GESTURE RECOGNITION BASED CALCULATOR

SREE BHARAT DASARI (PSU ID: 938833928)

The Gesture Recognition Based Calculator is a Kinect implemented project that works on the principle of Dynamic Time Warping. The main idea of developing this project was to develop a project that uses a simple Random function to create a series of random numbers and use a similar kind of random function to generate another series of random numbers and perform mathematical operations between the two numbers on the left hand side and display the result. The user has to guess the operator between the two numbers that caused the result to be displayed on the right hand side of the “=” operator.

This project was submitted as a fulfilment for a Single credit work for Spring 2013 term.

CONTENTS

DYNAMIC TIME WARPING

Interacting with computers using human motion is commonly employed in Human-Computer Interaction (HCI) applications. One way to incorporate human motion into HCI applications is to use a predefined set of human joint motions i.e., gestures. A variety of methods have been proposed for gesture recognition, ranging from the use of Dynamic Time Warping to Hidden Markov Models. DTW measures similarity between two time sequences which might be obtained by sampling a source with varying sampling rates or by recording the same phenomenon occurring with varying speeds.

Dynamic time warping (DTW) is an algorithm for measuring similarity between two sequences which may vary in time or speed. DTW has been applied to video, audio, and graphics — indeed, any data which can be turned into a linear representation can be analyzed with DTW. A well-known application has been automatic speech recognition, to cope with different speaking speeds. Other applications include speaker recognition and online signature recognition. Also it is seen that it can be used in partial shape matching application.

In general, DTW is a method that allows a computer to find an optimal match between two given sequences (e.g. time series) with certain restrictions. The sequences are "warped" non-linearly in the time dimension to determine a measure of their similarity independent of certain non-linear variations in the time dimension. This sequence alignment method is often used in time series classification.

The conventional DTW algorithm is basically a dynamic programming algorithm, which uses a recursive update of DTW cost by adding the distance between mapped elements of the two sequences at each recursion step. The distance between two elements is oftentimes the Euclidean distance, which gives equal weights to all dimensions of a sequence sample. However, depending on the problem a weighted distance might perform better in assessing the similarity between a test sequence and a reference sequence. For example in a typical gesture recognition problem, body joints used in a gesture can vary from gesture class to gesture class. Hence, not all joints are equally important in recognizing a gesture.

GESTURE RECOGNITION

Gesture recognition interprets human gestures via mathematical algorithms. Gestures can originate from any bodily motion or state but commonly originate from the face or hand. Many approaches have been made using cameras and computer vision algorithms to interpret sign language. However, the identification and recognition of posture, gait, proxemics, and human behaviors is also the subject of gesture recognition techniques. Gesture recognition can be seen as a way for computers to begin to understand human body language, 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 enables humans to communicate with the machine (HMI) and interact naturally without any mechanical devices. Using the concept of gesture recognition, it is possible to point a finger at the computer screen so that the cursor will move accordingly. This could potentially make conventional input devices such as mouse, keyboards and even touch-screens redundant. Gesture recognition can be conducted with techniques from computer vision and image processing.

FEATURES

Graphical User Interface:

This program has an informative graphical user interface. It is not highly advanced but just enough to tell you what is going on, and what you need to do to be able to interact with the Kinect. All you have to do is stand in front of the Kinect and once when you can see that your skeleton is being tracked, you can hit the calibrate button on the GUI after selecting the motion that you want the Kinect to calibrate your gesture. Once you are done calibrating the gestures, you can continue to form gestures so that the Kinect detects your gesture and gives the appropriate output accordingly. Please refer to the "Screen Shots" section below for the visual.

Steps to Follow:

- Load the project into the Visual studio 2010 c# express and see to it that all the references are also added to the project.
- Once the project is loaded and all the references added, try building the project. If all the references were properly added you should not be receiving any errors or warnings.
- Run the project and you should be able to see a GUI with Gesture Recognition Based Calculator as its title.
- For the first time, you have to calibrate the gestures for each specific function that you wish to perform.
- You can calibrate the gestures by selecting the function and hitting the calibrate button on the GUI.
- After you hit the calibrate button you will have 3 seconds to get back to your position where the Kinect can see you properly and your skeleton is detected on the Skeleton Viewer.

- Keep performing the gesture until you see “gesture recognized as” on the GUI and the status comes back to “reading”.
- Once all the gestures have been calibrated you can continue to answer the question that appears on the GUI by performing the corresponding gesture.
- You will hear sound indicating if your response was correct or wrong.
- The main advantage is that you need not remember what type of gesture you recorded for a specific type of function. You can always calibrate your own unique gestures and store them any time you run the program.

- ***WPF Viewers:***
- KinectColorViewer
- KinectDepthViewer
- KinectSensorChooser
- KinectSkeletonViewer

These WPF viewers are to be added as a reference to the project. These viewers are mainly responsible for the code-less job of displaying what the Kinect sees and makes our job easy. There are also some other types of WPF viewers such as the KinectAudioViewer which uses the Kinect’s audio recording capabilities and displays the data to the user. Together these viewers are to be added in the project for getting their respective views of the Kinect.

Coding4Fun Tool Kit:

This project also uses the coding4fun tool kit as a reference module. The main idea of the project was developed from the DTW gesture recognition, which also this tool kit. This tool kit also has some interesting libraries that can be added as a reference for other projects. In this project I also use a set of images that are saved within the project. Whenever the random function return a number, the number is matched with its corresponding image from the collection of images that I put in and that image is loaded onto the GUI.

There are a lot of project that you can find in the internet that you can do with a Kinect. Licensing is never an issue as Microsoft decided to make the Kinect an open source project where in anyone can use the source code and modify the parameter and the data extracted from the design to meet the specifications of their projects.

System Requirements:

- Windows 7 or higher
- Kinect SDK
- Visual Studio 2010 C# Express
- Coding4Fun toolkit
- A latest .NET

References:

I would like to extend my sincere thanks and respect to Rhemyst and Rymix for their awesome Kinect DTW Gesture Recognition which is the base of this project and also Ray Chambers, an outstanding teacher with his new innovative ideas in the field of Kinect.

Useful Links:

KinectDTW project on Codeplex:
http://kinectdtw.codeplex.com/
Microsoft's Kinect SDK terms of use:
http://research.microsoft.com/en-us/um/legal/kinectsdk-tou_noncommercial.htm

Wikipedia's reliable explanation of dynamic time warping:
http://en.wikipedia.org/wiki/Dynamictimewarping

A real scientific paper :
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.114.3782&rep=rep1&type=pdf
Kinect SDK form on MSDN:
http://social.msdn.microsoft.com/Forums/en-US/category/kinectsdk
Microsoft Coding4Fun Toolkit
http://c4fkinect.codeplex.com/

If you are new to Kinect, this link will really help you to get things started

http://channel9.msdn.com/Series/KinectQuickstart/

Last but not the least, the most important one of all,

https://www.google.com/ ;)

SOURCE CODE

The MainWindow.xaml.cs file:

```
//-----------------------------------------------------------------------
// <copyright file="MainWindow.xaml.cs" company="Rhemyst and Rymix">
//     Open Source. Do with this as you will. Include this statement or
//     don't - whatever you like.
//
//     No warranty or support given. No guarantees this will work or meet
//     your needs. Some elements of this project have been tailored to
//     the authors' needs and therefore don't necessarily follow best
//     practice. Subsequent releases of this project will (probably) not
//     be compatible with different versions, so whatever you do, don't
//     overwrite your implementation with any new releases of this
//     project!
//
//     Enjoy working with Kinect!
// </copyright>
// Gesture Recognition Based Calculator
// Modified by Sree Bharat Dasari
//-----------------------------------------------------------------------

namespace DTWGestureRecognition
{
    using System;
    using System.Media;
    using System.Collections;
    using System.Collections.Generic;
    using System.Diagnostics;
    using System.Windows;
    using System.Windows.Forms;
    using System.Windows.Media;
    using System.Windows.Media.Imaging;
    using System.Windows.Shapes;
    using System.Linq;
    using Microsoft.Kinect;

    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow
    {
        decimal number1questionconst;
        decimal number2questionconst;
        string number3operatorconst;
        decimal answertoquestion;
        Random random = new Random();
        // We want to control how depth data gets converted into false-color
data
        // for more intuitive visualization, so we keep 32-bit color frame
buffer versions of
        // these, to be updated whenever we receive and process a 16-bit
frame.
```

```
        /// <summary>
        /// The red index
        /// </summary>
        private const int RedIdx = 2;

        /// <summary>
        /// The green index
        /// </summary>
        private const int GreenIdx = 1;

        /// <summary>
        /// The blue index
        /// </summary>
        private const int BlueIdx = 0;

        /// <summary>
        /// How many skeleton frames to ignore (_flipFlop)
        /// 1 = capture every frame, 2 = capture every second frame etc.
        /// </summary>
        private const int Ignore = 2;

        /// <summary>
        /// How many skeleton frames to store in the _video buffer
        /// </summary>
        private const int BufferSize = 32;

        /// <summary>
        /// The minumum number of frames in the _video buffer before we
attempt to start matching gestures
        /// </summary>
        private const int MinimumFrames = 6;

        /// <summary>
        /// The minumum number of frames in the _video buffer before we
attempt to start matching gestures
        /// </summary>
        private const int CaptureCountdownSeconds = 3;

        /// <summary>
        /// Where we will save our gestures to. The app will append a
data/time and .txt to this string
        /// </summary>
        private const string GestureSaveFileLocation = @"C:\Users\sree\My
Documents\Gestures\";

        /// <summary>
        /// Where we will save our gestures to. The app will append a
data/time and .txt to this string
        /// </summary>
        private const string GestureSaveFileNamePrefix = @"RecordedGestures";

        /// <summary>
        /// Dictionary of all the joints Kinect SDK is capable of tracking.
You might not want always to use them all but they are included here for
thouroughness.
        /// </summary>
        private readonly Dictionary<JointType, Brush> _jointColors = new
Dictionary<JointType, Brush>
        {
            {JointType.HipCenter, new SolidColorBrush(Color.FromRgb(169, 176,
155))},
            {JointType.Spine, new SolidColorBrush(Color.FromRgb(169, 176,
155))},
            {JointType.ShoulderCenter, new SolidColorBrush(Color.FromRgb(168,
230, 29))},
            {JointType.Head, new SolidColorBrush(Color.FromRgb(200, 0, 0))},
            {JointType.ShoulderLeft, new SolidColorBrush(Color.FromRgb(79,
84, 33))},
            {JointType.ElbowLeft, new SolidColorBrush(Color.FromRgb(84, 33,
42))},
            {JointType.WristLeft, new SolidColorBrush(Color.FromRgb(255, 126,
0))},
            {JointType.HandLeft, new SolidColorBrush(Color.FromRgb(215, 86,
0))},
            {JointType.ShoulderRight, new SolidColorBrush(Color.FromRgb(33,
79,  84))},
            {JointType.ElbowRight, new SolidColorBrush(Color.FromRgb(33, 33,
84))},
            {JointType.WristRight, new SolidColorBrush(Color.FromRgb(77, 109,
243))},
            {JointType.HandRight, new SolidColorBrush(Color.FromRgb(37,  69,
243))},
            {JointType.HipLeft, new SolidColorBrush(Color.FromRgb(77, 109,
243))},
            {JointType.KneeLeft, new SolidColorBrush(Color.FromRgb(69, 33,
84))},
            {JointType.AnkleLeft, new SolidColorBrush(Color.FromRgb(229, 170,
122))},
            {JointType.FootLeft, new SolidColorBrush(Color.FromRgb(255, 126,
0))},
            {JointType.HipRight, new SolidColorBrush(Color.FromRgb(181, 165,
213))},
            {JointType.KneeRight, new SolidColorBrush(Color.FromRgb(71, 222,
76))},
            {JointType.AnkleRight, new SolidColorBrush(Color.FromRgb(245,
228, 156))},
            {JointType.FootRight, new SolidColorBrush(Color.FromRgb(77, 109,
243))}
        };

        /// <summary>
        /// The depth frame byte array. Only supports 320 * 240 at this time
        /// </summary>
        private readonly short[] _depthFrame32 = new short[320 * 240 * 4];

        /// <summary>
        /// Flag to show whether or not the gesture recogniser is capturing a
new pose
        /// </summary>
        private bool _capturing;

        /// <summary>
        /// Dynamic Time Warping object
        /// </summary>
        private DtwGestureRecognizer _dtw;

        /// <summary>
        /// How many frames occurred 'last time'. Used for calculating frames
per second
        /// </summary>
        private int _lastFrames;

        /// <summary>
        /// The 'last time' DateTime. Used for calculating frames per second
        /// </summary>
        private DateTime _lastTime = DateTime.MaxValue;

        /// <summary>
        /// The Natural User Interface runtime
        /// </summary>
        private KinectSensor  _nui;

        /// <summary>
        /// Total number of framed that have occurred. Used for calculating
frames per second
        /// </summary>
        private int _totalFrames;

        /// <summary>
        /// Switch used to ignore certain skeleton frames
        /// </summary>
        private int _flipFlop;

        /// <summary>
        /// ArrayList of coordinates which are recorded in sequence to define
one gesture
        /// </summary>
        private ArrayList _video;

        /// <summary>
        /// ArrayList of coordinates which are recorded in sequence to define
one gesture
        /// </summary>
        private DateTime _captureCountdown = DateTime.Now;

        /// <summary>
        /// ArrayList of coordinates which are recorded in sequence to define
one gesture
        /// </summary>
        private Timer _captureCountdownTimer;

        /// <summary>
        /// Initializes a new instance of the MainWindow class
        /// </summary>
        public MainWindow()
        {
            InitializeComponent();
        }

        /// <summary>
        /// Opens the sent text file and creates a _dtw recorded gesture
sequence
        /// Currently not very flexible and totally intolerant of errors.
        /// </summary>
        /// <param name="fileLocation">Full path to the gesture file</param>
        public void LoadGesturesFromFile(string fileLocation)
        {
            int itemCount = 0;
            string line;
            string gestureName = String.Empty;

            // Defaulting this to 12 here for now as it meets my current
need.
            ArrayList frames = new ArrayList();
            double[] items = new double[12];

            // Read the file and display it line by line.
            System.IO.StreamReader file = new
System.IO.StreamReader(fileLocation);
            while ((line = file.ReadLine()) != null)
            {
                if (line.StartsWith("@"))
                {
                    gestureName = line;
                    continue;
                }

                if (line.StartsWith("~"))
                {
                    frames.Add(items);
                    itemCount = 0;
                    items = new double[12];
                    continue;
                }

                if (!line.StartsWith("----"))
                {
                    items[itemCount] = Double.Parse(line);
                }

                itemCount++;

                if (line.StartsWith("----"))
                {
                    _dtw.AddOrUpdate(frames, gestureName);
                    frames = new ArrayList();
                    gestureName = String.Empty;
                    itemCount = 0;
                }
            }

            file.Close();
        }

        /// <summary>
        /// Called each time a skeleton frame is ready. Passes skeletal data
to the DTW processor
```

```
        /// </summary>
        /// <param name="sender">The sender object</param>
        /// <param name="e">Skeleton Frame Ready Event Args</param>
        private static void SkeletonExtractSkeletonFrameReady(object sender,
SkeletonFrameReadyEventArgs e)
        {
            using (var skeletonFrame = e.OpenSkeletonFrame())
            {
                if (skeletonFrame == null) return; // sometimes frame image
comes null, so skip it.
                var skeletons = new
Skeleton[skeletonFrame.SkeletonArrayLength];
                skeletonFrame.CopySkeletonDataTo(skeletons);

                foreach (Skeleton data in skeletons)
                {
                    Skeleton2DDataExtract.ProcessData(data);
                }
            }
        }

        /// <summary>
        /// Converts a 16-bit grayscale depth frame which includes player
indexes into a 32-bit frame that displays different players in different
colors
        /// </summary>
        /// <param name="depthFrame16">The depth frame byte array</param>
        /// <returns>A depth frame byte array containing a player
image</returns>
        private short[] ConvertDepthFrame(short[] depthFrame16)
        {
            for (int i16 = 0, i32 = 0; i16 < depthFrame16.Length && i32 <
_depthFrame32.Length; i16 += 2, i32 += 4)
            {
                int player = depthFrame16[i16] & 0x07;
                int realDepth = (depthFrame16[i16 + 1] << 5) |
(depthFrame16[i16] >> 3);

                // transform 13-bit depth information into an 8-bit intensity
appropriate
                // for display (we disregard information in most significant
bit)
                var intensity = (short)(255 - (255 * realDepth / 0x0fff));

                _depthFrame32[i32 + RedIdx] = 0;
                _depthFrame32[i32 + GreenIdx] = 0;
                _depthFrame32[i32 + BlueIdx] = 0;

                // choose different display colors based on player
                switch (player)
                {
                    case 0:
                        _depthFrame32[i32 + RedIdx] = (byte)(intensity / 2);
                        _depthFrame32[i32 + GreenIdx] = (byte)(intensity /
2);

                        _depthFrame32[i32 + BlueIdx] = (byte)(intensity / 2);
                        break;
                    case 1:
                        _depthFrame32[i32 + RedIdx] = intensity;
                        break;
                    case 2:
                        _depthFrame32[i32 + GreenIdx] = intensity;
                        break;
                    case 3:
                        _depthFrame32[i32 + RedIdx] = (byte)(intensity / 4);
                        _depthFrame32[i32 + GreenIdx] = intensity;
                        _depthFrame32[i32 + BlueIdx] = intensity;
                        break;
                    case 4:
                        _depthFrame32[i32 + RedIdx] = intensity;
                        _depthFrame32[i32 + GreenIdx] = intensity;
                        _depthFrame32[i32 + BlueIdx] = (byte)(intensity / 4);
                        break;
                    case 5:
                        _depthFrame32[i32 + RedIdx] = intensity;
                        _depthFrame32[i32 + GreenIdx] = (byte)(intensity /
4);

                        _depthFrame32[i32 + BlueIdx] = intensity;
                        break;
                    case 6:
                        _depthFrame32[i32 + RedIdx] = (byte)(intensity / 2);
                        _depthFrame32[i32 + GreenIdx] = (byte)(intensity /
2);

                        _depthFrame32[i32 + BlueIdx] = intensity;
                        break;
                    case 7:
                        _depthFrame32[i32 + RedIdx] = (byte)(255 -
intensity);

                        _depthFrame32[i32 + GreenIdx] = (byte)(255 -
intensity);

                        _depthFrame32[i32 + BlueIdx] = (byte)(255 -
intensity);

                        break;
                }
            }

            return _depthFrame32;
        }

        /// <summary>
        /// Called when each depth frame is ready
        /// </summary>
        /// <param name="sender">The sender object</param>
        /// <param name="e">Image Frame Ready Event Args</param>
        private void NuiDepthFrameReady(object sender,
DepthImageFrameReadyEventArgs e)
        {
            //PlanarImage image = e.ImageFrame.Image;
            using (var image = e.OpenDepthImageFrame())
            {
                if (image == null) return; // sometimes frame image comes
null, so skip it.

                depthImage.Source = image.ToBitmapSource();
            }
            ++_totalFrames;

            DateTime cur = DateTime.Now;
            if (cur.Subtract(_lastTime) > TimeSpan.FromSeconds(1))
            {
                int frameDiff = _totalFrames - _lastFrames;
                _lastFrames = _totalFrames;
                _lastTime = cur;
                frameRate.Text = frameDiff + " fps";
            }
        }

        /// <summary>
        /// Gets the display position (i.e. where in the display image) of a
Joint
        /// </summary>
        /// <param name="joint">Kinect NUI Joint</param>
        /// <returns>Point mapped location of sent joint</returns>
        private Point GetDisplayPosition(Joint joint)
        {
            float depthX, depthY;
            var pos = _nui.MapSkeletonPointToDepth(joint.Position,
DepthImageFormat.Resolution320x240Fps30);

            depthX = pos.X;
            depthY = pos.Y;

            int colorX, colorY;

            // Only ImageResolution.Resolution640x480 is supported at this
point
            var pos2 = _nui.MapSkeletonPointToColor(joint.Position,
ColorImageFormat.RgbResolution640x480Fps30);
            colorX = pos2.X;
            colorY = pos2.Y;

            // Map back to skeleton.Width & skeleton.Height
            return new Point((int)(skeletonCanvas.Width * colorX / 640.0),
(int)(skeletonCanvas.Height * colorY / 480));
        }

        /// <summary>
        /// Works out how to draw a line ('bone') for sent Joints
        /// </summary>
        /// <param name="joints">Kinect NUI Joints</param>
        /// <param name="brush">The brush we'll use to colour the
joints</param>
        /// <param name="ids">The JointsIDs we're interested in</param>
        /// <returns>A line or lines</returns>
        private Polyline GetBodySegment(JointCollection joints, Brush brush,
params JointType[] ids)
        {

            var points = new PointCollection(ids.Length);
            foreach (JointType t in ids)
            {
                points.Add(GetDisplayPosition(joints[t]));
            }

            var polyline = new Polyline();
            polyline.Points = points;
            polyline.Stroke = brush;
            polyline.StrokeThickness = 5;
            return polyline;
        }

        /// <summary>
        /// Runds every time a skeleton frame is ready. Updates the skeleton
canvas with new joint and polyline locations.
        /// </summary>
        /// <param name="sender">The sender object</param>
        /// <param name="e">Skeleton Frame Event Args</param>
        private void NuiSkeletonFrameReady(object sender,
SkeletonFrameReadyEventArgs e)
        {
            Skeleton[] skeletons;
            using (var frame = e.OpenSkeletonFrame())
            {
                if (frame == null) return;
                skeletons = new Skeleton[frame.SkeletonArrayLength];
                frame.CopySkeletonDataTo(skeletons);
            }

            int iSkeleton = 0;
            var brushes = new Brush[6];
            brushes[0] = new SolidColorBrush(Color.FromRgb(255, 0, 0));
            brushes[1] = new SolidColorBrush(Color.FromRgb(0, 255, 0));
            brushes[2] = new SolidColorBrush(Color.FromRgb(64, 255, 255));
            brushes[3] = new SolidColorBrush(Color.FromRgb(255, 255, 64));
            brushes[4] = new SolidColorBrush(Color.FromRgb(255, 64, 255));
            brushes[5] = new SolidColorBrush(Color.FromRgb(128, 128, 255));

            skeletonCanvas.Children.Clear();
            foreach (var data in skeletons)
            {
                if (SkeletonTrackingState.Tracked == data.TrackingState)
                {
                    // Draw bones
                    Brush brush = brushes[iSkeleton % brushes.Length];
                    skeletonCanvas.Children.Add(GetBodySegment(data.Joints,
brush, JointType.HipCenter, JointType.Spine, JointType.ShoulderCenter,
JointType.Head));
                    skeletonCanvas.Children.Add(GetBodySegment(data.Joints,
brush, JointType.ShoulderCenter, JointType.ShoulderLeft, JointType.ElbowLeft,
JointType.WristLeft, JointType.HandLeft));
                    skeletonCanvas.Children.Add(GetBodySegment(data.Joints,
brush, JointType.ShoulderCenter, JointType.ShoulderRight,
JointType.ElbowRight, JointType.WristRight, JointType.HandRight));
                    skeletonCanvas.Children.Add(GetBodySegment(data.Joints,
brush, JointType.HipCenter, JointType.HipLeft, JointType.KneeLeft,
JointType.AnkleLeft, JointType.FootLeft));
```

```
                    skeletonCanvas.Children.Add(GetBodySegment(data.Joints,
brush, JointType.HipCenter, JointType.HipRight, JointType.KneeRight,
JointType.AnkleRight, JointType.FootRight));

                    // Draw joints
                    foreach (Joint joint in data.Joints)
                    {
                        Point jointPos = GetDisplayPosition(joint);
                        var jointLine = new Line();
                        jointLine.X1 = jointPos.X - 3;
                        jointLine.X2 = jointLine.X1 + 6;
                        jointLine.Y1 = jointLine.Y2 = jointPos.Y;
                        jointLine.Stroke = _jointColors[joint.JointType];
                        jointLine.StrokeThickness = 6;
                        skeletonCanvas.Children.Add(jointLine);
                    }
                }

                iSkeleton++;
            } // for each skeleton
        }

        /// <summary>
        /// Called every time a video (RGB) frame is ready
        /// </summary>
        /// <param name="sender">The sender object</param>
        /// <param name="e">Image Frame Ready Event Args</param>
        private void NuiColorFrameReady(object sender,
ColorImageFrameReadyEventArgs e)
        {
            // 32-bit per pixel, RGBA image
            using (var image = e.OpenColorImageFrame())
            {
                if (image == null) return; // sometimes frame image comes
null, so skip it.

                videoImage.Source = image.ToBitmapSource();
            }
        }

        /// <summary>
        /// Runs after the window is loaded
        /// </summary>
        /// <param name="sender">The sender object</param>
        /// <param name="e">Routed Event Args</param>
        private void WindowLoaded(object sender, RoutedEventArgs e)
        {
            _nui = (from i in KinectSensor.KinectSensors
                    where i.Status == KinectStatus.Connected
                    select i).FirstOrDefault();

            if (_nui == null)
                throw new NotSupportedException("No kinects connected!");

            try
            {
```

```
_nui.DepthStream.Enable(DepthImageFormat.Resolution320x240Fps30);
                _nui.SkeletonStream.Enable();

_nui.ColorStream.Enable(ColorImageFormat.RgbResolution640x480Fps30);
                _nui.Start();
            }
            catch (InvalidOperationException)
            {
                System.Windows.MessageBox.Show("Runtime initialization
failed. Please make sure Kinect device is plugged in.");
                return;
            }

            _lastTime = DateTime.Now;

            _dtw = new DtwGestureRecognizer(12, 0.6, 2, 2, 10);
            _video = new ArrayList();

            // If you want to see the depth image and frames per second then
include this
            _nui.DepthFrameReady += NuiDepthFrameReady;
            _nui.SkeletonFrameReady += NuiSkeletonFrameReady;
            _nui.SkeletonFrameReady += SkeletonExtractSkeletonFrameReady;

            // If you want to see the RGB stream then include this
            _nui.ColorFrameReady += NuiColorFrameReady;

            Skeleton2DDataExtract.Skeleton2DdataCoordReady +=
NuiSkeleton2DdataCoordReady;

            // Update the debug window with Sequences information
            //dtwTextOutput.Text = _dtw.RetrieveText();

            Debug.WriteLine("Finished Window Loading");
            Number1Question();
            Number2Question();
            Number3Operator();




        }
            private void Number1Question()
            {
              int randomNumber = random.Next(1, 9);
                if (randomNumber == 0)
                {
                number1.Source = new BitmapImage(new
Uri("/DTWGestureRecognition;component/Images/0.png", UriKind.Relative));
                number1questionconst = 0.00m;

                }
                else
                    if (randomNumber == 1)
                    {
                        number1.Source = new BitmapImage(new
Uri("/DTWGestureRecognition;component/Images/1.png", UriKind.Relative));
                        number1questionconst = 1.00m;
                    }
                else
                    if (randomNumber == 2)
                        {
                            number1.Source = new BitmapImage(new
Uri("/DTWGestureRecognition;component/Images/2.png", UriKind.Relative));
                            number1questionconst = 2.00m;
                        }
                else
                    if (randomNumber == 3)
                        {
                            number1.Source = new BitmapImage(new
Uri("/DTWGestureRecognition;component/Images/3.png", UriKind.Relative));
                            number1questionconst = 3.00m;
                        }
                else
                    if (randomNumber == 4)
                        {
                            number1.Source = new BitmapImage(new
Uri("/DTWGestureRecognition;component/Images/4.png", UriKind.Relative));
                            number1questionconst = 4.00m;
                        }
                 else
                     if (randomNumber == 5)
                         {
                             number1.Source = new BitmapImage(new
Uri("/DTWGestureRecognition;component/Images/5.png", UriKind.Relative));
                             number1questionconst = 5.00m;
                         }
                 else
                      if (randomNumber == 6)
                         {
                             number1.Source = new BitmapImage(new
Uri("/DTWGestureRecognition;component/Images/6.png", UriKind.Relative));
                             number1questionconst = 6.00m;
                         }
                 else
                       if (randomNumber == 7)
                         {
                               number1.Source = new BitmapImage(new
Uri("/DTWGestureRecognition;component/Images/7.png", UriKind.Relative));
                               number1questionconst = 7.00m;
                         }
                 else
                      if (randomNumber == 8)
                         {
                               number1.Source = new BitmapImage(new
Uri("/DTWGestureRecognition;component/Images/8.png", UriKind.Relative));
                               number1questionconst = 8.00m;
                         }
                 else
                      if (randomNumber == 9)
                         {
                              number1.Source = new BitmapImage(new
Uri("/DTWGestureRecognition;component/Images/9.png", UriKind.Relative));
                              number1questionconst = 9.00m;
                        }
            }

            private void Number2Question()
            {
            int randomNumber2 = random.Next(1, 9);
            if (randomNumber2 == 0)
                {
                    number2.Source = new BitmapImage(new
Uri("/DTWGestureRecognition;component/Images/0.png", UriKind.Relative));
                    number2questionconst = 0.00m;
                }
            else
                if (randomNumber2 == 1)
                    {
                        number2.Source = new BitmapImage(new
Uri("/DTWGestureRecognition;component/Images/1.png", UriKind.Relative));
                        number2questionconst = 1.00m;
                    }
            else
                if (randomNumber2 == 2)
                    {
                        number2.Source = new BitmapImage(new
Uri("/DTWGestureRecognition;component/Images/2.png", UriKind.Relative));
                        number2questionconst = 2.00m;
                    }
              else
                 if (randomNumber2 == 3)
                    {
                        number2.Source = new BitmapImage(new
Uri("/DTWGestureRecognition;component/Images/3.png", UriKind.Relative));
                        number2questionconst = 3.00m;
                    }
              else
                  if (randomNumber2 == 4)
                    {
                        number2.Source = new BitmapImage(new
Uri("/DTWGestureRecognition;component/Images/4.png", UriKind.Relative));
                        number2questionconst = 4.00m;
                    }
              else
                  if (randomNumber2 == 5)
                    {
                        number2.Source = new BitmapImage(new
Uri("/DTWGestureRecognition;component/Images/5.png", UriKind.Relative));
                        number2questionconst = 5.00m;
                    }
              else
                  if (randomNumber2 == 6)
                    {
                        number2.Source = new BitmapImage(new
Uri("/DTWGestureRecognition;component/Images/6.png", UriKind.Relative));
                        number2questionconst = 6.00m;
                    }
                else
                    if (randomNumber2 == 7)
                      {
                            number2.Source = new BitmapImage(new
Uri("/DTWGestureRecognition;component/Images/7.png", UriKind.Relative));
                            number2questionconst = 7.00m;
                      }
                else
                    if (randomNumber2 == 8)
                      {
                             number2.Source = new BitmapImage(new
Uri("/DTWGestureRecognition;component/Images/8.png", UriKind.Relative));
                             number2questionconst = 8.00m;
                      }
                else
                    if (randomNumber2 == 9)
                      {
                             number2.Source = new BitmapImage(new
Uri("/DTWGestureRecognition;component/Images/9.png", UriKind.Relative));
                             number2questionconst = 9.00m;
                      }
            }
            private void Number3Operator()
            {

            Random random3 = new Random();
            int randomNumber3 = random3.Next(0, 4);

            switch (randomNumber3)
            {
                case 0:
                    number3operatorconst = "*";
                    answertoquestion = number1questionconst *
number2questionconst;
                    answerbox.Text = answertoquestion.ToString("N2");
                    break;
                case 1:
                    number3operatorconst = "/";
                    answertoquestion = number1questionconst /
number2questionconst;
                    answerbox.Text = answertoquestion.ToString("N2");
                    break;
                case 2:
                    number3operatorconst = "+";
                    answertoquestion = number1questionconst +
number2questionconst;
                    answerbox.Text = answertoquestion.ToString("N2");
                    break;
                case 3:
                    number3operatorconst = "subtract";
                    answertoquestion = number1questionconst -
number2questionconst;
                    answerbox.Text = answertoquestion.ToString("N2");
                    break;
                case 4:
                    number3operatorconst = "subtract";
                    answertoquestion = number1questionconst -
number2questionconst;
                    answerbox.Text = answertoquestion.ToString("N2");
                    break;
            }
            }

        /// </summary>
        /// <param name="sender">The sender object</param>
        /// <param name="e">Event Args</param>
        private void WindowClosed(object sender, EventArgs e)
        {
            Debug.WriteLine("Stopping NUI");
            _nui.Stop();
            Debug.WriteLine("NUI stopped");
            Environment.Exit(0);
        }

        /// <summary>
        /// Runs every time our 2D coordinates are ready.
        /// </summary>
        /// <param name="sender">The sender object</param>
        /// <param name="a">Skeleton 2Ddata Coord Event Args</param>
        private void NuiSkeleton2DdataCoordReady(object sender,
Skeleton2DdataCoordEventArgs a)
        {
            currentBufferFrame.Text = _video.Count.ToString();

            // We need a sensible number of frames before we start attempting
to match gestures against remembered sequences
            if (_video.Count > MinimumFrames && _capturing == false)
            {
                string s = _dtw.Recognize(_video);

                if (s.Contains("Addition"))
                {
                    if (number3operatorconst == "+")
                    {

                    Number1Question();
                    Number2Question();
                    Number3Operator();

                    SoundPlayer correct = new SoundPlayer("correct.wav");
                    correct.Play();

                    }

                    else

                    {
                        SoundPlayer correct = new SoundPlayer("wrong.wav");
                        correct.Play();
                    }
                }
                else
                {
                }

                if (s.Contains("Subtraction"))
                {
                    if (number3operatorconst == "subtract")
                    {
                        Number1Question();
                        Number2Question();
                        Number3Operator();
                        SoundPlayer correct = new SoundPlayer("correct.wav");
                        correct.Play();
                    }

                    else {
                        SoundPlayer correct = new SoundPlayer("wrong.wav");
                        correct.Play();
                    }
                }
                else
                {
                }

                if (s.Contains("Multiplying"))
                {
                    if (number3operatorconst == "*")
                    {
                        Number1Question();
                        Number2Question();
                        Number3Operator();
                        SoundPlayer correct = new SoundPlayer("correct.wav");
                        correct.Play();
                    }

                    else {
                        SoundPlayer correct = new SoundPlayer("wrong.wav");
                        correct.Play();

                    }
                }
                else
                {
                }

                if (s.Contains("Dividing"))
                {
                    if (number3operatorconst == "/")
                    {
                        Number1Question();
                        Number2Question();
                        Number3Operator();
                        SoundPlayer correct = new SoundPlayer("correct.wav");
                        correct.Play();
                    }

                    else {
                        SoundPlayer correct = new SoundPlayer("wrong.wav");
                        correct.Play();
```

```
                    }
                }
                else
                {
                }
                results.Text = "Recognised as: " + s;
                if (!s.Contains("__UNKNOWN"))
                {
                    // There was no match so reset the buffer
                    _video = new ArrayList();
                }
            }

            // Ensures that we remember only the last x frames
            if (_video.Count > BufferSize)
            {
                // If we are currently capturing and we reach the maximum
buffer size then automatically store
                if (_capturing)
                {
                    DtwStoreClick(null, null);
                }
                else
                {
                    // Remove the first frame in the buffer
                    _video.RemoveAt(0);
                }
            }

            // Decide which skeleton frames to capture. Only do so if the
frames actually returned a number.
            if (!double.IsNaN(a.GetPoint(0).X))
            {
                // Optionally register only 1 frame out of every n
                _flipFlop = (_flipFlop + 1) % Ignore;
                if (_flipFlop == 0)
                {
                    _video.Add(a.GetCoords());
                }
            }

            // Update the debug window with Sequences information
            //dtwTextOutput.Text = _dtw.RetrieveText();
        }

        /// <summary>
        /// Read mode. Sets our control variables and button enabled states
        /// </summary>
        /// <param name="sender">The sender object</param>
        /// <param name="e">Routed Event Args</param>
        private void DtwReadClick(object sender, RoutedEventArgs e)
        {
            // Set the buttons enabled state
            dtwRead.IsEnabled = false;
            dtwCapture.IsEnabled = true;
            //dtwStore.IsEnabled = false;
            // Set the capturing? flag
            _capturing = false;

            // Update the status display
            status.Text = "Reading";
        }

        /// <summary>
        /// Starts a countdown timer to enable the player to get in position
to record gestures
        /// </summary>
        /// <param name="sender">The sender object</param>
        /// <param name="e">Routed Event Args</param>
        private void DtwCaptureClick(object sender, RoutedEventArgs e)
        {
            _captureCountdown =
DateTime.Now.AddSeconds(CaptureCountdownSeconds);

            _captureCountdownTimer = new Timer();
            _captureCountdownTimer.Interval = 50;
            _captureCountdownTimer.Start();
            _captureCountdownTimer.Tick += CaptureCountdown;
        }

        /// <summary>
        /// The method fired by the countdown timer. Either updates the
countdown or fires the StartCapture method if the timer expires
        /// </summary>
        /// <param name="sender">The sender object</param>
        /// <param name="e">Event Args</param>
        private void CaptureCountdown(object sender, EventArgs e)
        {
            if (sender == _captureCountdownTimer)
            {
                if (DateTime.Now < _captureCountdown)
                {
                    status.Text = "Wait " + ((_captureCountdown -
DateTime.Now).Seconds + 1) + " seconds";
                }
                else
                {
                    _captureCountdownTimer.Stop();
                    status.Text = "Recording gesture";
                    StartCapture();
                }
            }
        }

        /// <summary>
        /// Capture mode. Sets our control variables and button enabled
states
        /// </summary>
        private void StartCapture()
        {
            // Set the buttons enabled state
            dtwRead.IsEnabled = false;
            dtwCapture.IsEnabled = false;
            //dtwStore.IsEnabled = true;

            // Set the capturing? flag
            _capturing = true;

            ////_captureCountdownTimer.Dispose();

            status.Text = "Recording gesture" + gestureList.Text;

            // Clear the _video buffer and start from the beginning
            _video = new ArrayList();
        }

        /// <summary>
        /// Stores our gesture to the DTW sequences list
        /// </summary>
        /// <param name="sender">The sender object</param>
        /// <param name="e">Routed Event Args</param>
        private void DtwStoreClick(object sender, RoutedEventArgs e)
        {
            // Set the buttons enabled state
            dtwRead.IsEnabled = false;
            dtwCapture.IsEnabled = true;
            //dtwStore.IsEnabled = false;

            // Set the capturing? flag
            _capturing = false;

            status.Text = "Remembering " + gestureList.Text;

            // Add the current video buffer to the dtw sequences list
            _dtw.AddOrUpdate(_video, gestureList.Text);
            results.Text = "Gesture " + gestureList.Text + "added";

            // Scratch the _video buffer
            _video = new ArrayList();

            // Switch back to Read mode
            DtwReadClick(null, null);
        }

        /// <summary>
        /// Stores our gesture to the DTW sequences list
        /// </summary>
        /// <param name="sender">The sender object</param>
        /// <param name="e">Routed Event Args</param>
        private void DtwSaveToFile(object sender, RoutedEventArgs e)
        {
            string fileName = GestureSaveFileNamePrefix +
DateTime.Now.ToString("yyyy-MM-dd_HH-mm") + ".txt";
            System.IO.File.WriteAllText(GestureSaveFileLocation + fileName,
_dtw.RetrieveText());
            status.Text = "Saved to " + fileName;
        }

        /// <summary>
        /// Loads the user's selected gesture file
        /// </summary>
        /// <param name="sender">The sender object</param>
        /// <param name="e">Routed Event Args</param>
        private void DtwLoadFile(object sender, RoutedEventArgs e)
        {
            // Create OpenFileDialog
            Microsoft.Win32.OpenFileDialog dlg = new
Microsoft.Win32.OpenFileDialog();

            // Set filter for file extension and default file extension
            dlg.DefaultExt = ".txt";
            dlg.Filter = "Text documents (.txt)|*.txt";

            dlg.InitialDirectory = GestureSaveFileLocation;

            // Display OpenFileDialog by calling ShowDialog method
            Nullable<bool> result = dlg.ShowDialog();

            // Get the selected file name and display in a TextBox
            if (result == true)
            {
                // Open document
                LoadGesturesFromFile(dlg.FileName);
                //dtwTextOutput.Text = _dtw.RetrieveText();
                status.Text = "Gestures loaded!";
            }
        }

        /// <summary>
        /// Stores our gesture to the DTW sequences list
        /// </summary>
        /// <param name="sender">The sender object</param>
        /// <param name="e">Routed Event Args</param>
        private void DtwShowGestureText(object sender, RoutedEventArgs e)
        {
            //dtwTextOutput.Text = _dtw.RetrieveText();
        }

        private void button1_Click(object sender, RoutedEventArgs e)
        {
            Number1Question();
            Number2Question();
            Number3Operator();

        }

        private void videoImage_ImageFailed(object sender,
ExceptionRoutedEventArgs e)
        {

        }
    }
}
```

The BitmapSourceExtensions.cs file:

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

using System.Diagnostics.CodeAnalysis;
using System.IO;
using System.Security;
using System.Windows.Media;
using System.Windows.Media.Imaging;

namespace Microsoft.Kinect
{
    public static class BitmapSourceExtensions
    {
        public static BitmapSource ToBitmapSource(this byte[] pixels, int
width, int height)
        {
            return ToBitmapSource(pixels, width, height, PixelFormats.Bgr32);
        }

        private static BitmapSource ToBitmapSource(this byte[] pixels, int
width, int height, System.Windows.Media.PixelFormat format)
        {
            return BitmapSource.Create(width, height, 96, 96, format, null,
pixels, width * format.BitsPerPixel / 8);
        }
    }
}
```

The DTWGestureRecognizer.cs file:

```
//-----------------------------------------------------------------------
// <copyright file="DtwGestureRecognizer.cs" company="Rhemyst and Rymix">
//     Open Source. Do with this as you will. Include this statement or
//     don't - whatever you like.
//
//     No warranty or support given. No guarantees this will work or meet
//     your needs. Some elements of this project have been tailored to
//     the authors' needs and therefore don't necessarily follow best
//     practice. Subsequent releases of this project will (probably) not
//     be compatible with different versions, so whatever you do, don't
//     overwrite your implementation with any new releases of this
//     project!
//
//     Enjoy working with Kinect!
//
// </copyright>
// Gesture Recognition Based Calculator
// Modified by Sree Bharat Dasari
// Used lots of references that i found in google.
// by saying lots i mean a LOT !! ;)
//
//-----------------------------------------------------------------------

using System.Diagnostics;

namespace DTWGestureRecognition
{
    using System;
    using System.Collections;

    /// <summary>
    /// Dynamic Time Warping nearest neighbour sequence comparison class.
    /// </summary>
    internal class DtwGestureRecognizer
    {
        /// <summary>
        /// Size of obeservations vectors.
        /// </summary>
        private readonly int _dimension;

        /// <summary>
        /// Maximum distance between the last observations of each sequence.
        /// </summary>
        private readonly double _firstThreshold;

        /// <summary>
        /// Minimum length of a gesture before it can be recognised
        /// </summary>
        private readonly double _minimumLength;

        /// <summary>
        /// Maximum DTW distance between an example and a sequence being
classified.
        /// </summary>
        private readonly double _globalThreshold;
        /// <summary>
        /// The gesture names. Index matches that of the sequences array in
_sequences
        /// </summary>
        private readonly ArrayList _labels;

        /// <summary>
        /// Maximum vertical or horizontal steps in a row.
        /// </summary>
        private readonly int _maxSlope;

        /// <summary>
        /// The recorded gesture sequences
        /// </summary>
        private readonly ArrayList _sequences;

        /// <summary>
        /// Initializes a new instance of the DtwGestureRecognizer class
        /// First DTW constructor
        /// </summary>
        /// <param name="dim">Vector size</param>
        /// <param name="threshold">Maximum distance between the last
observations of each sequence</param>
        /// <param name="firstThreshold">Minimum threshold</param>
        public DtwGestureRecognizer(int dim, double threshold, double
firstThreshold, double minLen)
        {
            _dimension = dim;
            _sequences = new ArrayList();
            _labels = new ArrayList();
            _globalThreshold = threshold;
            _firstThreshold = firstThreshold;
            _maxSlope = int.MaxValue;
            _minimumLength = minLen;
        }

        /// <summary>
        /// Initializes a new instance of the DtwGestureRecognizer class
        /// Second DTW constructor
        /// </summary>
        /// <param name="dim">Vector size</param>
        /// <param name="threshold">Maximum distance between the last
observations of each sequence</param>
        /// <param name="firstThreshold">Minimum threshold</param>
        /// <param name="ms">Maximum vertical or horizontal steps in a
row</param>
        public DtwGestureRecognizer(int dim, double threshold, double
firstThreshold, int ms, double minLen)
        {
            _dimension = dim;
            _sequences = new ArrayList();
            _labels = new ArrayList();
            _globalThreshold = threshold;
            _firstThreshold = firstThreshold;
            _maxSlope = ms;
            _minimumLength = minLen;
        }

        /// <summary>
        /// Add a seqence with a label to the known sequences library.
        /// The gesture MUST start on the first observation of the sequence
and end on the last one.
        /// Sequences may have different lengths.
        /// </summary>
        /// <param name="seq">The sequence</param>
        /// <param name="lab">Sequence name</param>
        public void AddOrUpdate(ArrayList seq, string lab)
        {
            // First we check whether there is already a recording for this
label. If so overwrite it, otherwise add a new entry
            int existingIndex = -1;

            for (int i = 0; i < _labels.Count; i++)
            {
                if ((string)_labels[i] == lab)
                {
                    existingIndex = i;
                }
            }

            // If we have a match then remove the entries at the existing
index to avoid duplicates. We will add the new entries later anyway
            if (existingIndex >= 0)
            {
                _sequences.RemoveAt(existingIndex);
                _labels.RemoveAt(existingIndex);
            }

            // Add the new entries
            _sequences.Add(seq);
            _labels.Add(lab);
        }

        /// <summary>
        /// Recognize gesture in the given sequence.
        /// It will always assume that the gesture ends on the last
observation of that sequence.
        /// If the distance between the last observations of each sequence is
too great,
        /// or if the overall DTW distance between the two sequence is too
great, no gesture will be recognized.
        /// </summary>
        /// <param name="seq">The sequence to recognise</param>
        /// <returns>The recognised gesture name</returns>
        public string Recognize(ArrayList seq)
        {
            double minDist = double.PositiveInfinity;
            string classification = "__UNKNOWN";
            for (int i = 0; i < _sequences.Count; i++)
            {
                var example = (ArrayList) _sequences[i];
                if (Dist2((double[]) seq[seq.Count - 1], (double[])
example[example.Count - 1]) < _firstThreshold)
```

```
                {
                    double d = Dtw(seq, example) / example.Count;
                    if (d < minDist)
                    {
                        minDist = d;
                        classification = (string)_labels[i];
                    }
                }
            }

            return (minDist < _globalThreshold ? classification :
"__UNKNOWN") + " " /*+minDist.ToString()*/;
        }

        /// <summary>
        /// Retrieves a text represeantation of the _label and its associated
_sequence
        /// For use in dispaying debug information and for saving to file
        /// </summary>
        /// <returns>A string containing all recorded gestures and their
names</returns>
        public string RetrieveText()
        {
            string retStr = String.Empty;

            if (_sequences != null)
            {
                // Iterate through each gesture
                for (int gestureNum = 0; gestureNum < _sequences.Count;
gestureNum++)
                {
                    // Echo the label
                    retStr += _labels[gestureNum] + "\r\n";

                    int frameNum = 0;

                    //Iterate through each frame of this gesture
                    foreach (double[] frame in
((ArrayList)_sequences[gestureNum]))
                    {
                        // Extract each double
                        foreach (double dub in (double[])frame)
                        {
                            retStr += dub + "\r\n";
                        }

                        // Signifies end of this double
                        retStr += "~\r\n";

                        frameNum++;
                    }

                    // Signifies end of this gesture
                    retStr += "----";
                    if (gestureNum < _sequences.Count - 1)
                    {
                        retStr += "\r\n";
                    }
                }
            }

            return retStr;
        }

        /// <summary>
        /// Compute the min DTW distance between seq2 and all possible
endings of seq1.
        /// </summary>
        /// <param name="seq1">The first array of sequences to
compare</param>
        /// <param name="seq2">The second array of sequences to
compare</param>
        /// <returns>The best match</returns>
        public double Dtw(ArrayList seq1, ArrayList seq2)
        {
            // Init
            var seq1R = new ArrayList(seq1);
            seq1R.Reverse();
            var seq2R = new ArrayList(seq2);
            seq2R.Reverse();
            var tab = new double[seq1R.Count + 1, seq2R.Count + 1];
            var slopeI = new int[seq1R.Count + 1, seq2R.Count + 1];
            var slopeJ = new int[seq1R.Count + 1, seq2R.Count + 1];

            for (int i = 0; i < seq1R.Count + 1; i++)
            {
                for (int j = 0; j < seq2R.Count + 1; j++)
                {
                    tab[i, j] = double.PositiveInfinity;
                    slopeI[i, j] = 0;
                    slopeJ[i, j] = 0;
                }
            }

            tab[0, 0] = 0;

            // Dynamic computation of the DTW matrix.
            for (int i = 1; i < seq1R.Count + 1; i++)
            {
                for (int j = 1; j < seq2R.Count + 1; j++)
                {
                    if (tab[i, j - 1] < tab[i - 1, j - 1] && tab[i, j - 1] <
tab[i - 1, j] &&
                        slopeI[i, j - 1] < _maxSlope)
                    {
                        tab[i, j] = Dist2((double[]) seq1R[i - 1], (double[])
seq2R[j - 1]) + tab[i, j - 1];
                        slopeI[i, j] = slopeJ[i, j - 1] + 1;
                        slopeJ[i, j] = 0;
                    }
                    else if (tab[i - 1, j] < tab[i - 1, j - 1] && tab[i - 1,
j] < tab[i, j - 1] &&
                             slopeJ[i - 1, j] < _maxSlope)
                    {
                        tab[i, j] = Dist2((double[]) seq1R[i - 1], (double[])
seq2R[j - 1]) + tab[i - 1, j];
                        slopeI[i, j] = 0;
                        slopeJ[i, j] = slopeJ[i - 1, j] + 1;
                    }
                    else
                    {
                        tab[i, j] = Dist2((double[]) seq1R[i - 1], (double[])
seq2R[j - 1]) + tab[i - 1, j - 1];
                        slopeI[i, j] = 0;
                        slopeJ[i, j] = 0;
                    }
                }
            }

            // Find best between seq2 and an ending (postfix) of seq1.
            double bestMatch = double.PositiveInfinity;
            for (int i = 1; i < (seq1R.Count + 1) - _minimumLength; i++)
            {
                if (tab[i, seq2R.Count] < bestMatch)
                {
                    bestMatch = tab[i, seq2R.Count];
                }
            }

            return bestMatch;
        }

        /// <summary>
        /// Computes a 1-distance between two observations. (aka Manhattan
distance).
        /// </summary>
        /// <param name="a">Point a (double)</param>
        /// <param name="b">Point b (double)</param>
        /// <returns>Manhattan distance between the two points</returns>
        private double Dist1(double[] a, double[] b)
        {
            double d = 0;
            for (int i = 0; i < _dimension; i++)
            {
                d += Math.Abs(a[i] - b[i]);
            }

            return d;
        }

        /// <summary>
        /// Computes a 2-distance between two observations. (aka Euclidian
distance).
        /// </summary>
        /// <param name="a">Point a (double)</param>
        /// <param name="b">Point b (double)</param>
        /// <returns>Euclidian distance between the two points</returns>
        private double Dist2(double[] a, double[] b)
        {
            double d = 0;
            for (int i = 0; i < _dimension; i++)
            {
                d += Math.Pow(a[i] - b[i], 2);
            }

            return Math.Sqrt(d);
        }
    }
}
```

The ImageFrameCommonExtensions.cs file:

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

using System;
using System.Collections.Generic;
using System.Diagnostics;
using Microsoft.Kinect;

[assembly: CLSCompliant(true)]
namespace Microsoft.Kinect
{
    internal static class ImageFrameCommonExtensions
    {
        public const int RedIndex = 2;
        public const int GreenIndex = 1;
        public const int BlueIndex = 0;

        const float MaxDepthDistance = 4000; // 4000 seemed to be max value
returned
        const float MinDepthDistance = 800; // 800 seemed to be the min value
returned
        const float MaxDepthDistanceOffset = MaxDepthDistance -
MinDepthDistance;

        public static int GetDistance(this DepthImageFrame depthFrame, int x,
int y)
        {

            var width = depthFrame.Width;

            if (x > width)
                throw new ArgumentOutOfRangeException("x", "x is larger than
the width");

            if (y > depthFrame.Height)
                throw new ArgumentOutOfRangeException("y", "y is larger than
the height");

            if (x < 0)
                throw new ArgumentOutOfRangeException("x", "x is smaller than
zero");

            if (y < 0)
                throw new ArgumentOutOfRangeException("y", "y is smaller than
zero");


            var index = (width * y) + x;

            short[] allData = new short[depthFrame.PixelDataLength];

            depthFrame.CopyPixelDataTo(allData);
            return GetDepth(allData[index]);

        }

        public static void GetMidpoint(this short[] depthData, int width, int
height, int startX, int startY, int endX, int endY, int minimumDistance, out
double xLocation, out double yLocation)
        {
            if (depthData == null)
                throw new ArgumentNullException("depthData");

            if (width * height != depthData.Length)
                throw new ArgumentOutOfRangeException("depthData", "Depth
Data length does not match target height and width");

            if (endX > width)
                throw new ArgumentOutOfRangeException("endX", "endX is larger
than the width");

            if (endY > height)
                throw new ArgumentOutOfRangeException("endY", "endY is larger
than the height");

            if (startX < 0)
                throw new ArgumentOutOfRangeException("startX", "startX is
smaller than zero");

            if (startY < 0)
                throw new ArgumentOutOfRangeException("startY", "startY is
smaller than zero");

            xLocation = 0;
            yLocation = 0;

            var counter = 0;
            for (var x = startX; x < endX; x++)
            {
                for (var y = startY; y < endY; y++)
                {
                    var depth = depthData[x + width * y];
                    if (depth > minimumDistance || depth <= 0)
                        continue;

                    xLocation += x;
                    yLocation += y;

                    counter++;
                }
            }

            if (counter <= 0)
                return;

            xLocation /= counter;
            yLocation /= counter;
        }
        public static short[] ToDepthArray(this DepthImageFrame image)
        {
            if (image == null)
                throw new ArgumentNullException("image");

            var width = image.Width;
            var height = image.Height;

            short[] rawDepthData = new short[image.PixelDataLength];
            image.CopyPixelDataTo(rawDepthData);

            //loop through rawDepthData and add the depth

            short[] allPoints = new short[image.PixelDataLength];

            for (int i = 0; i < rawDepthData.Length; i++)
            {
                allPoints[i] = (short) GetDepth(rawDepthData[i]);
            }

            return allPoints;
        }


        public static byte CalculateIntensityFromDepth(int distance)
        {

            return (byte)(255 - (255 * Math.Max(distance - MinDepthDistance,
0) / (MaxDepthDistanceOffset)));
        }


        public static void SkeletonOverlay(ref byte redFrame, ref byte
greenFrame, ref byte blueFrame, int player)
        {
            switch (player)
            {
                default: // case 0:
                    break;
                case 1:
                    greenFrame = 0;
                    blueFrame = 0;
                    break;
                case 2:
                    redFrame = 0;
                    greenFrame = 0;
                    break;
                case 3:
                    redFrame = 0;
                    blueFrame = 0;
                    break;
                case 4:
                    greenFrame = 0;
                    break;
                case 5:
                    blueFrame = 0;
```

```
                        break;
                    case 6:
                        redFrame = 0;
                        break;
                    case 7:
                        redFrame /= 2;
                        blueFrame = 0;
                        break;
                }
            }

            public static byte[] ConvertDepthFrameToBitmap(DepthImageFrame 
depthFrame)
            {
                if (depthFrame == null)
                {
                    return null;
                }

                short[] depthData = new short[depthFrame.PixelDataLength];
                depthFrame.CopyPixelDataTo(depthData);

                Byte[] depthColors = new Byte[depthData.Length * 4]; //four 
colors for each pixel

                for (int colorIndex = 0, depthIndex = 0; colorIndex < 
depthColors.Length;  colorIndex += 4, depthIndex++)
                {

                    //get the depth, then calculate the intensity (0-255 based on 
the depth)
                    //depth of -1 = dark brown

                    int depth = GetDepth(depthData[depthIndex]);

                    if (depth == -1)
                    {
                        // dark brown
                        depthColors[colorIndex + RedIndex] = 66;
                        depthColors[colorIndex + GreenIndex] = 66;
                        depthColors[colorIndex + BlueIndex] = 33;
                    }
                    else
                    {
                        var intensity = 
ImageFrameCommonExtensions.CalculateIntensityFromDepth(depth);

                        depthColors[colorIndex + RedIndex] = intensity;
                        depthColors[colorIndex + GreenIndex] = intensity;
                        depthColors[colorIndex + BlueIndex] = intensity;
                    }



                    //if the pixel is a player, choose a color
                    int player = GetPlayerIndex(depthData[depthIndex]);
                    SkeletonOverlay(
                        ref depthColors[colorIndex + RedIndex],
                        ref depthColors[colorIndex + GreenIndex],
                        ref depthColors[colorIndex + BlueIndex], player);
                }
                return depthColors;


            }



            public static int GetPlayerIndex(short depthPoint)
            {
                return depthPoint & DepthImageFrame.PlayerIndexBitmask;
            }

            public static int GetDepth(short depthPoint)
            {
                return depthPoint >> DepthImageFrame.PlayerIndexBitmaskWidth;
            }


        }
    }
```

The ImageFrameExtensions.cs file:

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

using System;
using System.Windows;
using System.Windows.Media;
using System.Windows.Media.Imaging;

namespace Microsoft.Kinect
{
    public static class ImageFrameExtensions
    {

        public static short[] ToDepthArray(this DepthImageFrame image)
        {
            return ImageFrameCommonExtensions.ToDepthArray(image);
        }

        [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Naming",
"CA1704:IdentifiersShouldBeSpelledCorrectly", MessageId = "x"),
System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Naming",
"CA1704:IdentifiersShouldBeSpelledCorrectly", MessageId = "y")]
        public static int GetDistance(this DepthImageFrame image, int x, int
y)
        {
            return ImageFrameCommonExtensions.GetDistance(image, x, y);
        }

        public static Point GetMidpoint(this short[] depthData, int width,
int height, int startX, int startY, int endX, int endY, int minimumDistance)
        {
            double x;
            double y;
            depthData.GetMidpoint(width, height, startX, startY, endX, endY,
minimumDistance, out x, out y);

            return new Point(x, y);
        }

        public static BitmapSource ToBitmapSource(this short[] depthData, int
width, int height, int minimumDistance, Color highlightColor)
        {
            if (depthData == null)
            {
                return null;
            }
                //depthData must be array of distances already

                var depthColors = new byte[depthData.Length * 4];

                for (int colorIndex = 0, depthIndex = 0; colorIndex <
depthColors.Length; colorIndex += 4, depthIndex++)
                {
                    //get the depth, then calculate the intensity (0-255
based on the depth)
                    //depth of -1 = dark brown

                    if (depthData[depthIndex] == -1)
                    {
                        // dark brown
                        depthColors[colorIndex +
ImageFrameCommonExtensions.RedIndex] = 66;
                        depthColors[colorIndex +
ImageFrameCommonExtensions.GreenIndex] = 66;
                        depthColors[colorIndex +
ImageFrameCommonExtensions.BlueIndex] = 33;
                    }
                    else
                    {

                        var intensity =
ImageFrameCommonExtensions.CalculateIntensityFromDepth(depthData[depthIndex])
;

                        depthColors[colorIndex +
ImageFrameCommonExtensions.RedIndex] = intensity;
                        depthColors[colorIndex +
ImageFrameCommonExtensions.GreenIndex] = intensity;
                        depthColors[colorIndex +
ImageFrameCommonExtensions.BlueIndex] = intensity;

                        //change color to highlight color
                        if (depthData[depthIndex] <= minimumDistance &&
depthData[depthIndex] > 0)
                        {
                            var color = Color.Multiply(highlightColor,
intensity / 255f);

                            depthColors[colorIndex +
ImageFrameCommonExtensions.RedIndex] = color.R;
                            depthColors[colorIndex +
ImageFrameCommonExtensions.GreenIndex] = color.G;
                            depthColors[colorIndex +
ImageFrameCommonExtensions.BlueIndex] = color.B;
                        }
                    }



                }

                return depthColors.ToBitmapSource(width, height);

        }


        public static BitmapSource ToBitmapSource(this DepthImageFrame image)
        {
            if (image == null)
            {
                return null;
            }

            var bytes =
ImageFrameCommonExtensions.ConvertDepthFrameToBitmap(image);
            return bytes.ToBitmapSource(image.Width, image.Height);

        }

        public static BitmapSource ToBitmapSource(this ColorImageFrame image)
        {
            if (image == null)
            {
                return null;
            }

            byte[] colorData = new byte[image.PixelDataLength];
            image.CopyPixelDataTo(colorData);

            return colorData.ToBitmapSource(image.Width, image.Height);



        }
    }
}
```

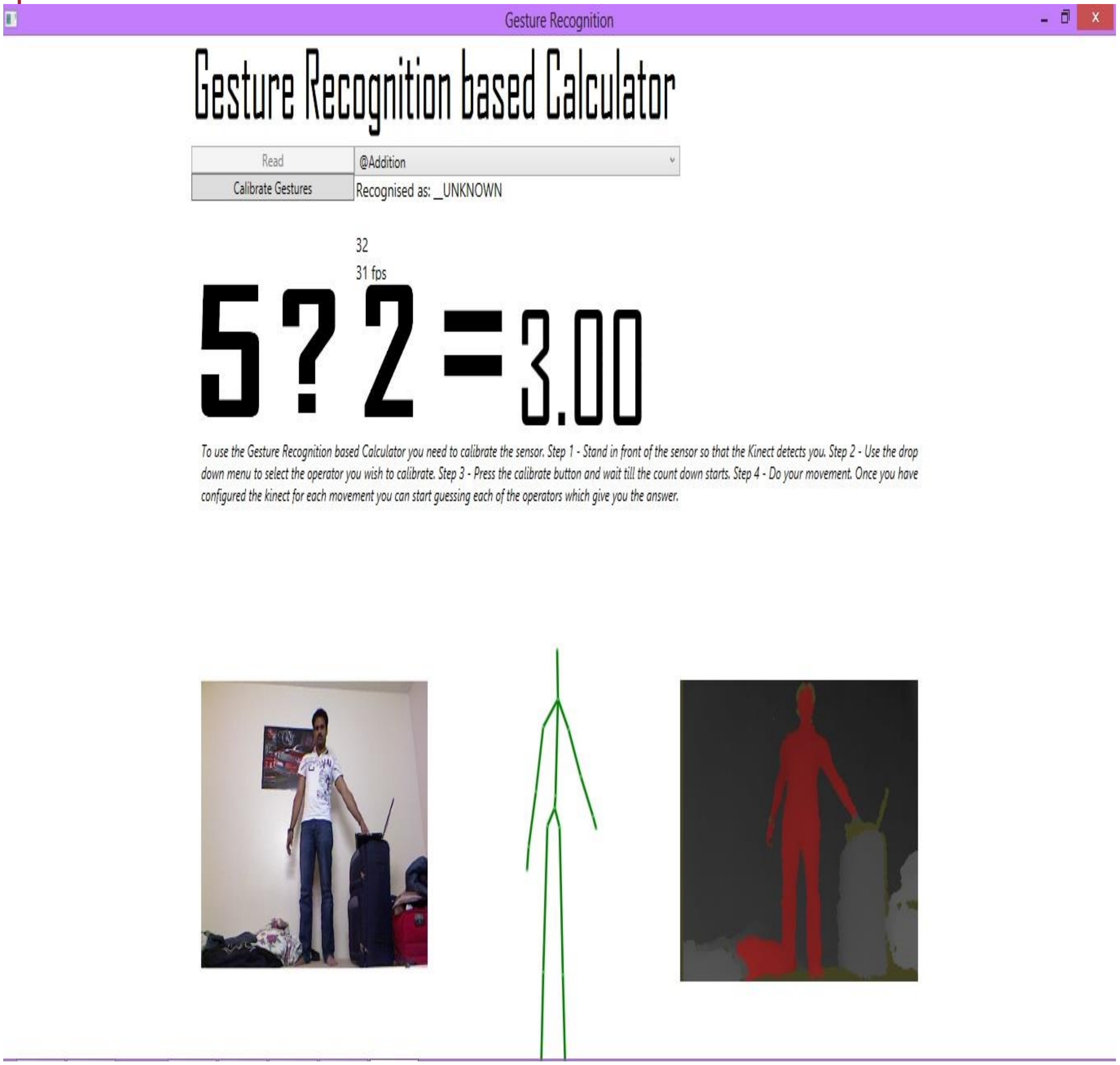
The Skeleten2DDataCoordEventArgs.cs file:

```
//-----------------------------------------------------------------------
// <copyright file="Skeleton2DdataCoordEventArgs.cs" company="Rhemyst and
Rymix">
//     Open Source. Do with this as you will. Include this statement or
//     don't - whatever you like.
//
//     No warranty or support given. No guarantees this will work or meet
//     your needs. Some elements of this project have been tailored to
//     the authors' needs and therefore don't necessarily follow best
//     practice. Subsequent releases of this project will (probably) not
//     be compatible with different versions, so whatever you do, don't
//     overwrite your implementation with any new releases of this
//     project!
//
//     Enjoy working with Kinect!
// </copyright>
// Gesture Recognition Based Calculator
// Modified by Sree Bharat Dasari
//-----------------------------------------------------------------------

namespace DTWGestureRecognition
{
    using System.Windows;

    /// <summary>
    /// Takes Kinect SDK Skeletal Frame coordinates and converts them intoo a
format useful to th DTW
    /// </summary>
    internal class Skeleton2DdataCoordEventArgs
    {
        /// <summary>
        /// Positions of the elbows, the wrists and the hands (placed from
left to right)
        /// </summary>
        private readonly Point[] _points;

        /// <summary>
        /// Initializes a new instance of the Skeleton2DdataCoordEventArgs
class
        /// </summary>
        /// <param name="points">The points we need to handle in this
class</param>
        public Skeleton2DdataCoordEventArgs(Point[] points)
        {
            _points = (Point[]) points.Clone();
        }

        /// <summary>
        /// Gets the point at a certain index
        /// </summary>
        /// <param name="index">The index we wish to retrieve</param>
        /// <returns>The point at the sent index</returns>
        public Point GetPoint(int index)
        {
            return _points[index];
        }

        /// <summary>
        /// Gets the coordinates of our _points
        /// </summary>
        /// <returns>The coordinates of our _points</returns>
        internal double[] GetCoords()
        {
            var tmp = new double[_points.Length * 2];
            for (int i = 0; i < _points.Length; i++)
            {
                tmp[2 * i] = _points[i].X;
                tmp[(2 * i) + 1] = _points[i].Y;
            }

            return tmp;
        }
    }
}
```

The Skeleten2DDataExtract.cs file:

```
//-----------------------------------------------------------------------
// <copyright file="Skeleton2DDataExtract.cs" company="Rhemyst and Rymix">
//     Open Source. Do with this as you will. Include this statement or
//     don't - whatever you like.
//
//     No warranty or support given. No guarantees this will work or meet
//     your needs. Some elements of this project have been tailored to
//     the authors' needs and therefore don't necessarily follow best
//     practice. Subsequent releases of this project will (probably) not
//     be compatible with different versions, so whatever you do, don't
//     overwrite your implementation with any new releases of this
//     project!
//
//     Enjoy working with Kinect!
// </copyright>
// Gesture Recognition Based Calculator
// Modified by Sree Bharat Dasari
//-----------------------------------------------------------------------

namespace DTWGestureRecognition
{
    using System;
    using System.Windows;
    using Microsoft.Kinect;

    /// <summary>
    /// This class is used to transform the data of the skeleton
    /// </summary>
    internal class Skeleton2DDataExtract
    {
        /// <summary>
        /// Skeleton2DdataCoordEventHandler delegate
        /// </summary>
        /// <param name="sender">The sender object</param>
        /// <param name="a">Skeleton 2Ddata Coord Event Args</param>
        public delegate void Skeleton2DdataCoordEventHandler(object sender,
Skeleton2DdataCoordEventArgs a);

        /// <summary>
        /// The Skeleton 2Ddata Coord Ready event
        /// </summary>
        public static event Skeleton2DdataCoordEventHandler
Skeleton2DdataCoordReady;

        /// <summary>
        /// Crunches Kinect SDK's Skeleton Data and spits out a format more
useful for DTW
        /// </summary>
        /// <param name="data">Kinect SDK's Skeleton Data</param>
        public static void ProcessData(Skeleton data)
        {
            // Extract the coordinates of the points.
            var p = new Point[6];
            Point shoulderRight = new Point(), shoulderLeft = new Point();
            foreach (Joint j in data.Joints)
            {
                switch (j.JointType)
                {
                    case JointType.HandLeft:
                        p[0] = new Point(j.Position.X, j.Position.Y);
                        break;
                    case JointType.WristLeft:
                        p[1] = new Point(j.Position.X, j.Position.Y);
                        break;
                    case JointType.ElbowLeft:
                        p[2] = new Point(j.Position.X, j.Position.Y);
                        break;
                    case JointType.ElbowRight:
                        p[3] = new Point(j.Position.X, j.Position.Y);
                        break;
                    case JointType.WristRight:
                        p[4] = new Point(j.Position.X, j.Position.Y);
                        break;
                    case JointType.HandRight:
                        p[5] = new Point(j.Position.X, j.Position.Y);
                        break;
                    case JointType.ShoulderLeft:
                        shoulderLeft = new Point(j.Position.X, j.Position.Y);
                        break;
                    case JointType.ShoulderRight:
                        shoulderRight = new Point(j.Position.X,
j.Position.Y);
                        break;
                }
            }

            // Centre the data
            var center = new Point((shoulderLeft.X + shoulderRight.X) / 2,
(shoulderLeft.Y + shoulderRight.Y) / 2);
            for (int i = 0; i < 6; i++)
            {
                p[i].X -= center.X;
                p[i].Y -= center.Y;
            }

            // Normalization of the coordinates
            double shoulderDist =
                Math.Sqrt(Math.Pow((shoulderLeft.X - shoulderRight.X), 2) +
                          Math.Pow((shoulderLeft.Y - shoulderRight.Y), 2));
            for (int i = 0; i < 6; i++)
            {
                p[i].X /= shoulderDist;
                p[i].Y /= shoulderDist;
            }
            // Launch the event!
            Skeleton2DdataCoordReady(null, new
Skeleton2DdataCoordEventArgs(p));
        }
    }
}
```

SCREENSHOT



Gesture Recognition based Calculator

Recognised as: _UNKNOWN

5 ? 2 = 3.00

To use the Gesture Recognition based Calculator you need to calibrate the sensor. Step 1 - Stand in front of the sensor so that the Kinect detects you. Step 2 - Use the drop down menu to select the operator you wish to calibrate. Step 3 - Press the calibrate button and wait till the count down starts. Step 4 - Do your movement. Once you have configured the kinect for each movement you can start guessing each of the operators which give you the answer.