint head = sd.Joints[JointType.Head];
Joint jointRight = sd.Joints[JointType.HandRight];
Joint jointLeft = sd.Joints[JointType.HandLeft];
Joint wristLeft = sd.Joints[JointType.WristLeft];
Joint shoulderLeft = sd.Joints[JointType.ShoulderLeft];
Joint wristRight = sd.Joints[JointType.WristRight];
Joint shoulderRight = sd.Joints[JointType.ShoulderRight];
Joint hipCenter = sd.Joints[JointType.HipCenter];
Joint elbowLeft = sd.Joints[JointType.ElbowLeft];
Joint elbowRight = sd.Joints[JointType.ElbowRight];
```

Figure 8: Definition of Joints to Be Tracked

Figure 9 shows the C# code to track both the right and left hand in order to initiate cursor movement. By default, once both hands are detected (See Figure 7b), the right hand moves the cursor. However, the application also allows users who are left handed to control the movement of the cursor on the screen with their left hand. To move the cursor within the border of the screen, these hands need to be scaled to the primary screen width and height.

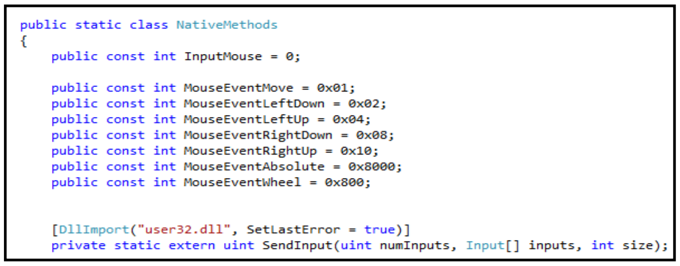


```
if (sd.Joints[JointType.HandLeft].TrackingState == JointTrackingState.Tracked &&
    sd.Joints[JointType.HandRight].TrackingState == JointTrackingState.Tracked)
```

Figure 9: Tracking a User’s Left and Right Hand

4.4.4 Cursor Assignment to the Hand and Cursor Movement

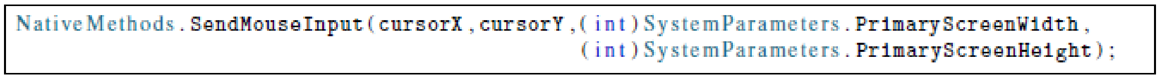
Before assigning the cursor to the hand and moving the cursor, a Win32 function called *SendInput* (Figure 10) must be imported. This function can send Windows messages to the topmost window on the computer screen to control inputs (mouse, keyboard, hardware). It uses some basic Win32 structures defined as follows for the mouse input:



```
public static class NativeMethods
{
    public const int InputMouse = 0;

    public const int MouseEventMove = 0x01;
    public const int MouseEventLeftDown = 0x02;
    public const int MouseEventLeftUp = 0x04;
    public const int MouseEventRightDown = 0x08;
    public const int MouseEventRightUp = 0x10;
    public const int MouseEventAbsolute = 0x8000;
    public const int MouseEventWheel = 0x800;


    [DllImport("user32.dll", SetLastError = true)]
    private static extern uint SendInput(uint numInputs, Input[] inputs, int size);
```

Figure 10: Library for Mouse Events and Movement

After the user's hand has been detected and scaled to a Windows PC screen size, the X and Y position of the cursor needs to be assigned to the user's hand. These parameters (Figure 11) are then sent to the Windows OS which in this case causes the *MouseEventMove* flag to be raised, thus once the user moves the right hand (or left hand if left-handed), the cursor also moves in the direction of the hand.

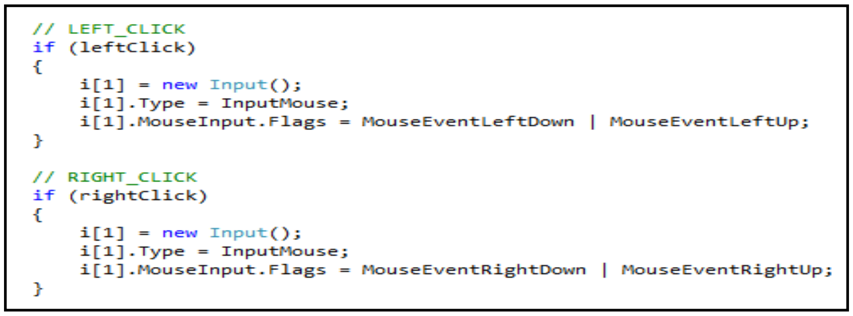


```
NativeMethods.SendMouseInput(cursorX,cursorY,(int)SystemParameters.PrimaryScreenWidth,
                                             (int)SystemParameters.PrimaryScreenHeight);
```

Figure 11: Sending The Windows Cursor X and Y position

4.4.6 Modeling Mouse Clicks and Detecting Predefined Gestures

The mouse Left-Click event flag is usually raised when a user hold the mouse left button down and then release it within a certain time frame. To simulate this action for example, the Windows OS user library described in Figure 10 above is used. The *MouseEventLeftDown* is sent to the OS followed by the *MouseEventLeftUp* corresponding to a user actually pressing a physical mouse left button. Double left click is achieved by performing the left-click action twice in quick succession. Figure 12 shows the C# coding for the Left and Right Click.

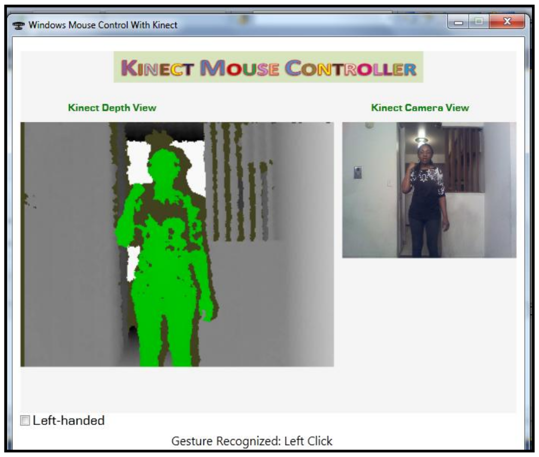


```
// LEFT_CLICK
if (leftClick)
{
    i[1] = new Input();
    i[1].Type = InputMouse;
    i[1].MouseInput.Flags = MouseEventLeftDown | MouseEventLeftUp;
}

// RIGHT_CLICK
if (rightClick)
{
    i[1] = new Input();
    i[1].Type = InputMouse;
    i[1].MouseInput.Flags = MouseEventRightDown | MouseEventRightUp;
}
```

Figure 12: Sample Mouse Events Simulation

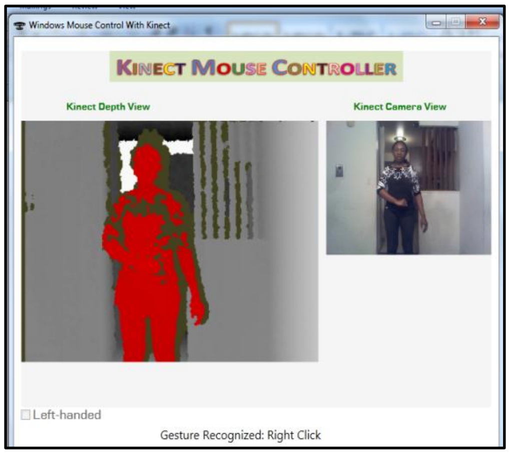
Each of the mouse events is triggered by gestures. The description of the gestures used for each mouse click is given as follows:

- *Left Click:* The distance between the left shoulder and the left wrist is calculated if the user is right handed and vice versa for a left-handed user. Section 4.4.7 describes how the distance between these two joints is obtained. If the distance is less than 0.2f, the left-click event is raised (Figure 13-a).
- *Right Click:* By default, Right Click is triggered if the distance between the hip center and the left wrist is less than 0.2f (Figure 13-b).
- *Double Left Click:* This is triggered if the hand is raised above the head (Figure 13-c)
- *Mouse Drag:* Holding the mouse left button down without releasing it allows for mouse drag. This event is triggered if the distance between the right shoulder and the left wrist is less than 0.4f (Figure 13-d).



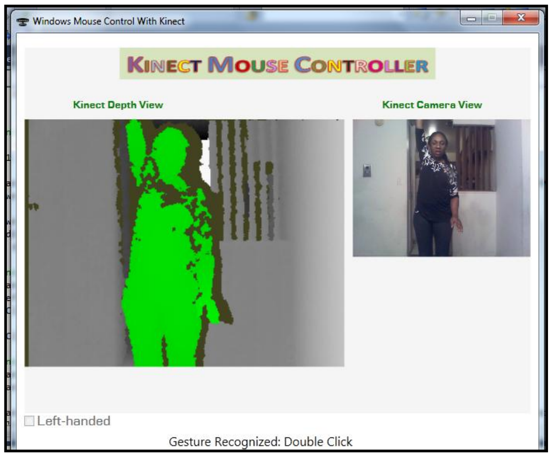
Gesture Recognized: Left Click

(a)



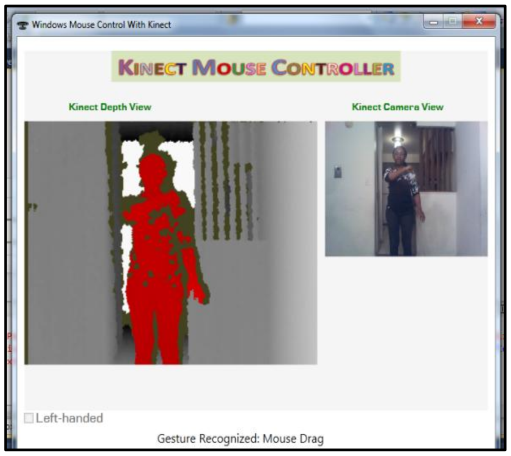
Gesture Recognized: Right Click

(b)



Gesture Recognized: Double Click

(c)



Gesture Recognized: Mouse Drag

(d)

Figure 13: (a) Gesture for *Left_click*, (b) Gesture for *Right_Click*, (c) Gesture for *Double_Click*, and (d) Gesture for *Drag* Operations

4.4.7 Calculating The Distance Between Two Joints In Space

Each joint position is given as 3D coordinates (X, Y, and Z) by the Kinect depth sensor. To better illustrate how to determine the distance between two joints in space in 3D; let's first consider how this is done in 2D using just the coordinates X and Y. The distance between two joint locations could be obtained by subtracting the X and Y components as follows:

```
float dX = first.Position.X - second.Position.X;
float dY = first.Position.Y - second.Position.Y;
```

As shown in Figure 14, this gives the distance across and the distance up that one needs to move when traveling from one point to the other.

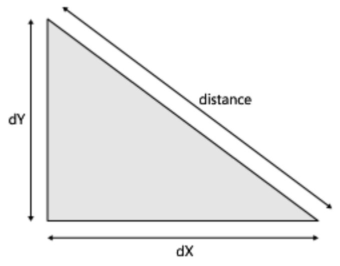
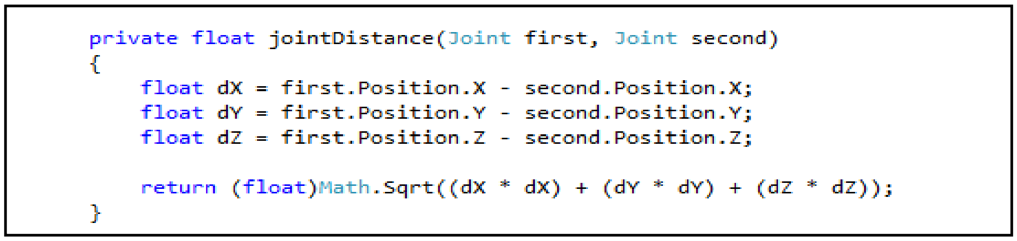


Figure 14: Triangles and Distances

Using Pythagorean Theorem, thus the distance could then be calculated as:

```
distance = Math.Sqrt((dX * dX) + (dY * dY));
```

The Math library in .NET provides a square root method called *Sqrt* that provides the distance value. The Pythagorean Theorem is also applicable in 3D, and the only thing required is to add the distance in the third dimension [17]. This is used to create the method in Figure 15 that returns the distance between two joints.



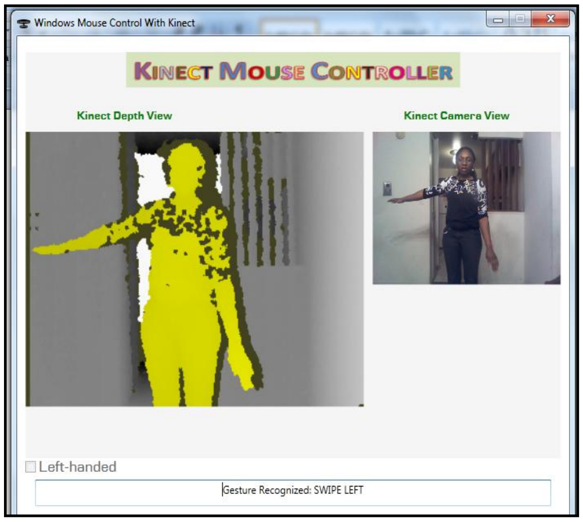
```
private float jointDistance(Joint first, Joint second)
{
    float dX = first.Position.X - second.Position.X;
    float dY = first.Position.Y - second.Position.Y;
    float dZ = first.Position.Z - second.Position.Z;

    return (float)Math.Sqrt((dX * dX) + (dY * dY) + (dZ * dZ));
}
```

Figure 15: Method for Calculating the Distance between Two Joints

4.4.8 PowerPoint Presentation Control

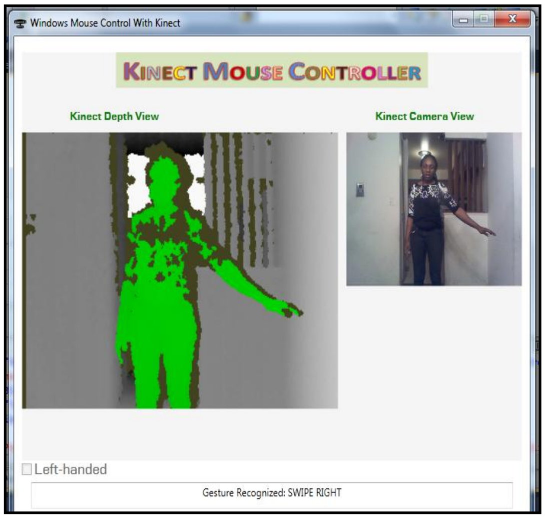
One of the application areas of the gesture-based HCI interface developed in this paper is in presentation control. The simulation of the right-arrow and the left arrow of the keyboard allow the presentation to move forward or go backward. Using the tracked X,Y,Z position of the head, the right hand and the left hand, two predefined gestures (*Swipe_Left* and *Swipe_Right*) are used for this keyboard simulation.

The *Swipe_Left* gesture (Figure 16-a) is recognized once the left hand is diverged 45 cm from the head which moves the presentation one slide backward while the *Swipe_Right* gesture (Figure 16-b) which moves the presentation forward is recognized once the right hand is diverged 45 cm from the head [20].



Gesture Recognized: SWIPE LEFT

(a)



Gesture Recognized: SWIPE RIGHT

(b)

Figure 16: (a) Gesture for Moving a Presentation Backward, and (b) Gesture for Moving a Presentation Forward

5.0 Conclusion

Human movements (human motion analysis) in recent years has become a natural interface for HCI and has been the focus of recent researchers in modeling, analyzing and recognition of gestures.

Hand gesture recognition still remains a prominent research area in the field of HCI as it provides a more natural way of interaction between humans and machine thus building a richer bridge between machines and humans than primitive text user interfaces or even GUIs (graphical user interfaces), which still limit the majority of input to keyboard and mouse.

Gesture recognition especially hand gestures is applicable over a wide spectrum of topics such as medicine, surveillance, robot control, teleconferencing, sign language recognition, facial gestures recognition, games and animation.

References

[1] Kurtenbach, G. & E.A. Hulteen. Gestures in Human-Computer Communication. In: The Art and Science of Interface Design, Laurel, B. (Ed.). Reading, Mass: Addison-Wesley Publishing Co., Wasley, pages 309-317, 1990.

[2] Kawade Sonam P & V.S. Ubale. Gesture Recognition - A Review. In OSR Journal of Electronics and Communication Engineering (IOSR-JECE), pages 19-26.

[3] Zimmerman, T., Lanier, J., Blanchard, C., Bryson, S. and Harvil, Y. A Hand Gesture Interface Device. In Proceedings of CHI 87 and GI, pages 189-192, 1987.

[4] Dipietro, L., Sabatini, A. M., & Dario, P. Survey of glove-based systems and their applications. *IEEE Transactions on systems, Man and Cybernetics, Part C: Applications and reviews, 38*(4), pages 461-482, 2008.

[5] [Untitled photograph of Data-glove]. Retrieved March 10, 2013, from: http://www.digitalrune.com/Support/Blog/tabid/719/EntryId/100/Scattered-Interpolation-ndash-Example-2-Interpolation-of-Animation.aspx

[6] Matthew Turk. Gesture Recognition. In the Handbook of Virtual Environment Technology, Chapter 10, 2012.

[7] Porta, M. Vision-based user interfaces: Methods and applications. *Elsevier, International Journal Human-Computer Studies, 2002*(57), pages 27-73, 2002.

[8] Rafiqul Zaman Khan & Noor Adnan Ibraheem. Hand Gesture Recognition – A Literature Review. In International Journal of Artificial Intelligence & Applications (IJAIA), Vol.3, No.4, July 2012.

[9] [Untitled photograph of Kinect]. Retrieved March 10, 2013, from: http://buildsmartrobots.ning.com/profiles/blogs/building-a-kinect-based-robot-for-under-500-00

[10] Mark Theodore Draelos. The Kinect Up Close: Modifications for Short-Range Depth Imaging, pages 11 -15, 2012.

[11] About Kinect. Retrieved from http://stackoverflow.com/tags/kinect/info.

[12] David Katuhe. Programming with the Kinect for Windows Software Development Kit, page 3, 2012.

[13] "Kinect develop overview," Microsoft, 2012, system requirements for Kinect for Windows SDK. Retrieved from: http://www.microsoft.com/en-us/kinectforwindows/develop/overview.aspx

[14] Zhou Ren, Junsong Yuan & Zhengyou Zhang. Robust Hand Gesture Recognition Based on Finger-Earth Mover's Distance with a Commodity Depth Camera, 2012.

[15] Daniel James Ryan. Finger and gesture recognition with Microsoft Kinect, (n.d).

[16] Murphy Stein (n.d.) Retrieved From http://makematics.com/code/FingerTracker/

[17] Rob Miles. Start Here! Learn the Kinect API, 1st Edition, 2012.

[18] [Untitled Photograph of Kinect Sensors on Human Body] Retrieved March 10, 2013 from: http://gmv.cast.uark.edu/uncategorized/working-with-data-from-the-kinect/attachment/kinect-sensors-on-human-body/.

[19] Samet Erap. Gesture-Based PC Interface With Kinect Sensor, 2012.