# Body Pose Tracking and Instrumented Personal Training using Kinect

Michael Sherman

Capstone Project Report for BSEE

Rutgers University

## Table of Contents

## Table of Figures

## Introduction

In most, if not all physical endeavors, proper form is essential for effective and safe training. However, it is often difficult to figure out by oneself. Tutorials are available in texts and online, but a limited perception of body position limits use. A coach or trainer can observe movements and correct as needed, providing this information, but their services can be time consuming or expensive to obtain. This causes high dropout rates among beginners lacking a way to get over their initial knowledge gap.

The benefit provided by automated real-time feedback eliminates the interruption to training from document based learning and the limited coaching availability. Consumer electronics has created a technology base and opportunity for development of these tools. One application of this new infrastructure to personal training is the focus of this report.

Application examples include GPS running watches and cycling computers. These track route taken, distance, speed, elevation, cadence, and heart rate. Athletes use this to chart their performance over time and adjust their training for maximal effectiveness. These systems also have a large market, because of their utility, low cost, and ease of use.

This market is untapped for weight training. The only extant system for weightlifting relies on optical tracking of markers placed on the joints, and is only used at an elite level. (Wm A. Sands, 2008) The system is very expensive (~$20k), and requires trained personnel to attach the markers and operate. Markerless systems require multiple cameras, can be difficult to set up, and the cheapest one ($600) doesn't operate in real time. (METAmotion), This is too high a cost threshold for most participants in the sport, especially beginners.

Recent advances in low cost computer vision offer an alternative. This report explores using the Microsoft Kinect to do full body motion tracking. The Kinect is cheap (~$100) consumer hardware, and while far less accurate than the optical tracking systems, is sufficient for this application. In the feasibility study reported here, the Kinect was connected to a laptop and used to obtain joint positions and angles. These were then judged for correctness, and feedback provided to the user.

The challenge in this effort was in creating a software framework using the Kinect software development kit (SDK), in which data from the Kinect is input to decision rules that are the basis of corrections given to the user.

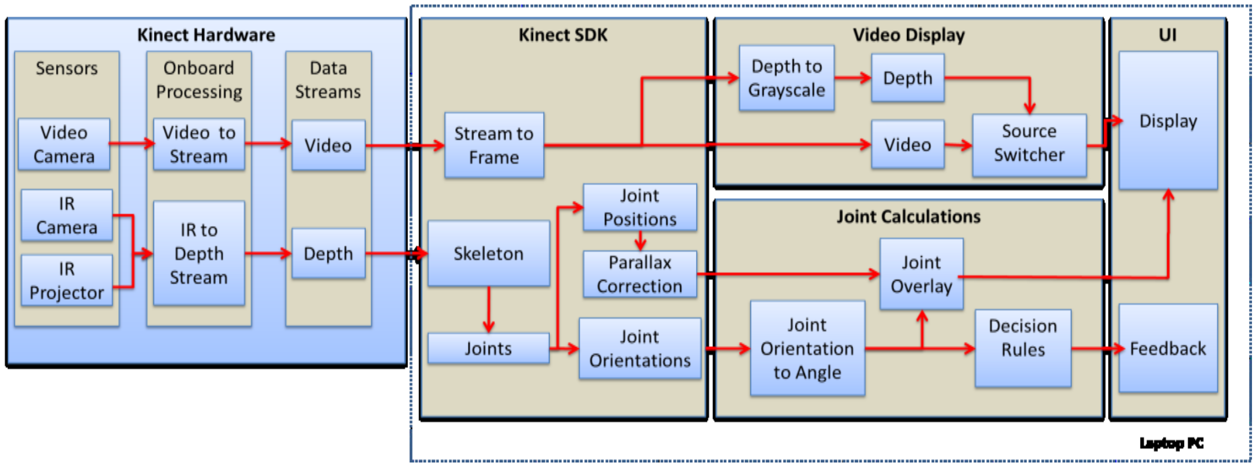


Figure 1  Block diagram of Kinect-based sports motion trainer

The functional block diagram in Figure 1 gives an overview of the pose tracking and feedback system. The Kinect streams video and depth data via USB to a laptop. Functions from the Kinect user accessible software development kit (SDK) are used to extract the angle of each joint, which is then passed to a set of decision rules. A visual marker of each estimated joint is then overlaid on top of the video stream. Finally, the decision rule's output is displayed to the user as visual feedback on a laptop display. The data flow from the Kinect to the laptop and pose computations are performed on each frame capture, at a rate of 30 frames per second.

The next section describes the hardware used, followed by PC hosted data processing, display, and user feedback. Results of feasibility tests with this system will be given, followed by conclusions and future directions for enhancements to the Kinect's capability for sports training.

## Hardware

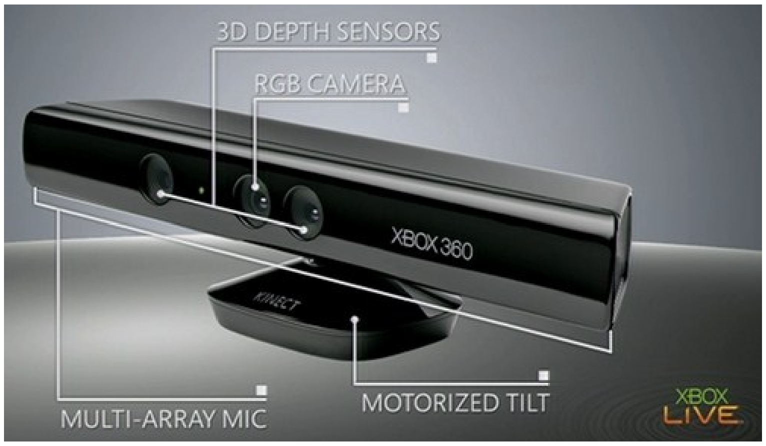


Figure 2 Diagram of the Microsoft Kinect, showing the major features.

## Early stages

The initial concept was to use inertial sensors to track the path of a barbell. This used the assumption that a vertical path was optimal for mechanical efficiency, and that deviations were errors in form. It was suggested to use a Microsoft Kinect to supplement the data, and the Kinect has since become the primary data source, tracking joint angles as opposed to directly tracking the bar path.

The initial concept used an android phone for processing, but these were found to have insufficient processing power, as well as lacking support for the Kinect Libraries. The current code relies on a laptop to provide real time joint estimation. A high performance CPU is needed for the calculations to fit a skeleton model to the data at 30 frames per second. In order to capture fast motions clearly, the frame rate needs to be as high as possible, but is limited by the Kinect's capture rate.

## Current Design

The current design uses only the Kinect for data acquisition, and a Windows laptop for processing and display.

The Kinect (ref. Figure 1) consists of three main parts, an infrared projector, an IR camera, and a color camera. The IR camera looks at a pattern of dots from the IR projector, and uses the displacement from a known pattern to find the depth of each pixel. This processing is done onboard the Kinect.

The color and depth values are then streamed to a laptop using a dedicated USB controller within the laptop, to ensure sufficient bandwidth. Once there, they are assembled into a point cloud. All further processing is done onboard the laptop. Full requirements are under System Requirements in the Appendix.

Using only the Kinect for data acquisition simplifies the hardware to the point where no further hardware development is necessary. The development work done by Microsoft and PrimeSense has resulted in the software development kit (SDK) for high level processing of the depth and point cloud data.

This project developed a software framework using the SDK functions that processed this point cloud and extracted joint angles, passing them to decision rules. The C# code for this is under Code on page 14.

# Software

As mentioned earlier, the software was developed in C# using Visual Studio 2012, .NET framework 4.0, and Kinect SDK version (1.0.3.191). Full details are available under System Requirements on 13.The software framework is broken down into three main sections: Data Acquisition, Decision Rules, and Display. These are described below.

## Data Acquisition

The Kinect provides several streams of data, a video stream, a depth stream, and a skeleton stream. The video stream is the feed from the VGA camera, while the depth stream is an array of depth values for each pixel. These depth values are taken from the displacement between a known IR pattern, and the pattern observed after projection. The skeleton stream consists of the positions and other data for each of twenty two body parts, as estimated by a classifier. (Jamie Shotton, 2011)

These positions are then analyzed to get the orientation of each bone, and the angles between bones. The orientations are made available referenced either to the connected joints, or to the Kinect sensor position. They are also available formatted either as rotation matrices, or as quaternions. For this application, only the magnitude of the angle between the bones is used, not their 3D orientation. This is taken from the quaternion representation with the equation below.

$$\theta = \left(\frac{180}{\pi}\right) * 2 * \cos^{-1}(W)$$

Equation 1

The quaternion representing the rotation of angle $\theta$ is of the form (w,x,y,z), where W represents the magnitude of the rotation, around the axis defined by vector (x,y,z). Equation 1 above shows the relationship between W and angle $\theta$ in degrees. (Vicci, 2001)

Data can also be logged to a file. This is used in the development of the decision rules (described below). The angle magnitude for each joint is saved to file, along with the start and end joints used in the angle calculation. The format is described under Log format on page 12.

The Kinect has built in smoothing algorithms that automatically filter data across frames. These were disabled, but a no noticeable difference was seen in the logged results.

## Decision Rules

The decision rules govern how the software determines if the measured angles represent correct form. The rule implemented in this demonstration only checks whether the returned angles fall within a specific range. These decision rules are specific for each exercise. New rules can either be developed to add a new exercise to the training system or to refine the joint angle thresholds or training goals (set of joint angles) for a current one.

The decision rule implemented here is for a one handed curl. It checks whether excessive motion of the hips is seen, indicative of poor form through use of uninvolved body parts to assist in the exercise. This is done through simple thresholding, detecting whether the hip angles are within a set range.

## Display

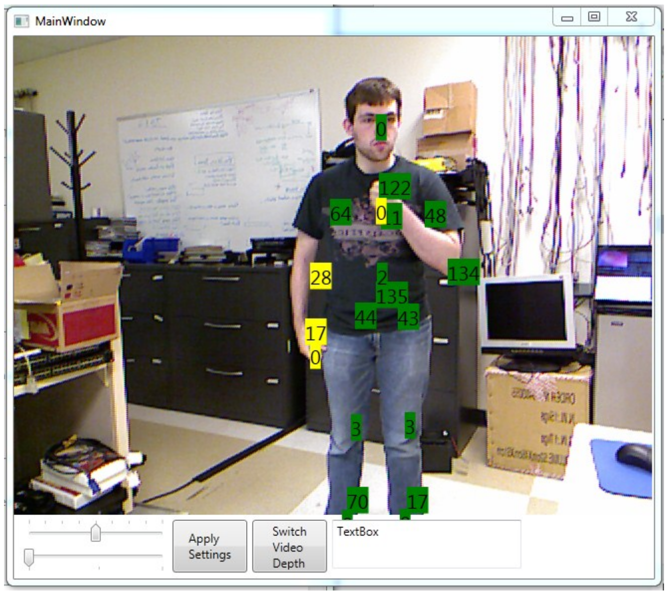
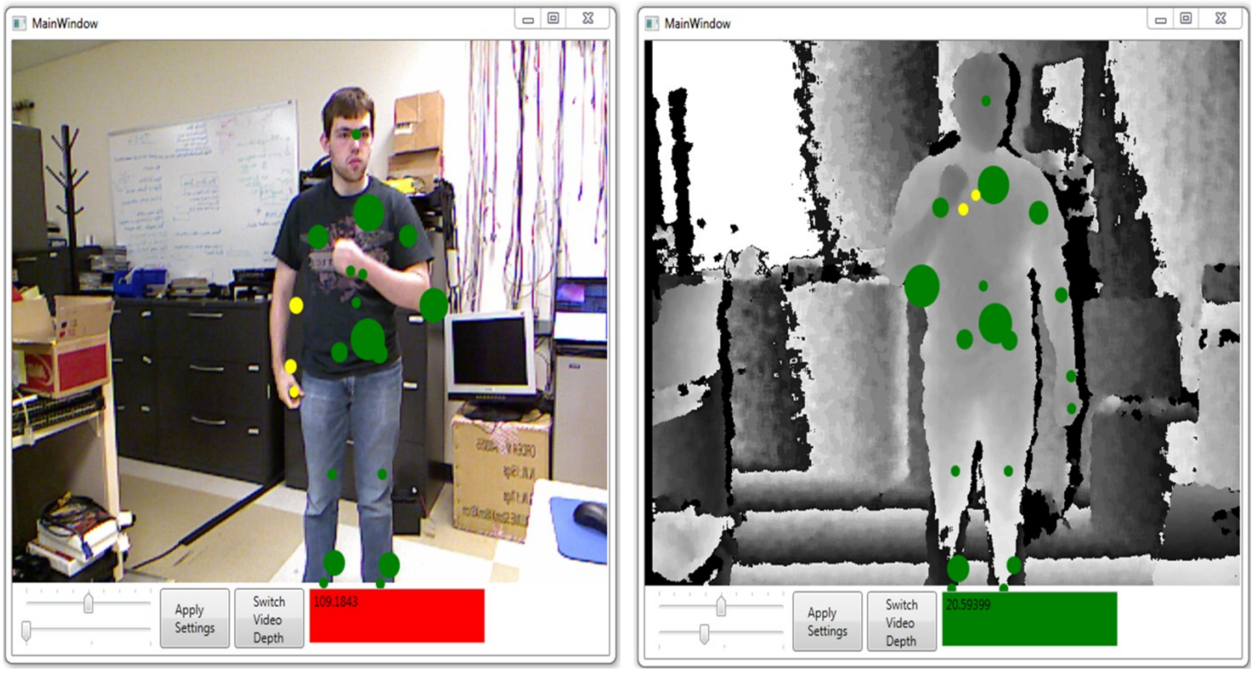


Figure 3 Screenshot of Joint Overlay

The software displays the angle of each joint, overlaid on top of the video stream. The position estimated for the joint is used, corrected for the parallax between the IR and video cameras. This is accomplished with a function included in the SDK that corrects for the change in viewing angle caused by the distance between the two sensors.

The primary function of these estimated joint angles is for use in the development of decision rules. In Figure 3 above, the estimated joint angles are shown on top of the video stream. The color coding (green/yellow) shows whether the Kinect is tracking the joint directly, or inferring its position due to occlusion, respectively.



109.1843

20.59399

**Figure 4 Color vs. Depth, UI example**

In Figure 4 above, an alternate display is shown. Here, the circles on top of each joint change in size to represent the magnitude of the joint angle. The color coding for the overlay is as before. This acts as simple visual feedback, to show the user that system is tracking. A text box below the image displays the angle being evaluated by the currently active decision rule, with the background color corresponding to the outcome. Red represents a violated rule, green for proper angles. Once more rules are implemented, the red coloration will be applied to the overlay marker in question, instead of the box below the image.

The video display can be switched to a greyscale depth image. In the depth image, each pixel's color is the depth value mapped onto 16 bit greyscale. This gray scale body display highlights pose vs. e.g. clothing features. Contrast can be adjusted by changing a multiplier on the depth value, before the modulus is taken with respect to the max value of the grayscale, in effect wrapping the value around. The distance between 0 and the minimum depth value is used for the color representing no valid depth value. This can arise from being too close, or shadowing the projected pattern. The range between the maximum depth value and 65535 is used for a distance that is too large. All of these have been adjusted for optimal viewing contrast.

## Results

This section shows plots of elbow and hip motion during a curl exercise and demonstrates the tool's ability to discriminate between good and poor form.

Figure 5 below, shows joint angle tracking both during large motions and periods of rest. The tracked motion is a two arm curl with left and right arms. In the second half of the plot, the arms are stationary, held to the sides. It's important to note that the joint angles are referenced to the Kinect's joint hierarchy, (Figure 8 in the Appendix) and do not relate to the absolute position of the arm in free space.

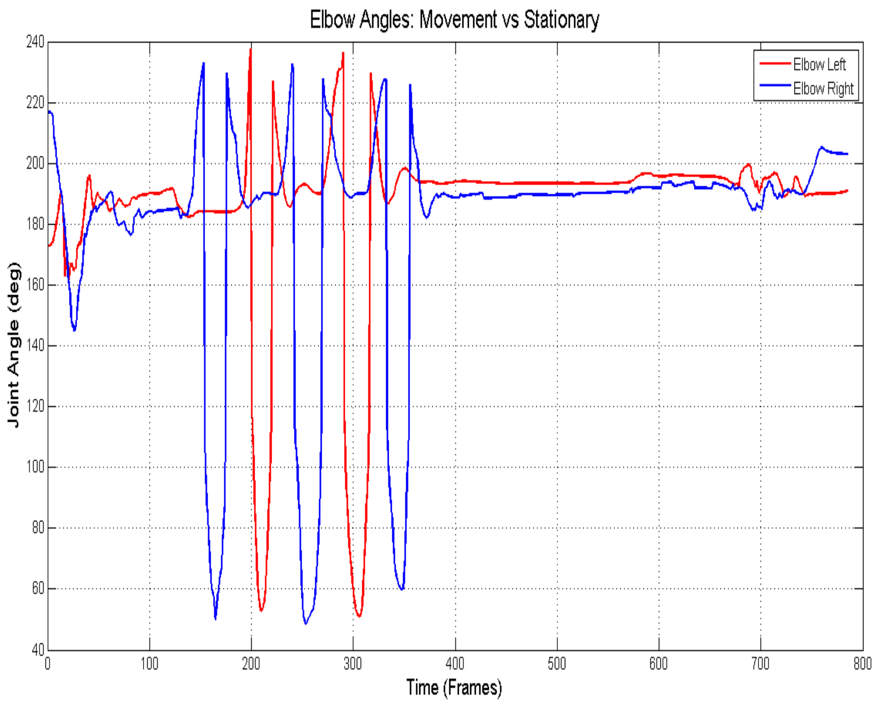


Figure 5 Curl motion elbow angles

Figure 6 and Figure 7 below show changes in hip and elbow angles that appear with the addition of weight to the curl.

With no weight, the hip angles stay constant to within a few degrees, and the elbow angles present smooth changes. This demonstrates what a joint uninvolved in the particular exercise should look like. After excess weight is added to the movement, extraneous motion is seen in the hips. This signifies that the hips are now active in the movement, and represents a form error. Additionally, the loss of smooth, repeatable motion is seen in the elbow angles, showing a loss of control.

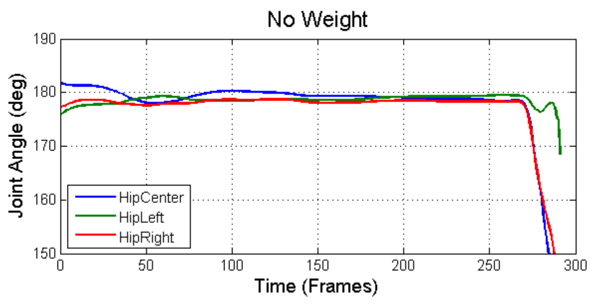


Figure 6 Hip angles

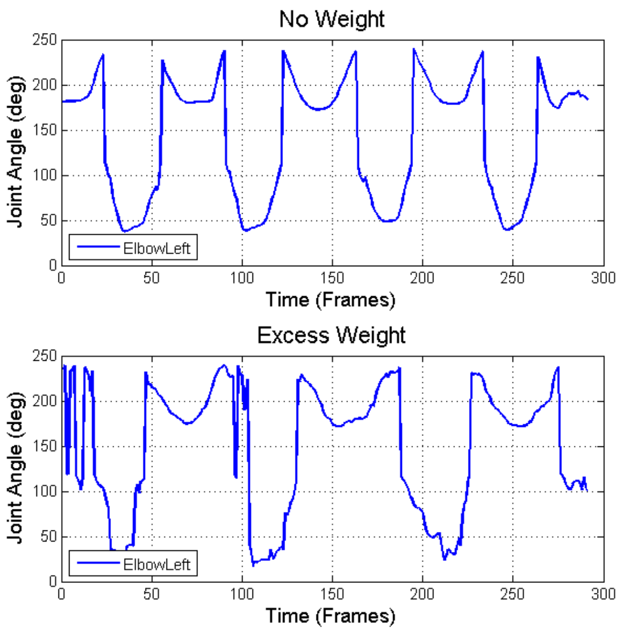


Figure 7 Elbow Angles

## Conclusion

This effort has been a proof of concept to demonstrate that the Kinect's accuracy is sufficient for a sports training application. It also is an application example of a framework for exercise. Form errors show up very clearly in the graphs generated by logging joint angles. The test case of detecting when a one armed curl is attempted with too much weight was a success, with the system informing the user based on the detected variation in hip angles.

Future improvements can again be broken into the three sections of Data Acquisition, Decision, and Feedback.

One difficulty is of creating decision rules that describe an exercise well, and are robust for different body types and proportions. Future improvements include looking at joint trajectories derived from correlating motion across multiple frames, and using full joint orientation. Ultimately, a learning algorithm could be implemented, neural networks or fuzzy logic for example, reducing the programming burden when criteria are poorly understood, or complex.

To further fulfill the idea of emulating a coach at home, features such as tracking reps and sets, as well as other statistics such as speed of lift or rest time between sets could be added. It could be used as an exercise planner, verbally walking the user through a workout regimen, but keeping track of the user's actions, as opposed to having them attempt to keep up with an exercise video.

The system could be improved with the addition of more detailed feedback, displaying which body part is in error. This, along with displaying the movement needed to correct pose, could be accomplished by drawing the bones between joints, and color coding for correctness, and arrows to indicate movement direction. There is the question of whether visual feedback is optimal. The display oriented feedback could be distracting, e.g. looking at a screen would impair correct head angle during a squat exercise. Verbal cues could be more useful, such as "Stand up straighter", or an instruction to move a given limb in a particular direction.

If higher bandwidth pose tracking is needed for training, inertial sensors could be added to the Kinect's dataflow. It may be sufficient to put the additional instrumentation on the barbell, avoiding the inconvenience of mounting sensors on the user. This also addresses the obstacle of occlusion, without adding multiple cameras. Inertial sensors could improve tracking accuracy on parameters such as bar path, acceleration, rotation, rest times, and could indicate performance improvement.

# Appendices

## Log format

| | | | |
|---|---|---|---|
| Start Joint of angle 1 | Start Joint of angle 2 | ... | Start Joint of angle 22 |
| End Joint of angle 1 | End Joint of angle 2 | ... | End Joint of angle 22 |
| frame 1 angle 1 value | frame 1 angle 2 value | ... | frame 1 angle 22 value |
| frame 2 angle 1 value | frame 2 angle 2 value | ... | frame 2 angle 22 value |
| ... | ... | ... | ... |
| frame n angle 1 value | frame n angle 2 value | ... | frame n angle 22 value |

Table 1 Format of Data Logs

## Joint hierarchy

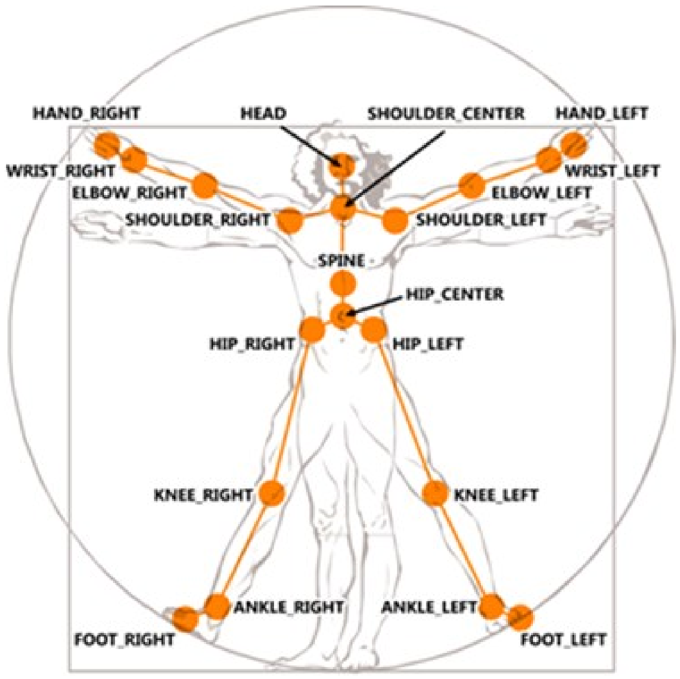


Figure 8 Hierarchy of Tracked Joints

## System Requirements

Taken from http://www.microsoft.com/en-us/kinectforwindows/develop/overview.aspx

(Microsoft, 2012)

Hardware

- 32 bit (x86) or 64 bit (x64) processor
- Dual-core 2.66-GHz or faster processor
- Dedicated USB 2.0 bus
- 2 GB RAM
- A Kinect for Windows sensor, which includes special USB/power cabling

## Works Cited

Jamie Shotton, A. F. (2011). Real-Time Human Pose Recognition in Parts from a Single Depth Image. *CVPR.* IEEE.

METAmotion. (n.d.). *iPi Desktop Motion Capture™ - FAQ*. Retrieved September 24, 2012, from metamotion.com: http://www.metamotion.com/software/iPiSoft/Markerless-Motion-Capture_iPiSoft-iPi_Studio_FAQ.html

Microsoft. (2012, May 2). *Kinect System Requirements*. Retrieved September 25, 2012, from Microsoft.com: http://www.microsoft.com/en-us/kinectforwindows/develop/overview.aspx

Organic Motion, Inc. (n.d.). *Openstage Realtime Markerless Motion Capture*. Retrieved September 24, 2012, from Organic Motion: http://www.organicmotion.com/products/openstage

Vicci, L. (2001, April 27). *Quaternions and Rotations in 3-Space: The Algebra and its Geometric Interpretation.* Retrieved September 25, 2012, from UNC Chapel Hill Tech Reports: http://www.cs.unc.edu/techreports/01-014.pdf

Wm A. Sands, P. (2008, March 04). *Assisting Olympic Weightlifting via Motion Tracking at the U.S. Olympic Training Center.* Retrieved September 24, 2012, from Polhemus: http://www.polhemus.com/polhemus_editor/assets/USOCWeightLiftingElbow.pdf

## Code

```
namespace WPFtut
{
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Text;
    using System.Threading.Tasks;
    using System.Windows;
    using System.Windows.Controls;
    using System.Windows.Data;
    using System.Windows.Documents;
    using System.Windows.Input;
    using System.Windows.Media;
    using System.Windows.Media.Imaging;
    using System.Windows.Navigation;
    using System.Windows.Shapes;
    using Microsoft.Kinect;
    using System.Globalization;
    using System.IO;

    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        //variables for sensor
        private KinectSensor sensor;
```

```
// Variables for color video
private WriteableBitmap colorBitmap;
private byte[] colorPixels;

//variables for depth video
private short[] depthPixels;
private ushort[] depthcolorPixels;
private WriteableBitmap depthcolorBitmap;

//variables for skeleton tracking
Skeleton[] skeletons = new Skeleton[0];

//Variables for logging
string savePath = "C:\\Users\\mike\\Documents\\Kinect Capstone\\LogFiles\\";
string fileName = "Test.csv";
string saveName = "";
bool logging = false;
bool firstLine = true;

//Init Window
public MainWindow()
{
   InitializeComponent();
}

//Setup when window loaded
private void Window_Loaded_1(object sender, RoutedEventArgs e)
{

   this.sensor = KinectSensor.KinectSensors[0];  //get sensor
```

```
        //Setup Video Stream and event handler
        this.sensor.ColorStream.Enable(ColorImageFormat.RgbResolution640x480Fps30);
        this.colorPixels = new byte[this.sensor.ColorStream.FramePixelDataLength];
        this.colorBitmap = new WriteableBitmap(this.sensor.ColorStream.FrameWidth,
this.sensor.ColorStream.FrameHeight, 96.0, 96.0, PixelFormats.Bgr32, null);
        this.image.Source = this.colorBitmap;
        //this.sensor.ColorFrameReady += this.sensor_ColorFrameReady;

        //Setup depth stream and event handler
        this.sensor.DepthStream.Enable(DepthImageFormat.Resolution640x480Fps30);
        this.depthPixels = new short[this.sensor.DepthStream.FramePixelDataLength];
        this.depthcolorPixels = new ushort[this.sensor.DepthStream.FramePixelDataLength *
sizeof(int)];
        this.depthcolorBitmap = new WriteableBitmap(this.sensor.DepthStream.FrameWidth,
this.sensor.DepthStream.FrameHeight, 96.0, 96.0, PixelFormats.Gray16, null);
        //this.sensor.DepthFrameReady += sensor_DepthFrameReady;

        this.sensor.AllFramesReady += sensor_AllFramesReady;

        //Setup skeleton stream and event handler
        TransformSmoothParameters smoothParam = new TransformSmoothParameters()
        {
          Correction = 0.5f,
          JitterRadius = 0.05f,
          MaxDeviationRadius = 0.04f,
          Prediction = 0.5f,
          Smoothing = 0.5f
        };

        this.sensor.SkeletonStream.Enable(smoothParam);
        //this.sensor.SkeletonFrameReady += sensor_SkeletonFrameReady;
```

```
    //Start sensor
    this.sensor.Start();
}

void sensor_AllFramesReady(object sender, AllFramesReadyEventArgs e)
{
    using (ColorImageFrame colorFrame = e.OpenColorImageFrame())
    using (DepthImageFrame depthFrame = e.OpenDepthImageFrame())
    using (SkeletonFrame skeletonFrame = e.OpenSkeletonFrame())
    {
        RenderColor(colorFrame); //get and show color video
        RenderDepth(depthFrame); //get and show depth video

        this.skeletons = GetSkeletons(skeletonFrame); //get skeleton

        RenderJoints(this.skeletons); // render joints
    }
}

private void logButton_Click(object sender, RoutedEventArgs e)
{
    if (logging == false)
    {
        logging = true;
        logButton.Content = "Stop Log";
        fileName = textBox.Text;
        saveName = savePath + fileName + ".csv";



    }
    else if (logging == true)
    {
        logging = false;
        logButton.Content = "Start Log";
    }
}


private void LogJoint(Skeleton skelCheck)
{
    double[] angArr = new double[skelCheck.BoneOrientations.Count];
    string angString = null;
    string jointString = null;
    string jointEndString = null;
    if (logging == true)
    {
        foreach (BoneOrientation boneAng in skelCheck.BoneOrientations)
        {
            float angVal = 0;
            angVal = boneAng.AbsoluteRotation.Quaternion.W;
            double angValDeg = (2 * Math.Acos(angVal) * (180 / Math.PI));
            //angArr[i++] = angValDeg;
            angString = (angString + angValDeg.ToString() + ",");
            jointString = (jointString + boneAng.StartJoint.ToString() + ",");
            jointEndString = (jointEndString + boneAng.EndJoint.ToString() + ",");
        }


        using (StreamWriter file = new StreamWriter(saveName,true))
        {
            if (firstLine == true)
            {
                file.WriteLine(jointString);
                file.WriteLine(jointEndString);
                firstLine = false;
            }
            file.WriteLine(angString);
        }
    }
}

//Stop sensor on window close
private void Window_Closing_1(object sender,
System.ComponentModel.CancelEventArgs e)
{
    //this.sensor.Stop();
    this.sensor.Dispose(); //stops all data streaming, trying to avoid errors from unhandled
streams
}



private void RenderJoints(Skeleton[] skeletons)
{
    canvas.Children.Clear();

    if (skeletons.Length != 0)
    {
        foreach (Skeleton skel in skeletons)
        {
            if (skel.TrackingState == SkeletonTrackingState.Tracked)
            {
                //ElbowAngleCheck(skel);
                HipAngleCheck(skel);
```

```
                    LogJoint(skel);
                    foreach (Joint joint in skel.Joints)
                    {
                        Brush drawBrush = null;

                        if (joint.TrackingState == JointTrackingState.Tracked)
                        {
                            drawBrush = Brushes.Green;
                        }
                        else if (joint.TrackingState == JointTrackingState.Inferred)
                        {
                            drawBrush = Brushes.Yellow;
                        }

                        if (drawBrush != null)
                        {
                            DepthImagePoint depthPoint =
this.sensor.MapSkeletonPointToDepth(joint.Position,
DepthImageFormat.Resolution640x480Fps30);
                            ColorImagePoint colorPoint =
this.sensor.MapSkeletonPointToColor(joint.Position,
ColorImageFormat.RgbResolution640x480Fps30);

                            Point drawPoint = new Point(depthPoint.X, depthPoint.Y);

                            if (this.image.Source == this.colorBitmap)
                            {
                                drawPoint.X = colorPoint.X;
                                drawPoint.Y = colorPoint.Y;
                            }

                            if ((drawPoint.X <= 640) & (drawPoint.Y <= 480)) //try to avoid drawing out
of frame
```

```
                        {
                            Ellipse el = new Ellipse();
                            TextBlock jointText = new TextBlock();
                            double angle = 0;
                            int size = 10;
                            foreach (BoneOrientation boneAng in skel.BoneOrientations)
                            {
                                if (boneAng.StartJoint != boneAng.EndJoint)
                                {
                                    if (boneAng.StartJoint == joint.JointType)
                                    {
                                        //size =
SetJointSizeAngle(boneAng.HierarchicalRotation.Quaternion);
                                        angle = (double)(2 *
Math.Acos(boneAng.HierarchicalRotation.Quaternion.W)*(180 / Math.PI));
                                        angle = Math.Round(angle, 0);
                                    }
                                }
                            }

                            el.Width = 5;
                            el.Height = 5;
                            el.Fill = drawBrush;

                            jointText.Text = Convert.ToString(angle);
                            jointText.Background = drawBrush;
                            jointText.FontSize = 20;
                            Canvas.SetLeft(jointText, drawPoint.X);
                            Canvas.SetTop(jointText, drawPoint.Y);



                            Canvas.SetLeft(el, drawPoint.X);
                            Canvas.SetTop(el, drawPoint.Y);
                            canvas.Children.Add(el);
                            canvas.Children.Add(jointText);
                        }

                    }
                }
            }
        }
    }
}

private Skeleton[] GetSkeletons(SkeletonFrame skeletonFrame)
{
    Skeleton[] skeletonsArr = new Skeleton[0];
    if (skeletonFrame != null)
    {
        skeletonsArr = new Skeleton[skeletonFrame.SkeletonArrayLength];
        skeletonFrame.CopySkeletonDataTo(skeletonsArr);
    }
    return skeletonsArr;
}

private void ElbowAngleCheck(Skeleton skel)
{
    foreach (BoneOrientation boneAng in skel.BoneOrientations)
    {
        if (boneAng.StartJoint == JointType.ElbowRight)
        {
            Vector4 jointAng = boneAng.HierarchicalRotation.Quaternion;
            float angle = (float)(2 * Math.Acos(jointAng.W));
            float angleDeg = (float)(angle * (180 / Math.PI));
            if (angleDeg < 90)
            {
                textBox.Background = Brushes.Green;
            }
            else
            {
                textBox.Background = Brushes.Red;
            }
            textBox.Text = angleDeg.ToString();
        }
    }
}

private void HipAngleCheck(Skeleton skel)
{
    float angleDeg = -1;
    int outOfbounds = 0;
    foreach (BoneOrientation boneAng in skel.BoneOrientations)
    {
        if (boneAng.StartJoint == JointType.HipCenter)
        {
            if ((boneAng.EndJoint == JointType.HipRight))
            {

                Vector4 jointAng = boneAng.HierarchicalRotation.Quaternion;
                float angle = (float)(2 * Math.Acos(jointAng.W));
                angle = (float)(angle * (180 / Math.PI));
                angleDeg = angle;
                if ((angle > 136) || (angle < 130))
                {
```

```
                outOfbounds = 1;
            }
        }
    }
}
if ((outOfbounds == 0) && (angleDeg >=0))
{
    textBox.Background = Brushes.Green;
}
else
{
    textBox.Background = Brushes.Red;
}
textBox.Text = angleDeg.ToString();
}

private int SetJointSizeDepth(DepthImagePoint dPoint)
{
    int depth = 4096 - dPoint.Depth;
    int dia = 10;
    if (depth > -1)
    {
        dia = 1 + depth/200;
    }
    return dia;
}

private int SetJointSizeAngle(Vector4 quat)
{
    float angle = (float)(2 * Math.Acos(quat.W));
    float angleDeg = (float)(angle * (180 / Math.PI));
```

```
            int dia = (int)(10 + angleDeg / 5);
            return dia;
        }

        //void sensor_DepthFrameReady(object sender, DepthImageFrameReadyEventArgs e)
        //{
        //    using (DepthImageFrame depthFrame = e.OpenDepthImageFrame())
        //    {
        //        RenderDepth(depthFrame);
        //    }
        //}

        private void RenderDepth(DepthImageFrame depthFrame)
        {
            if (depthFrame != null)
            {
                depthFrame.CopyPixelDataTo(this.depthPixels);//get pixels from depth frame

                this.depthcolorPixels = ScaleDepth(this.depthPixels);

                this.depthcolorBitmap.WritePixels(
                    new Int32Rect(0, 0, this.depthcolorBitmap.PixelWidth,
this.depthcolorBitmap.PixelHeight),
                    this.depthcolorPixels,
                    this.depthcolorBitmap.PixelWidth * sizeof(short),
                    0); // after loop done, write new pixel array to screen
            }
        }

        private ushort[] ScaleDepth(short[] depthPixels)
        {
            int colorPixelIndex = 0;//index for pixels to display
        int maxGrey = ushort.MaxValue; //get max value for display
        int topBuffer = 5000;// set distance from white
        int bottomBuffer = 5000;// set distance from black
        int mod = maxGrey - topBuffer - bottomBuffer; //calculate upper bound of modulo
        int contrastMult = 16; //map 4096 values onto 65535 values
        int contrastRepeat = (int)sliderRep.Value; //number of times to loop over whole range

        for (int i = 0; i < depthPixels.Length; i++)
        {
          //shift bits to remove playerID from first 3
          short depth = (short)(depthPixels[i] >> DepthImageFrame.PlayerIndexBitmaskWidth);
          ushort intensity = 0; //initialize intensity

          if (depth == -1) //If depth unknown, set color to black
          {
            intensity = (ushort)0;
          }
          else if (depth == 4095) //If depth too far, set color to black
          {
            intensity = (ushort)65535;
          }
          else
          {
            intensity = (ushort)(depth); //convert short to ushort, values of depth will be in range
800-4000
            intensity = (ushort)(intensity * contrastMult); //scale depth values to fill whole range
in grey16 65535 values
            intensity = (ushort)(bottomBuffer + ((intensity * contrastRepeat) % mod)); //apply
modulo, scale smaller depth range for contrast
          }
          this.depthcolorPixels[colorPixelIndex++] = intensity; //set current pixel to display to
intensity, increment pixel
        }
```

```
    return depthcolorPixels;
}







//void sensor_ColorFrameReady(object sender, ColorImageFrameReadyEventArgs e)
//{
//    using (ColorImageFrame colorFrame = e.OpenColorImageFrame())
//    {
//        RenderColor(colorFrame);
//    }
//}

private void RenderColor(ColorImageFrame colorFrame)
{
    if (colorFrame != null)
    {
        colorFrame.CopyPixelDataTo(this.colorPixels);

        this.colorBitmap.WritePixels(
            new Int32Rect(0, 0, this.colorBitmap.PixelWidth, this.colorBitmap.PixelHeight),
            this.colorPixels,
            this.colorBitmap.PixelWidth * sizeof(int),
            0);// after loop done, write new pixel array to screen
    }
}

private void button_Click(object sender, RoutedEventArgs e)
{
```

```
            if (Math.Abs(sensor.ElevationAngle) < 21)
            {
                sensor.ElevationAngle = (int)sliderAngle.Value;
            }

        }

        private void button1_Click(object sender, RoutedEventArgs e)
        {
            if (this.image.Source == this.depthcolorBitmap)
            {
                this.image.Source = this.colorBitmap;
            }
            else if (this.image.Source == this.colorBitmap)
            {
                this.image.Source = this.depthcolorBitmap;
            }
        }


    }
}
```