Finger Tracking and Gesture Interfacing
Using the Nintendo® Wiimote

By:
Ross Williams

Advisor:
Dr. Michael Branton

ABSTRACT:
Nintendo’s Wii-Mote is a revolution in gaming technology, being able to detect motion as it is moved around as well as infrared sensors linking it to the Wii console to show what the user is pointing at. Other gaming enthusiasts have been able to use the Wii-Mote for more than just a controller for the Wii by connecting it to a computer through its Bluetooth connectivity and turning it into a receiver for infrared signals to create “finger-tracking”. This new type of gesture interface could prove to be a very useful tool, not only for gaming but for anyone who wants to make a presentation more interactive rather than using the traditional mouse and keyboard.

A Senior research paper submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in the Department of Mathematics and Computer Science in the College of Arts and Sciences at Stetson University DeLand, Florida. Fall Semester 2009.
# TABLE OF CONTENTS:

1. Introduction.................................................................................................................. 3

2. Related Work.................................................................................................................. 3

3. C# and OpenGL............................................................................................................. 4
   1. Importing OpenGL with CsGL................................................................................. 5
   2. Drawing OpenGL Graphics on the Main Program Form.................................... 5

4. Making the Gloves...................................................................................................... 7

5. Connecting to and Managing the Wiimotes............................................................... 8
   1. The Wiimote Managed Library.............................................................................. 8
   2. Managing the Infrared Signals.............................................................................. 10
   3. Using the Infrared Signals to Move a Grid............................................................. 11
   4. Using the Infrared Signals to Zoom a Grid In and Out........................................ 13

6. Using Gestures to Control the Cursor in Windows XP............................................... 15
   1. Moving the Cursor and Smoothing it’s Movement.............................................. 16
   2. Gesture for a Mouse Click.................................................................................... 19
   3. Trying Multiple Cursors...................................................................................... 21

7. 3D Finger Tracking..................................................................................................... 22
   1. Zooming with the Grid using One Hand.............................................................. 22
   2. Three Dimensional Rotation with New Gloves.................................................. 24

8. Interactive Picture Gallery........................................................................................... 26

9. Future Work.................................................................................................................. 27

10. References................................................................................................................... 29
1. Introduction

This project is influenced by the work of Johnny Lee, with his implementation of a gesture controlled interface displayed on a screen showing a 3D wireframe grid that moved, rotated, and resized with the movements of his two index fingers using a Nintendo Wiimote and infrared light [1]. The drawback was that he could only be a certain distance away from the screen, since the transmission of infrared light from his fingers to the wiimote was coming from reflective tape on the ends of his fingers that reflected light shining toward him from an infrared LED array on top of the screen. A similar gesture interface was shown in a movie, Minority Report, which used gloves to control a very large interactive display in a similar way [13].

2. Related Work

Other than Johnny Lee’s implementation, there have been various other interfaces which use hand gestures as the main form of input. The general categories of these types of interfaces are: ones that use touch screens such as tablet PCs and iPhones, ones that use complicated algorithms to analyze a human figure and recognize a human hand to gather input from [2,3,4], and those that use hardware on the hand itself [5,6,7].

For touch screens, a great example of a gesture interface is how the iPhone has the ability to recognize two different points on its screen being touched at once. This allows the user to zoom in on an image by placing two fingers on the screen and separating them or zoom out by moving them closer together. The user can also rotate an image by twisting their hand, rotating the two points being touched around a fixed point, which the iPhone recognizes and then makes the proper adjustments to the image’s rotation.
Other interfaces will use algorithms which analyze a live video feed of the user, which determine the users position by recognizing the human body, then concentrate on the motions of the user’s hand [2,3,4]. This is a very convenient interface for the user, since anyone could simply walk up and start using it without having to put on hardware. Unfortunately there is a very large amount of overhead when analyzing each frame, and there is a larger possibility for gestures from the user to be inaccurately distinguished.

Interfaces that use hardware such as gloves placed on the hands reduces the overhead since the user’s hands are detected immediately by the computer by the signal the hardware is emitting [5,6,7]. Two implementations of this kind are the ‘Atlas Gloves’ [9], and a platform called ‘G-Speak’ [10]. These implementations both use gloves and hand gestures to control an interface. The ‘G-Speak’ interface is extremely complex, using multiple screens and multiple sensors, making it extremely expensive, but most like the futuristic Minority Report interface.

With this project we hope to make an interface that is just as impressive, yet affordable with hardware more readily available to the general public, and discover new possibilities this technology might be able to offer.

3. C# and OpenGL

We started by making a similar program to Johnny Lee’s, but using a mouse as the input device before importing the wiimote libraries to make sure OpenGL would work effectively with C#, since Lee’s project used DirectX. The reason we use OpenGL instead of DirectX is simply a matter of having more experience using OpenGL and better understanding of how objects are positioned in 3D space rendered to the screen, as well as experience with texturing images to
objects. OpenGL doesn’t have a very easily accessible library for use with C#, so some research was done to get it working properly.

3.1 Importing OpenGL with CsGL

There are two ways generally used to implement OpenGL applications in C#: the Tao framework [14], and the CsGL library [15]. We chose to use the CsGL library since Neon Helium [12], a popular tutorial and lesson source for OpenGL, uses CsGL in its lessons making it easier to understand how OpenGL is used in C# since there are examples for most the techniques we planned to use in this project.

In order to import the CsGL library to start using OpenGL in C# we first had to copy all of the DLL files used for CsGL into the system32 file inside of windows, then import them directly into our project in Visual Studio. Once this is complete we can then start using OpenGL functions in our program after importing the CsGL namespace into the class we are using at the time by invoking: using CsGL.OpenGL;

3.2 Drawing OpenGL Graphics on the Main Program Form

When using C# in Visual Studio our application uses a window called a form to display our program. To show our OpenGL environment we make a class which implements the OpenGLControl class, then add it to the main form of the program so our 3D environment will be shown inside the window, but before we run the program we need initialize our OpenGL properties by making a function called InitGLContext which will initialize the properties of OpenGL we will use in our 3D environment. This is an override method, meaning that it needs to be implemented by the programmer before our class will be able to implement OpenGLContol.
protected override void InitGLContext()
{
    GL.glShadeModel(GL.GL_SMOOTH); // enable smooth shading
    GL.glClearColor(0.0f, 0.0f, 0.0f, 0.5f); // black background
    GL.glClearDepth(1.0f); // depth buffer setup
    GL.glEnable(GL.GL_DEPTH_TEST); // enables depth testing
    GL.glDepthFunc(GL.GL_LEQUAL); // type of depth testing
}

Then we also need to implement another override function called glDraw, which is where all of our graphics are drawn inside our 3D environment.

public override void glDraw()
{
    // Clear Screen and Depth Buffer
    GL.glClearColor(GL.GL_COLOR_BUFFER_BIT | GL.GL_DEPTH_BUFFER_BIT);

    // reset the current modelview matrix
    GLMatrixMode(GL.GL_MODELVIEW);

    // defines the texture
    GL.glLoadIdentity();

    /*
     * OpenGL Drawing functions
     */

}  

Once OpenGL was up and running, we move on to write the imitation of Johnny Lee’s grid program. Starting out, we don’t use the wiimote, but use the mouse pointer to represent one infrared dot to move the grid around on the screen by clicking and holding the mouse button down. Our program’s main class is called ControllerGL, which we add a mouse handler to in order to receive input from the mouse’s position and if it is being clicked. After drawing two grids on the form, one lying on the XY axis and another lying on the XZ axis(for perspective), we added mouse event handlers to change the grid’s position depending on the change in the position of the mouse pointer after it was clicked and being held down. To control how the object
moves in and out, we used the scroll wheel for the time being. So now we are able to zoom in and out while moving up, down, left and right. After this basic motion is working smoothly, we move on to getting the WiiMote sensor and infrared dots to control these basic motions instead.

4. Making the Gloves

First we need to construct hardware to be placed on the hands with an infrared bulb that has its own power source. The idea was to use a glove, with the bulb and power source attached by some means. To start out with testing the gloves, bulbs were placed on both index fingers of both hands, with the power source attached above the wrist with the wires running along the finger as shown below:

For the purpose of the first demo program, only one infrared bulb needed to be put on each hand. One of the signals will control the position of the grid on the X and Y axis, and the
second signal received will control the zoom of the grid (Z axis along with scaling) by using an equation that will be explained in another section of this paper.

5. Connecting to and Managing the Wiimotes

The computer used for this project has Bluetooth capability which is the technology used for the Nintendo wiimote to connect to the Wii console, allowing us to connect to the wiimote with the computer using software called BlueSoleil [16]. BlueSoleil is an effective Bluetooth device manager which allows us to connect to all wiimotes in range that are waiting to be synced. Once they are connected to our computer through Bluetooth, we can start to use them with our demo program.

5.1 The Wiimote Managed Library

The wiimotes are controlled in our program by using the wii-mote managed library developed by Brian Peek [11]. This library allows us to use most of the functions the wiimote can provide, including the infrared sensor on top of each wiimote.

In our code we import the wiimote managed librarby source files into our visual studio project. The library has a class called WiimoteCollection, which connects to all available wiimotes in the computers HID list connected through BlueSoleil. We create an instance of WiimoteCollection and run one of its main functions known as FindAllWiimotes(), which stores each found wiimote into an array within the WiimoteCollection class. This should be done within a try/catch function, as not to crash the program with a “wiimote not found error”.

Once all of our connected wiimotes are collected into our instance of the WiimoteCollection class, we can then create our own instances of individual wiimotes using the
Wiimote class. After declaring one or more wiimote variables, we assign each one in order from the array within the WiimoteCollection instance. For now we only use one wiimote, placed underneath a projection screen showing our computer’s display, faced toward the user to receive infrared signals. The following code shows the connection process for a wiimote:

```csharp
//CREATE VARIABLES
WiimoteCollection wiiCollector;
Wiimote remote1;

//FIND WIIMOTES
try{
    wiiCollector = new WiimoteCollection();
    wiiCollector.FindAllWiimotes();
} catch (Exception x){
    MessageBox.Show("Cannot find a wii remote: "+x.Message);
}

//CONNECT TO FIRST WIIMOTE
try{
    //Assign remote1 to the first wiimote found from collection
    remote1 = wiiCollector[0];

    /*Activate the wiimote and enable the functions to be used (infrared in this case) and set the first LED on the wiimote on so we know which wiimote the program is calling remote1 */
    remote1.Connect();
    remote1.SetReportType(InputReport.IRAccel, true);
    remote1.WiimoteState.Battery.ToString();
    remote1.SetLEDs(true, false, false, false);
    remote1.WiimoteChanged += ControllerGL_OnWiimoteChanged;
} catch (Exception x){
    MessageBox.Show("Cannot connect to remote1: "+x.Message);
}
```

In the bottom half of this code we construct the wiimote object, connect to it, set it up to report infrared signatures, check the battery, set the LED on the wiimote to the index of the wiimote we are using in case of multiple wiimotes, and assign the event handler to detect changes in the wiimote’s state, in this case being infrared sensor changes.
The code segment:

```csharp
remote1.WiimoteChanged += ControllerGL_OnWiimoteChanged;
```

is what tells our program that we have movement on remote1’s infrared sensor, allowing us to read in the new information from remote1 and use it to manipulate our program accordingly by calling the function ControllerGL_OnWiimoteChanged.

### 5.2 Managing the Infrared Signals

We’ve created a class called PointManager, which holds all of the information for the infrared signatures found by one wiimote, and then manipulates the grid’s position and scaling accordingly. The construction of the PointManager class requires being passed the wiimote it is to watch along with the grid it is to control. The following code is in the ControllerGL class as well:

```csharp
PointManager pointManager1;
pointManager1 = new PointManager(remote1, grid);
```

The declaration of the pointManager1 is right after declaring the other variables, so it is used globally within the class, and the assignment takes place right after connecting to and setting up remote1. The PointManager class has a method called ‘update()’ which is executed in the ControllerGL_OnWiimoteChanged function, so all information is updated when the wiimote experiences a change.

We’ve created a class called WiiCursor to be used within the PointManager class, which holds positioning and other data for a single infrared signature that the wiimote finds, including whether or not it is active, not active, or was active. When an instance of WiiCursor is created, it needs to be passed an integer value from 1 through 4, each representing a different color that
cursor will be when it is drawn to the screen, so we will not be confused when we see multiple cursors.

After constructing the instance of up to four WiiCursors and passing each the index value for which cursor color it is (1 through 4), we can then parse the wiimote data.

```java
WiiCursor cursor1 = new WiiCursor(1);
```

Our instance of the PointManager class, pointManager1, uses our previously created instance of the Wiimote class called remote1 to give our wiiCursors the proper values using an array of 2D points called wiimotepointsNormalized. The wiimote class has a property called WiimoteState, which is the property the wiimote event handler monitors for changes. WiimoteState has a property called IRState, which contains all the data for current infrared activity. IRState has an array of four IRSensors properties, which contain the various data that will be analyzed by the program, such as position, intensity, etc.

### 5.3 Using the Infrared Signals to Move a Grid

When parsing the data, we first determine if the point is detected, and if it is, we assign our wiimotepointsNormalized array the proper value by modifying the IRSensors position property to be usable with our OpenGL grid, and setting a Boolean variable representing that the cursor is down (detected) to true. The IRSensors position is represented by a value between 0 and 1 for both the X and Y axis, while the cursors showing on the screen will have position values from around -15 to 15. This is why we manipulate the value assigned to wiimotepointsNormalized by multiplying it by a value, in this case ‘disposition’ which is usually assigned 30, then subtracting 15 from that value resulting in an appropriate value to use in
drawing the cursor. Also the X axis must be flipped since the sensor is facing the opposite direction. This is easily solved by subtracting the IR position on the X axis by 1.

```csharp
//Find first infrared signal and turn it into a cursor
if (remote.WiimoteState.IRState.IRSensors[0].Found){

    //Get infrared positioning info and scale it to the proper size
    wiimotePointsNormalized[0].x =
        (1.0f - remote.WiimoteState.IRState.IRSensors[0].Position.X) *
        disposition - 15f;
    wiimotePointsNormalized[0].y =
        (remote.WiimoteState.IRState.IRSensors[0].Position.Y) *
        disposition - 15f;

    //Set cursor as active
    cursor1.isDown = true;
}
else{
    cursor1.isDown = false;
    cursor1.wasDown = false;
}

//Update cursor 1 info
if (cursor1.isDown)
{
    //Save previous position
    originalX = cursor1.X;
    originalY = cursor1.Y;

    //Update cursor’s screen position and draw it to the screen
    cursor1.setDown(wiimotePointsNormalized[0].x,
        wiimotePointsNormalized[0].y);
    cursor1.display();

    //Move the grid with the cursor
    if (cursor1.wasDown){
        grid.xPos += (wiimotePointsNormalized[0].x - originalX);
        grid.yPos += (wiimotePointsNormalized[0].y - originalY);
    }
    cursor1.wasDown = true;
}
```

If the cursor is down, we display the cursor in the program as a small colored sphere by calling the WiiCursor’s display function, and add to or subtract from the grid’s current position depending on the changes made to the cursor’s position since it was first found. Now we have simple 2D motion with one infrared signature.
5.4 Using the Infrared Signals to Zoom a Grid In and Out

At this point, the first infrared signal detected by remote1 becomes the first WiiCursor named cursor1, which acts as the cursor that moves the grid in the X and Y direction. Now we need to make another instance of the wiiCursor class and attempt to have two wiiCursors interact, so we construct a differently named instance of wiiCursor, cursor2, passing it the index of 2. When parsing the wiimote data this time, if one signal is found, cursor1 is assigned its information allowing it to move the grid, and if a second point is found after that, cursor2 is assigned its information so that we may create a zooming effect using the interaction of both cursors. An example of this zooming technique is used by the iPhone when viewing a webpage. The user can zoom in and out by using two fingers placed on the screen and separate them to zoom in or move them closer together to zoom out. With an infrared signal coming from each hand of the user in this project, we attempt to accomplish a similar effect by zooming in when the distance between the two signals increases, or the hands moving apart from each other, and zooming out when they get closer together.
The next step is to manipulate the grid depending on this information. Before we were checking to see if only the first cursor is down, and move the grid if it is. Previously we had one ‘if’ statement to check if one cursor was down, and move the grid if it is. Now we will have two statements: one if cursor1 is down and a second right afterwards if both cursor1 and cursor2 is down. If both cursor1 and cursor2 is down then we begin the zooming effect.

In OpenGL, there are two ways to accomplish this: adding to and subtracting from the Z position of the grid which will move it closer to or further away from the viewpoint, or changing the scale of the grid itself making it larger or smaller. In this case we will use the scaling option since there’s a limit to how close the grid can move to the viewpoint by changing its Z position before it becomes invisible.

```java
if (cursor1.isDown && cursor2.isDown){
    //find distance between the two cursors
    float xDis = wiimotePointsNormalized[1].x – wiimotePointsNormalized[0].x;
    float yDis = wiimotePointsNormalized[1].y – wiimotePointsNormalized[0].y;

    //set the original if cursor2 has just been set down
    if(originalDistance == null)
        originalDistance = (float)Math.Sqrt(xDis * xDis + yDis * yDis);

    //get current distance between the points
    currentDistance = (float)Math.Sqrt(xDis * xDis + yDis * yDis);

    //change the scale
    grid.scale = ((currentDistance) / (originalDistance));

    cursor1.wasDown = true;
    cursor2.wasDown = true;
} else{
    originalDistance = null;
}
```

We’ve now accomplished 6 simple motions: up, down, left, right, back, and forth by using the motions of the user’s hands. These are the motions we decided to begin with since the
general use of an interface today is by using a mouse which is moved up, down, left, and right. The motions back and forth are used in this demo to show how two points can interact. These are typically the first motions implemented in the gesture interfaces we’ve previously discussed, and are also the basic motions used in Johnny Lee’s demo [1].

6. Using Gestures to Control the Cursor in Windows XP

So far this project has used the wiimote and gloves with infrared emitters and an OpenGL program to demonstrate simple motion with gestures, now we use this technology to create something practical for the user, by using the gloves and wiimote to control the mouse pointer inside of an operating system, in this case Windows XP.

The first assumption was to begin looking at mouse drivers and find one that was open source for XP that we might be able to learn from and apply to our program. Luckily, since this program is written in C#, we have access to the Windows.System.Forms library which controls the Cursor class which is the actual mouse pointer used in windows. This Cursor class can be accessed in any class or form in the program, as long as the class or form is including Windows.System.Forms. To change the position we cannot change the x and y value separately,
but would need to make a new point and assign Cursor.Position the value, which would in turn move the cursor to the new position anywhere on the screen within windows.

6.1 Moving the Cursor and Smoothing its Movement

The position of Cursor is actually the value the cursor is resting on in the screen’s resolution, so if we have a screen resolution of 1024x768, the value of X ranges from 0 (the left) to 1024 while Y ranges from 0 (the bottom) to 768. In this case we no longer need OpenGL, since our objective is to use our gloves with the wiimote to control the mouse itself over the entire desktop and not within a graphical application. So now when we detect a point, we need to adjust the values to make a new point that’s on the same scale as the position values the mouse cursor uses while the transition between current point and the new point is visually equivalent to what the user is used to, as in using a real mouse. This is a lengthy and complicated process that not only uses the IRSensor’s position converted to the correct scale, but makes the cursor smoothly glide across the screen and cover more distance the faster it’s moved, just like a real computer mouse on a flat surface.

When changing the position of the cursor, we simply keep our own global point variable called ‘cursorPos’ which is the point that the windows cursor is assigned whenever our program detects an infrared signal. Whenever we wish to move the cursor inside of windows we add or subtract small values from cursorPos’s X and Y values which is then made the new position of the windows cursor each frame. In the grid program we did this by adding the difference of the current position and last position of the cursor to the grid’s position, but we now want to improve this method and make it smoother.
The way we do this is to create a statically sized array of points, which tracks the position of the infrared signal into each slot of the array from the first slot to the last repeatedly and averages the points each time a new point is added, then subtracting that average from the current position of the infrared signal instead of only its previous position. This is happening extremely fast within the program, but to the user it gives the effect of moving slowly as the cursor begins to move, then accelerates depending on how far the signal is moved, and then decelerates when the user has stopped moving so the cursor does not fly past the intended destination.

For the size of the array of points to be averaged, we experimentally decided on the value of 20. The larger the size of the array, the slower the program becomes, since the average will take longer to calculate for each frame. However, if the value is too small, the movement will not appear to be as smooth. When the infrared signature is first detected we initialize all of the points in the array to the cursor’s current position. After that we begin looping from 0 to 19, recording each position the signal is at, then subtract the average from cursor1’s current position. Keep in mind that these values of the points for the infrared signals are between 0 and 1, so difference will be a very small number. Since the X and Y values for cursorPos are integers between 0 and the resolution size, we need to scale this difference to be large enough to be added to the new position of the cursor. We do this by multiplying the X and Y values of the difference of the current position and the average by the respective resolution size, which we call screenOffsetX and screenOffsetY. For a 1024x768 size screen these values are set to 1024 and 768 respectively.

This method smoothes the cursor’s movement as it starts and stops, but we also want to cover more distance with the cursor when the signal is moving fast. This happens with a normal mouse, for instance when a user moves a mouse slowly across a desk the mouse doesn’t cover as
much distance across the screen compared to if the mouse was moved quickly along the desk. To add this type of acceleration to the cursor in our program we also add another value to the position of the cursor which is a fraction of the distance between the current and previous position of the infrared signal, scaled to screen size as well. This covers further distance of the cursor the faster the infrared signal is moved.

Here is the code for when a single infrared signal is picked up:

```java
//We call this function when we receive a signal
public void updateCursorPos()
{
    //Check to see if the signal has already been found
    if (!cursor1.wasDown) {
        //Set our own cursor point to the windows cursor position
        cursorPos.X = Cursor.Position.X;

        //initialize our array for smoothing
        initPoints();
    } else {
        //add current infrared point to smoothing array
        updatePoints();

        //set the amount of acceleration to be used
        float cursorSpeed = 1/5f;

        //find the values x and y to add to the windows cursor’s position
        cursorSpeedX = (cursor1.X - getAveX()) * screenOffsetX +
                       (cursor1.X - cursor1.lastX) * cursorSpeed * screenOffsetX;
        cursorSpeedY = (cursor1.Y - getAveY()) * screenOffsetY +
                       (cursor1.Y - cursor1.lastY) * cursorSpeed * screenOffsetY;

        //convert the values to int
        int tempx = (int)cursorSpeedX;
        int tempy = (int)cursorSpeedY;

        //check that the new value will be within the screen bounds
        if (((tempx + cursorPos.X) >= 0 &&
             (tempx + cursorPos.X) <= screenOffsetX) &&
            ((tempy + cursorPos.Y) >= 0 &&
             (tempy + cursorPos.Y) <= screenOffsetY))
        {
```
6.2 The Gesture for a Mouse Click

Next we create a gesture that can be made with the gloves to simulate left and right mouse clicks. We modified the glove to be used for this application to have one infrared bulb on top of the hand right above the knuckle at the base of the pinky finger, and another at the tip of the index finger. If the user points a closed fist at the wiimote with the glove, the sensor will only detect one signal. If the user points their index finger at the wiimote the sensor will then detect two signals.

When the user points a fist at the wiimote, the program will detect one infrared signal via the wiimote and begin to move the mouse cursor as the user moves their fist. When the user raises their index finger and a second signal is detected, it will be interpreted by the program as a mouse click. This is an example of a gesture controlling our interface. We have two types of mouse clicks in windows, left and right, so we need two different gestures to accomplish this. When the second signal is detected, the program will determine the difference between the X
position values of the two signals, and the Y position values of the two signals. If the difference of the X values is greater than the difference of the Y values the program will perform a left click, and if the opposite is true then the program will perform a right click. This means that if the user is pointing their fist palm down at the wiimote and raises their index finger then the difference in Y values will be closer together than the difference in X values, signaling a left click. If the user turns their fist sideways and does the same gesture, the opposite will be true and signal a right click.

The gesture of flicking the index finger outward and back in is equivalent in this case to pressing a mouse button down and releasing it immediately, while keeping the index finger extended is equivalent to holding the button down. Because of this we need two functions, ‘clickCursorDown’ when the second signal is received and ‘clickCursorUp’ when the signal is lost.

```java
private const int MOUSEEVENTF_LEFTDOWN = 0x02;
private const int MOUSEEVENTF_LEFTUP = 0x04;
private const int MOUSEEVENTF_RIGHTDOWN = 0x08;
private const int MOUSEEVENTF_RIGHTUP = 0x10;

public void clickCursor1Down(){
    xDiff = (remote.WiimoteState.IRState.IRSensors[0].Position.X - remote.WiimoteState.IRState.IRSensors[1].Position.X);

    if (xDiff > yDiff) {
        mouse_event(MOUSEEVENTF_LEFTDOWN,
                    cursorPos.X, cursorPos.Y, 0, 0);
        leftClicked = true;
    }
    else{
        mouse_event(MOUSEEVENTF_RIGHTDOWN,
                    cursorPos.X, cursorPos.Y, 0, 0);
        rightClicked = true;
    }
}
```
public void clickCursor1up()
{
    if (leftClicked) {
        mouse_event(MOUSEEVENTF_LEFTUP, cursorPos.X, cursorPos.Y, 0, 0);
        leftClicked = false;
    }
    if (rightClicked) {
        mouse_event(MOUSEEVENTF_RIGHTUP, cursorPos.X, cursorPos.Y, 0, 0);
        rightClicked = false;
    }
}

As shown above, the clickCursorDown function checks to see if the second signal is representing a left or right mouse click, then sets a Boolean value to be used by clickCursorUp later once the signal is lost. The windows Cursor class automatically clocks the time difference between when a button is pressed and released to determine if the cursor should start to drag or select the object the cursor is placed over, along with determining double clicks, so this did not need to be implemented in our program.

6.3 Trying Multiple Cursors

Windows XP does not support multiple cursors, but we were curious to see the effect of controlling two cursors at one time, one with the right hand of the user and another with the left hand. Using the mouse events in the functions shown above, we can simulate a mouse click anywhere on the screen, even if the cursor is somewhere else, by passing the mouse event a different position than the cursor. Unfortunately the cursor in windows is drawn to the screen deep within the system functions of windows and can only be called at one time and can’t be replicated. To substitute for this we try to draw our own, but unfortunately are limited to draw within the displayed form of the program we are running, and not on top of the operating system itself. There is a way around this by drawing an object using the class that draws the windows
operating system to the screen, using another property inside of windows.System.Forms called
ControlPaint, but this slows the system down and leaves unwanted remnants left over from the
objects being drawn especially if they are moving about the screen. When we tested having a
second cursor, it became very difficult to keep track of what was happening, not within the
program, but with what the user was trying to accomplish by having two cursors since keeping
track of where one cursor is moving seems to be difficult enough.

7. 3D Finger Tracking

So far we’ve only shown how finger tracking can be accomplished with one wiimote,
using 2D positions of detected signals to control our programs. Previously we’ve mentioned that
the wiimote library has a WiimoteCollection class which finds not just one wiimote but all
wiimotes connected to the computer. The idea for this part of the project is to use not just one
wiimote to detect infrared signals, but to use a second wiimote facing the user from a different
direction, therefore being able to detect infrared signals for finger tracking in three dimensions
instead of two.

7.1 Zooming with the Grid using One Hand

We will still keep our first wiimote positioned underneath the display facing the user,
then position a second wiimote about the same distance away directly to the left side of the user.
We use the OpenGL grid program we created earlier to make a demonstration of simple 3D
finger tracking. Instead of having two gloves with an infrared bulb on each, we use one glove
with two bulbs, one facing forward to be picked up by the first wiimote positioned under the
screen, and the second facing directly to the left to be picked up by the second wiimote. The first
wiimote will receive its signal and move the grid up, down, left, and right just as before, moving
in two dimensions with the glove of the user. The second wiimote will then control the zooming
effect of the grid by watching it’s signal move toward and away from the user. To the user this motion will be bringing their hand closer to their body or pushing it away from them, but the second wiimote will see this as movement along its sensor’s X axis. So we don’t confuse the different responsibilities for each wiimote, we create two different PointManager classes, each of which are able to change properties of the same grid, but control different wiimotes. We keep the name PointManager for the first but allow it only to check if one signal is active and move the grid accordingly on the X and Y axis. We then create a new class ‘PointManager2’ which controls the second wiimote which also only checks for one signal and it’s changes on the X axis, then adjusts zooming effect accordingly. The code for the first point manager and 2D movement was shown earlier in this paper, so now we show the code for the PointManager2 receiving a signal:

```java
if (cursor1.isDown){
    //get current x position
    currentX = wiimotePointsNormalized[0].x;

    //check to see if already was found
    if (cursor1.wasDown){
        //manipulate the scale
        if (grid.scale > 0)
            grid.scale += (currentX - originalX) / 5;
        else
            grid.scale = 0.1f;

        //manipulate Z position
        grid.zPos += 2*(currentX - originalX);
    } //set the previous value to use in next frame’s calculation
    originalX = currentX;
}
```

Instead of only changing the scale as we did with the previous grid program, we chose to use a combination of altering the Z position and the scale at the same time. The difference in this case is that the program adjusts the zoom depending on one point moving across a straight line,
rather than the difference in the distance between two points on a 2D plane. This required a more in-depth approach to creating a zooming effect, and made it more sensitive to the user’s movements. This produces a simple demonstration of 3D finger tracking, allowing the user to pull the grid toward them or push it away while also moving it along the X and Y axis all with one hand.

7.2 Three Dimensional Rotation with New Gloves

We created another demonstration to show the possibility of 3D tracking, in this case using rotation of the grid. With one wiimote and two gloves we can accomplish rotation on the Z axis according to the difference of the current Y values and original Y values of the two signals being received, so if the user raises one hand and lowers the other then the grid will rotate accordingly.

With 3D tracking, we can also use this concept to rotate on the Y axis, using the the second wiimote and rotate the grid according to the difference of the X values. In order to accomplish this we needed to create two new gloves, each one with an infrared bulb pointing forward and another pointing toward the second wiimote, which in this case is to the left.
The new gloves shown above do not use a power switch to turn on, since that requires using your other hand, so instead we’ve put a metal connector on the index finger and thumb of each glove that when they touch together it connects the hot wire and activates the infrared bulbs. This is convenient since you can now turn the glove on and off using the hand it is on, avoiding unwanted infrared signals from being sent to the wiimotes during the motion of turning off a switch on the back of the hand.

When we run this demonstration, the first wiimote controls the Z axis rotation and the second wiimote controls the Y axis rotation, therefore we are able to rotate the grid in three dimensions by making a peddling motion with the gloves.
8. Interactive Picture Gallery

We created another application to use 3D tracking in a more practical way by making an interactive picture gallery in OpenGL. This program is very similar to the grid programs shown so far, except we have 9 objects being displayed, each able to be moved and enlarged separately.

We already have a cursor that is shown in the environment, but needed to implement a way to be able to select the individual squares. We accomplish this by using the same gesture used with the windows cursor program: by extending the index finger when the cursor is over the object the user wants to select. Once the program recognizes an object is being selected, the user will be able to move the object and set it down by bring their index finger back again. Once these basic functions were working we moved on to texturing the images.
Using OpenGL we are able to apply jpeg images to our rectangles. Currently we are implementing the same zooming effect used with the grid program and 3D tracking so the user can bring the image toward them on the screen by pulling their hand toward their body, along with a new gesture to rotate the image so the back faces front with some text revealing information about the picture. After the user selects an image by extending the index finger, if the user rotates their hand in this position from palm down to sideways, this motion will register to the program as the gesture to flip the image and reveal the text printed on the back.

Gestures like these can control a wide range of functions within programs used every day.

9. Future Work

It should be noted that the programs created in this project can be greatly enhanced with fine tuning, unfortunately it takes a great deal of trial and error with this type of technology.
There is always the changing factor of where the user is in accordance with the position of the
wiimote, which creates infrared sensitivity issues, two infrared signatures that are close enough
together but are far away from the sensor can appear to be only one signature to the wiimote.
Many other possible hand gestures can be created to use with the program for invoking endless
combinations of commands, and with those will come new designs of gloves and different ways
the infrared bulbs can be attached to them. This type of technology itself is in a way only a
demonstration of how interactive our bodies can be when it comes to controlling computers, and
opens our minds for what can be done in the future.
10. References:

1. “Tracking Fingers with the Wii-mote”.  
   http://www.cs.cmu.edu/~johnny/projects/wii/

2. “A Virtual 3D Blackboard: 3D Finger Tracking using a Single Camera”.  

3. Real-time Hand Tracking and Finger Tracking for Interaction  

4. Finger Track - A Robust and Real-Time Gesture Interface  

5. “Inverse Kinematic Infrared Optical Finger Tracking”.  

6. “Finger tracking for interaction in augmented environments”.  
   http://www.ims.tuwien.ac.at/media/documents/publications/FingerTracker.pdf

7. “Portable Absolute Position Tracking System For Human Hand Fingertips”.  
   http://www.robotcub.org/misc/review3/06_Bezdiece_Caldwell.pdf


   http://atlasgloves.org/about


11. “Managed Library for Nintendo’s Wii-mote”.  

12. “NeHe OpenGL Lessons and Examples”.  
    http://nehe.gamedev.net/lesson.asp?index=01

13. “Interactive User Interface in Minority Report”.  
    http://www.youtube.com/watch?v=NwVBzx0LMNQ

