DEVELOPER MANUAL

Kinecticons

Version 2.2

May 5, 2013

TEXAS CHRISTIAN UNIVERSITY

Revision History

The following is a history of document revisions.

Version	Changes	Edited
Version 1.0	Initial draft	2/19/2013
Version 2.0	Updated to version 2, made revisions	2/19/2013
Version 2.1	Corrected some spelling mistakes. Filled in most sections, detailing how to set up a project to use the Kinect, as well as add on to our program.	4/24/2013
Version 2.2	Corrected grammatical errors and section labeling. Updated Add Injury Window description. Added description of database into section 2.4	5/05/2013

Revision Sign-off

By signing the following, the team member asserts that he has read the entire document and has, to the best of his knowledge, found the information contained herein to be accurate, relevant, and free of typographical error.

Name	Signature	Date
Davis Farish		
Scott Grace		
Kyle Sarantsev		
Chris Walton		
Chris Witter		

Table of Contents

1. Introduction

1.1. Purpose

The purpose of this document is to assist new developers in installing and setting up Therapy Kinection in order to make changes or expansions to the system. The separate parts of the system will be outlined and key components detailed for the developer's benefit.

1.2. Overview of Document

The document contains the following sections:

Section 2 – System Overview:

Section 3 – Development Getting Started

Section 4 – Database Development

Section 5 – Desktop Program Development

Section 6 – Glossary of Terms

Section 7 – Appendix

2. System Overview

2.1. System Components

The Therapy Kinection system consists of three components; the desktop program, database, and the Kinect. These components are described below.

2.2. Therapy Kinection Desktop Program

The Therapy Kinection desktop program is the tool that uses the Kinect and a database. All interactions with the database occur through the desktop program. While the Kinect is tracking a user for the Vail Sport Test, a visual representation of the patient is shown on the window, as well as detailed measurements calculated from data coming from the Kinect.

2.3. Kinect

The Kinect allows the Therapy Kinection desktop program to grade the Vail Sport Test for any user that steps in front of the device. It is connected via USB, with its own power source. The Kinect tracks 20 joints on the human body at a frequency of 30 Hz. The 3D position, as well as tracking state, of each of the joints can then be analyzed in the program.

2.4. Database

Our database, which is stored locally, plays a huge role in our system. It holds all of the patient data which includes information like name, injury, and even hip measurements. Our database also stores test results of the patients which is crucial for the therapists to compare and validate their own visual test they conducted on the patient.

3. Development Getting Started

3.1. Kinect

To use the Kinect for development, a developer must obtain a Kinect for Windows PC. A Kinect for Xbox 360 will not work. Once the Kinect is plugged into a USB port on a computer, the developer will need to install the Kinect driver, available from Microsoft. In addition, the developer should visit http://www.microsoft.com/en-us/kinectforwindows/develop/developer-downloads.aspx to download the Kinect SDK and the Kinect Developer Toolkit. Therapy Kinection was written using v1.6.0 of the SDK and Developer Toolkit.

3.2. Therapy Kinection Desktop Program

In addition to the Kinect requirements listed below, development of Therapy Kinection requires Visual Studio 2010 or later, as well as .Net Framework 4.0. To create an application, you must create a new C# WPF project. Name your project and you may begin.

When beginning development, you will have to specify that your program requires the Kinect DLL files. To do this, follow the instructions below:

1. With your Visual Studio 2010 solution open, right click on your solution, and click “Add Reference”
2. In the window that appears, select the tab titled “.NET”.
3. Ensure that there is a listing titled “Microsoft.Kinect”.
4. If there is no reference for Microsoft Kinect, select the tab titled “Browse”, and navigate to “C:\Program Files\MicrosoftSDKs\Kinect\V1.6\Assemblies\Microsoft.Kinect.dll”. Finally, select OK, and the reference will be added to the project.
5. Now, in any class that you plan to access the Kinect from, you must type "using Microsoft.Kinect" in the import section of the code. This allows you to use the Kinect senor, as well as various objects created by Microsoft to facilitate development.

3.3. Database

To create a Microsoft SQL Server Compact database from scratch, select “Add New Item”, or press “Ctrl + Shift + A”. Select “Data” from the C# menu on the left. Select “Local Database” from the list in the center. Name the database at the bottom. Click “Add”. Select “Dataset”, and press “Next >”. Name the dataset, and press finish.

After the dataset is added, you can add tables by going to the Server Explorer, expanding the database, right clicking the Tables folder, and clicking “Add Table”.

To edit TableAdapters, double click the database.xsd file from the Solution Explorer. Drag and drop tables from the Server Explorer. Now you can right click on the TableAdapters to add queries.

4. Database Development

4.1. Introduction

For this database, we wanted to create something as simple, lightweight, and easy to use as possible. To accomplish this, we used the Microsoft SQL Server Compact Edition. To develop with this project, it is essential to have Microsoft SQL Server Compact 3.5 SP2.

4.2. Tables

We have seven tables in our database: Patient, Injury, Test, Component1, Component2, Component3, and Component4. We wanted to make as normalized a database as possible for this project, so we created tables for all data that would be grouped and changed together.

4.2.1 PATIENT TABLE

The patient table holds information that is unique to each individual patient, such as name, Texas Health Resources ID numbers, and joint offsets.

4.2.2 INJURY TABLE

The Injury Table was added so that the same patient could come back to the therapists multiple times for different injuries, without the need to add a new patient tuple. This was an issue with some of the current software that the therapists used, so we wanted to ensure that our database would be more versatile.

4.2.3 TEST TABLE

The Test Table is used to hold the test dates for the different tests patients take. It is used mostly as an index for the Component tables.

4.2.4 COMPONENT TABLE

The Component Tables hold the data for each component. These could be combined into one large table, but we felt that this would become obnoxious and creating queries for such a large table would be cumbersome.

4.3. SQL Queries

Our queries were pretty straight forward. When we needed to add or retrieve data to or from the database, we created a query. The queries were created using the SQL Table Adapters in the .NET framework. We have queries for inserting data, searching data, and retrieving data. Here are some examples:

SearchDataByLName

```
SELECT      pat.Lname, pat.Fname, pat.CreateDate, pat.THR_ID, pat.Minit, inj.PID, inj.Physician, inj.Diag,
inj.StudyNum, inj.SubjectID, inj.InjuryNum, inj.DateSurg, pat.ICGT, pat.ICASIS, inj.Side
FROM         Patient AS pat INNER JOIN
                  Injury AS inj ON pat.PID = inj.PID
WHERE      (pat.Lname = @param1)
```

GetComboInfo

```
SELECT      InjuryNum, Diag, Side, PID
FROM         Injury
WHERE       (PID = @param1)
```

4.4. Desktop Program Integration

The integration of the database into the desktop program is quite simple thanks to the Microsoft SQL Server Compact Edition. The Server Compact Edition is made so that programs can be easily deployed with a database that requires no outside setup. Everything that is needed is installed with the program installer.

To send and retrieve data from the database, we used TableAdapters. Using these, we could easily create methods that executed SQL statements, and from here we populated DataTables, which were easily parsed and manipulated to get the data. Inserts were also easy, as the data to be inserted was simply passed to the correct methods of the TableAdapters, which then sent it to the database.

One issue we encountered with the database however was due to an improper connection string. The issue was, when we were running the program, all the data would save and be searched correctly to the database throughout opening and closing the program, and even opening and closing Visual Studio. However, when we did anything to the database in Visual Studio, all the information stored while running the program was lost. The problem was that the connection string was set to "|DataDirectory|\TherapyKinectionDatabase.sdf". The "|DataDirectory|" part is a shortcut, set by default to the directory of the executable. While working in Visual Studio, this is the same project folder that all of the different .cs and .xaml files are located. However, once the program is run, it creates a copy of this database, and puts it into the /bin/Debug folder, since that is where the executable is running from. The changes made to this copy never get transferred to the original database, and whenever the original is changed or reconnected to, the copy is replaced.

A remedy to this is to change the DataDirectory variable to point to the folder two folders up in the system. However, when the program is actually deployed for use, the DataDirectory should be set back to its default status. This is because, at this point, the executable and database are in the same folder. This fix should only be used during development and testing, NOT during actual use.

5. Desktop Program Development

5.1. Introduction

Therapy Kinection was developed to allow therapists to use the Microsoft Kinect to analyze the Vail Sport Test. The program proves that the Kinect can be used in the physical therapy field. While our program only analyzes the Vail Sport Test, the Kinect could be used to evaluate many other sports therapy rehabilitation tests. The program we have created can be used as tool by other developers for the purpose of using the Kinect to analyze other exercises.

5.2. Window Classes

In WPF Applications, each window in the interface has its own class, consisting of an XAML file and CS file.

5.2.1 MAIN WINDOW

This window is used as the main class of the program. This class has all of the Kinect initialization methods needed for the program to communicate with the Kinect. It also contains the methods that take the joint data provided by the Kinect and draws the skeleton on the MainWindow window. This class handles the drawing of the boundary lines for components 2, 3, and 4 and controls the timer that is used to time each component. The main class also handles all the button events that are present on the MainWindow window, and initializes the Patient object class, setting up our created objects to point to the joints the Kinect is analyzing. Finally this class contains the methods that handle the execution of each component.

5.2.2 NEW USER WINDOW

This window handles the creating of a new patient for testing. The therapist can either add the patient to the database to use for testing later, or the therapist can save the patient information to the database and begin testing using the new patient that is being added. This class has the event methods required to write the new patient information into the database when save is clicked, as well as the event methods to save the patient information and set up the program for testing with the patient being added.

5.2.3 EXISTING USER WINDOW

This window allows the therapist to search for an existing patient in the database. From this window the therapist can select a patient for testing, view the patient information, update the patient information, or remove the patient from the database. By default, the therapist can only search for active injuries. However, there is the option to search for active and inactive injuries.

5.2.4 TEST SELECTION WINDOW

This window allows the therapist to tell the program whether they would like to have the patient practice a component, or do a graded test that will be stored in the database. Practiced tests are not stored in the database upon completion. Once the therapist has told the program whether or not the patient is going to be practicing the test he can press the select button which will allow the therapist to initialize the Kinect and prepare it for the first component of the Vail Sports Test.

5.2.5 VIEW INFO WINDOW

This window allows the therapist to view the patient information for the patient selected from the search results in the Existing User Window. Here the therapist can select which injury is to be viewed. Also this window allows the therapist to choose which test to view in the TreeView pane. The TreeView pane serves as an outline for displaying the selected test, with expandable and collapsible sections for each of the components as well as each standard of each component in the Vail Sport Test. Also, from this window, the therapist can export the selected test to a text file that can be printed. In addition, this window allows the therapist to select this patient for testing.

5.2.6 UPDATE INFO WINDOW

This window allows the therapist to update the patient information for the patient that was selected from the search results in the Existing User Window. The therapist can change the patient information as well as adding an injury for a patient, if they have already received treatment previously for a different injury.

5.2.7 ADD INJURY WINDOW

This window allows the therapist to add a new injury for a patient that has previously received treatment for a different injury.

5.3. GUI Development

For the development of our GUI we decided to make use of multiple windows for the therapist to interact with, rather than combining all of the functions of the program into one big window. Each window has a specific purpose in the program; each window sends data back and forth between windows as needed. The widgets that were selected for the windows were chosen by which widgets fit our needs the best.

5.3.1 MAIN WINDOW

This window is our main window of the program that controls the program and allows the therapist to open the other windows. On this screen we make use of the expand window widget, which was modified to expand the width of our window to allow room for our live feedback console window that has text output showing how the patient is doing during the test. Also at the bottom of the main window there is an area that has all of the textboxes that are used to show the therapists live data about the patient’s knee angle, valgus, etc. These textboxes are covered and hidden from view until the therapist checks the checkbox that uncovers them. In the center of the screen there is an image area that is devoted to displaying the Kinect skeleton once initialization has occurred and a patient has been detected in front of the Kinect.

5.3.2 NEW USER WINDOW

This window is opened when the add patient button is pressed in main window. Here we went with text fields and radio buttons, to get the proper information from the therapist to put in the database.

5.3.3 EXISTING USER WINDOW

This window is opened from the Main Window by pressing "Search Patient" when a patient has already been added to the database and needs to be found to use for testing. We chose to use the table element to display the data because it allowed us to select and highlight entire rows of data in the results.

5.3.4 TEST SELECTION WINDOW

This window is only available after a new patient has been added to the database and was selected for testing, or if an existing patient was found and selected for testing. This window allows the therapist to choose whether the program will run in practice mode or full test mode. The practice mode selection radio buttons are activated until the therapist has selected the Yes radio button, indicating that the therapist wants to practice a component. Once the Yes has been selected, the therapist must select the radio button for the desired test they wish to practice before the select button will be enabled to begin the practice session. Also the select button will be enabled if the therapist selects the no radio button, indicating that a full graded test will be run. Here we chose to use radio buttons so that the therapist can only run one component at a time in practice mode. After the component practice is completed the therapist can repeat this process, selecting other components.

5.3.5 VIEW INFORMATION WINDOW

This window is what allows the therapist to look at past tests of an existing patient that was found in the search patient window. In this window, the therapist can choose which injury they would like to view for a patient from a combobox at the top of the window. The selected injury information will be displayed in disabled textboxes. Also for that patient's injury the therapist can view all of the tests run for that patient. In order to view a specific test the therapist must choose which one to view from the comboxbox above the tree view, then the test will appear below in the tree view widget. We chose to use the tree view widget because it displays the data similarly to the grading sheet currently used by the therapists.

5.3.6 UPDATE INFORMATION WINDOW

This window enables the therapist to update the information of an existing patient. Here the therapist can correct any mistakes they may have made to the patient information when they created the patient. Also this window allows the therapist to add a new injury for an existing patient, as well as storing whether the therapy is complete and the patient is now inactive for a given injury.

5.3.7 ADD INJURY WINDOW

This window enables the therapist to add a new injury for an existing patient. This allows the therapist to treat a return patient, without adding a whole new patient to the database. Since all the patient information is editable in the Update Information Window, the Add Injury Window only asks for information that will be entered into the Injury Table.

5.4. Kinect Integration

For this project we used the Microsoft Kinect because it allows us to accurately track the joints of a patient. Even though the data capture speed of 30FPS is too slow for quick motions like a golf swing, the Kinect is perfect for tracking slow moving motions like the rehabilitation exercise our program tracks.

5.4.1 DISCOVERING THE KINECT

In our program each time the initialization phase is run, we make sure there is a Kinect connected to the computer using the discover Kinect method. Once a Kinect has been found, we continue to the initialize Kinect code. The discover Kinect code is shown below.

```
private void DiscoverKinectSensor()
{
    foreach (KinectSensor sensor in KinectSensor.KinectSensors)
    {
        if (sensor.Status == KinectStatus.Connected)
        {
            // Found one, set our sensor to this
            kinect1 = sensor;
            break;
        }
    }

    if (this.kinect1 == null)
    {
        connectedStatus = "Found none Kinect Sensors connected to USB";
        Debug.WriteLine(connectedStatus);
        return;
    }

    // You can use the kinectSensor.Status to check for status
    // and give the user some kind of feedback
    switch (kinect1.Status)
    {
        case KinectStatus.Connected:
            {
                connectedStatus = "Status: Connected";
                Debug.WriteLine(connectedStatus);
                break;
            }
        case KinectStatus.Disconnected:
            {
                connectedStatus = "Status: Disconnected";
                Debug.WriteLine(connectedStatus);
                break;
            }
        case KinectStatus.NotPowered:
            {
                connectedStatus = "Status: Connect the power";
                Debug.WriteLine(connectedStatus);
                break;
            }
        default:
            {
                connectedStatus = "Status: Error";
                Debug.WriteLine(connectedStatus);
                break;
            }
    }

    // Init the found and connected device
    if (kinect1.Status == KinectStatus.Connected)
    {
        InitializeKinect();
    }
}
```

5.4.2 INITIALIZING THE KINECT

Once a Kinect has been discovered, the Kinect is ready to be initialized. Here we use smoothing to smooth out all of the jitters that the Kinect picks up to allow for more consistent and accurate data readings. Also we initialize the skeleton stream so that we can use it to read the joint data and then we start the Kinect. Every time the Kinect is initialized with this method a new skeleton frame ready handler is created. If too many of these are created the program will hang. In order to prevent this from happening, and the end of testing each handler needs to be removed.

```
private void InitializeKinect()
{
    hipY = new float[3];                        //temp array that holds 3 values for the hip's y location
    rfY = new float[3];                         //temp array that holds 3 values for the rightfoots's y location
    lfY = new float[3];                         //temp array that holds 3 values for the leftfoots's y location
    kinect1 = KinectSensor.KinectSensors[0];    //kinect1 is the first kinect.
    dg = new DrawingGroup();                    //Create a drawing group
    imgsrc = new DrawingImage(dg);              //Link the imageSource to the drawing group
    Image.Source = imgsrc;

    TransformSmoothParameters smoothingParam = new TransformSmoothParameters(); //smoothing parameters put in to filter
    {
        smoothingParam.Smoothing = 0.5f;
        smoothingParam.Correction = 0.55f;
        smoothingParam.Prediction = 0.5f;
        smoothingParam.JitterRadius = 0.07f;
        smoothingParam.MaxDeviationRadius = 0.04f;
    };

    if (kinect1 != null)//If there is such a Kinect
    {
        kinect1.SkeletonStream.Enable(smoothingParam);          //enables skeleton stream with deifne smoothing above
        kinect1.SkeletonFrameReady += kinect1_SkeletonFrameReady;   //Create a handler for when the Skeleton is ready
        video = new Queue<ColorImageFrame>();
        kinect1.Start();                                        //turn on the Kinect
        progressBar1.Maximum = 150;
    }

    startseg.IsEnabled = true;
}
```

5.4.3 SKELETON FRAME READY

The method is ideally called 30 times for every second the Kinect is running. This method is used when there is a skeleton present and being tracked. From here we assign all of the joints to their proper object classes, as well as perform the calibrate phase to account for each individual's flexibility differences. Also this method draws the boundaries for the third and fourth components and performs the logic that is used for drawing the skeleton on screen. We draw the boundaries before we draw the skeleton so that you can see when the patient's foot is on or has crossed the boundary. This method is basically what drives the program, from here it goes to the proper component for testing.

```
/*method that performs the operations on the skeleton when it is detected*/
void kinect1_SkeletonFrameReady(object sender, SkeletonFrameReadyEventArgs e)//Event Handler for when the Skeleton is ready
{
    dc = dg.Open();
    //creates a drawing context for skeleton grid to be drawn on screen

    Skeleton skeleton = FindSkeleton(e);

    //if there is a skeleton, do analysis on the skeleton
    if (skeleton != null)//skeletons.Length != 0
    {
        Skeleton mainSkel = skels[0];   //stores the first skeleton in the skels array in the mainskel variable

        foreach (Skeleton s in skels)
        {
            if (s.TrackingState == SkeletonTrackingState.Tracked)
            {
                mainSkel = s;
                break;
            }
        }



        JointCollection joints = mainSkel.Joints;//Create a joint collection of all the joints in mainSkel

        Joint rh = joints[JointType.HipRight], rk = joints[JointType.KneeRight], ra = joints[JointType.AnkleRight], rf = joints[JointType.FootRight],
            rs = joints[JointType.ShoulderRight], re = joints[JointType.ElbowRight], rw = joints[JointType.WristRight], rhd = joints[JointType.HandRight],
            ch = joints[JointType.HipCenter], sp = joints[JointType.Spine], cs = joints[JointType.ShoulderCenter], head = joints[JointType.Head],
            lf = joints[JointType.FootLeft], la = joints[JointType.AnkleLeft], lk = joints[JointType.KneeLeft], lh = joints[JointType.HipLeft],
            ls = joints[JointType.ShoulderLeft], le = joints[JointType.ElbowLeft], lw = joints[JointType.WristLeft], lhd = joints[JointType.HandLeft]; //assigns the joints

        Patient.Right.Knee.Joint = rk;
        Patient.Right.Ankle.Joint = ra;
        Patient.Right.Foot.Joint = rf;
        Patient.Right.Hip.Joint = rh;
        Patient.Right.centerHip.Joint = ch;
        Patient.Right.centerShoulder.Joint = cs;

        Patient.Left.Knee.Joint = lk;
        Patient.Left.Ankle.Joint = la;
        Patient.Left.Foot.Joint = lf;
        Patient.Left.Hip.Joint = lh;
        Patient.Left.centerHip.Joint = ch;
        Patient.Left.centerShoulder.Joint = cs;


        int i = 0;
        if (!calibrateDone) //if  statements to perform the calibration steps
        {
            progressBar1.Value++;
            if (calibrateCounter < 150)//5 seconds, starts timer to get measurement for leg length
            {
                //rightLegLength += getLegLength(rh, rk, ra);   //gets the right leg legth measurement
                rightLegLength += Patient.Right.getLegLength();

                //leftLegLength += getLegLength(lh, lk, la);   //gets the right leg legth measurement
                leftLegLength += Patient.Left.getLegLength();

                //Sidetrunkangle += getTrunkSideAngle(cs, ch, rh);
                Sidetrunkangle += Patient.Right.getSideTrunkAngle();

                //FBtrunkangle += getTrunkFBAngleR(cs, ch, rk);
                FBtrunkangle += Patient.Right.getFrontBackTrunkAngle();

                //kneeLockRatio += (rk.Position.Z - rh.Position.Z);
                rightKneeLockRatio += (Patient.Right.Knee.Z - Patient.Right.Hip.Z);
                leftKneeLockRatio += (Patient.Left.Knee.Z - Patient.Left.Hip.Z);

                //lfGround += lf.Position.Y;
                lfGround += Patient.Left.Foot.Y;

                //rfGround += rf.Position.Y;
                rfGround += Patient.Right.Foot.Y;

                //debugBox.Text = "Stand with leg straight and knees locked";
            }
            else if (calibrateCounter == 150)
```

```
        {
            rightKneeLockRatio /= 150;
            Patient.Right.kneeLockRatio = rightKneeLockRatio;
            leftKneeLockRatio /= 150;
            Patient.Left.kneeLockRatio = leftKneeLockRatio;
            rightLegLength /= 150;//final leg length
            Patient.Right.legLength = rightLegLength;
            leftLegLength /= 150;//final leg length
            Patient.Left.legLength = leftLegLength;
            FBtrunkangle /= 150;
            Patient.Right.fbBaseTrunkAngle = FBtrunkangle;
            Patient.Left.fbBaseTrunkAngle = FBtrunkangle;
            FBbaselinetrunkangle = FBtrunkangle;
            Sidetrunkangle /= 150;
            Patient.Right.sideBaseTrunkAngle = Sidetrunkangle;
            Patient.Left.sideBaseTrunkAngle = Sidetrunkangle;
            Sidebaselinetrunkangle = Sidetrunkangle;
            lfGround /= 150;
            Patient.Left.footBaseGroundValue = lfGround;
            rfGround /= 150;
            Patient.Right.footBaseGroundValue = rfGround;
            calibrateDone = true;
            startseg.IsEnabled = true;
            between = true;
        }

        calibrateCounter++;
    }

    Rect tempRect = new Rect(0, 0, 640, 480);//Draw a rectangle so you have a background to draw onto.
    dc.DrawRectangle(Brushes.Black, null, tempRect);

    if (Current.SelectedTest.docomp3 == true && calibrateDone == true && comp4done == false && (comp2done == true || Current.SelectedTest.isPractice == true))
    {
        if (secs == 120)
        {
            leftZBoundary = la;
            rightZBoundary = ra;
        }

        DepthImagePoint depthPoint = kinect1.CoordinateMapper.MapSkeletonPointToDepthPoint(rightZBoundary.Position, DepthImageFormat.Resolution640x480Fps30);

        SkeletonPoint newLateralBoundary = new SkeletonPoint();
        newLateralBoundary.X = rightZBoundary.Position.X + (float)rightLegLength;
        newLateralBoundary.Y = rightZBoundary.Position.Y;
        newLateralBoundary.Z = rightZBoundary.Position.Z;

        //drawing left boundary
        Point leftBottomOfBound = new Point(depthPoint.X - 100, depthPoint.Y);
        dc.DrawEllipse(boundaryBrush, null, leftBottomOfBound, 7, 7);
        Point leftTopOfBound = new Point(depthPoint.X + 50, depthPoint.Y);
        dc.DrawEllipse(boundaryBrush, null, leftTopOfBound, 7, 7);

        dc.DrawLine(boundarypen, leftBottomOfBound, leftTopOfBound);
    }
    else if (Current.SelectedTest.docomp4 == true && calibrateDone == true && (comp3done == true || Current.SelectedTest.isPractice == true) && comp4done == false)
    {
        if (secs == 120)
        {
            leftZBoundary = lf;
            rightZBoundary = rf;
        }

        SkeletonPoint newLateralBoundary = new SkeletonPoint();
        newLateralBoundary.X = rightZBoundary.Position.X;
        newLateralBoundary.Y = rightZBoundary.Position.Y;
        newLateralBoundary.Z = rightZBoundary.Position.Z - (float)0.0127;

        DepthImagePoint depthPoint = kinect1.CoordinateMapper.MapSkeletonPointToDepthPoint(newLateralBoundary, DepthImageFormat.Resolution640x480Fps30);

        //drawing left boundary
        Point leftBottomOfBound = new Point(depthPoint.X - 100, depthPoint.Y);
        dc.DrawEllipse(boundaryBrush, null, leftBottomOfBound, 7, 7);
        Point leftTopOfBound = new Point(depthPoint.X + 50, depthPoint.Y);
        dc.DrawEllipse(boundaryBrush, null, leftTopOfBound, 7, 7);

        dc.DrawLine(boundarypen, leftBottomOfBound, leftTopOfBound);
    }
```

```
        if (mainSkel.TrackingState == SkeletonTrackingState.Tracked)  //if current skeleton is being tracked, do analysis
        {
            foreach (Joint j in joints)
            {
                Brush drawBrush = null;

                if (j.TrackingState == JointTrackingState.Tracked)
                {
                    drawBrush = trackedbrush;
                }
                else if (j.TrackingState == JointTrackingState.Inferred)
                {
                    drawBrush = inferredbrush;
                }

                if (drawBrush != null)
                {
                    dc.DrawEllipse(drawBrush, null, SkeletonPointToScreen(j.Position), 7, 7);//Radius of size 10 should make the joints easier to see
                }
            }

            //Include the DrawingContext in all the function calls since it is no longer global.
            doDraw(dc, head, cs);

            doDraw(dc, cs, ls);
            doDraw(dc, ls, le);
            doDraw(dc, le, lw);
            doDraw(dc, lw, lhd);

            doDraw(dc, cs, rs);
            doDraw(dc, rs, re);
            doDraw(dc, re, rw);
            doDraw(dc, rw, rhd);

            doDraw(dc, cs, sp);
            doDraw(dc, sp, ch);

            doDraw(dc, ch, lh);
            doDraw(dc, lh, lk);
            doDraw(dc, lk, la);
            doDraw(dc, la, lf);

            doDraw(dc, ch, rh);
            doDraw(dc, rh, rk);
            doDraw(dc, rk, ra);
            doDraw(dc, ra, rf);

            dg.ClipGeometry = new RectangleGeometry(new Rect(0, 0, 640, 480));//Smooths the joints out when a joint has left the Kinect's view.

            if ((calibrateDone == true) && (between == false))//if calibrate phase is done, go into testing components
            {
                if ((testing == false) && (between == true))
                {
                    startseg.IsEnabled = true;
                }
                //if any of these are set to true, then this test is in practice mode
                if (Current.SelectedTest.isPractice == true)
                {
                    if (Current.SelectedTest.docomp1 == true)//if comp1 is true will do component 1
                        Component1();
                    else if (Current.SelectedTest.docomp2 == true)//if comp2 is true will do component 2
                        Component2();
                    else if (Current.SelectedTest.docomp3 == true)//if comp3 is true will do component 3
                        Component3();
                    else if (Current.SelectedTest.docomp4 == true)//if comp4 is true will do component 4
                        Component4();
                }
                else  //if none of those are true, the it is not in practice mode and will recording the scoring
                {
                    if (comp1done == false)//if compontent is not done then it should be done first, becuase of ordering
                        Component1();
                    else if (comp1done == true && comp2done == false)//if component 1 is done and component 2 is not done, then do component 2
                        Component2();
                    else if (comp2done == true && comp3done == false)//if component 2 is done and component 3 is not done, then do component 3
                        Component3();
                    else if (comp3done == true && comp4done == false)//if component 3 is done and component 4 is not done, then do component 4
                        Component4();

                }
            }
        }
    }
    dc.Close();
}
```

5.4.4 FIND SKELETON

The method is called when there is no skeleton being tracked or the last skeleton that was being tracked has been lost. This method will continue looking for a skeleton until one is found. The skeleton closest to the Kinect is the one that will be selected for tracking. This enables people to walk behind the Kinect, or the patient, and not disrupt the test.

```
Skeleton FindSkeleton(SkeletonFrameReadyEventArgs e)
{
    Skeleton skeleton = null;

    using (SkeletonFrame SFrame = e.OpenSkeletonFrame())
    {
        if (SFrame == null)
        {
            return null;    //when there is no skeleton seen, no skeleton is created
        }

        if (skels == null)
        {
            skels = new Skeleton[SFrame.SkeletonArrayLength];   //creates a new skeleton when there isnt one
        }

        SFrame.CopySkeletonDataTo(skels);   //puts the skeleton found in the skeleton frame into the global skeleton used for testing

        if (CurrentTrackingId != 0)  //looking for skeleton to see if same as already tracking
        {
            skeleton =(from s in skels where s.TrackingState == SkeletonTrackingState.Tracked &&
                       (s.Joints[JointType.Head].TrackingState == JointTrackingState.Tracked || s.Joints[JointType.ShoulderCenter].TrackingState == JointTrackingState.Tracked)
                       && s.TrackingId == CurrentTrackingId
                 select s).FirstOrDefault();

            if (skeleton == null) //if skeleton no foudn then its gone, forget that skeleton and look for another
            {
                CurrentTrackingId = 0;
                kinect1.SkeletonStream.AppChoosesSkeletons = false;
            }

        }
        else
        {
            // Try to find someone new
            skeleton = (from s in skels where s.TrackingState == SkeletonTrackingState.Tracked &&
                       s.Joints[JointType.Head].TrackingState == JointTrackingState.Tracked select s).FirstOrDefault();

            if (skeleton != null)
            {
                // Found one; start tracking them
                CurrentTrackingId = skeleton.TrackingId;
                kinect1.SkeletonStream.AppChoosesSkeletons = true;
                kinect1.SkeletonStream.ChooseSkeletons(CurrentTrackingId);
            }
        }
    }

    return skeleton;
}
```

5.5 Patient Object Classes

To make things easier for accessing all of the joints for a given side and make the standard calculations easier, we have made these methods a part of our side class.

5.5.1 DIAGRAMS

Figure 5.1 shows the relationship between the body, side, and joint classes. Figure 5.2 shows the inheritance between the joint classes and the bodyjoint class.

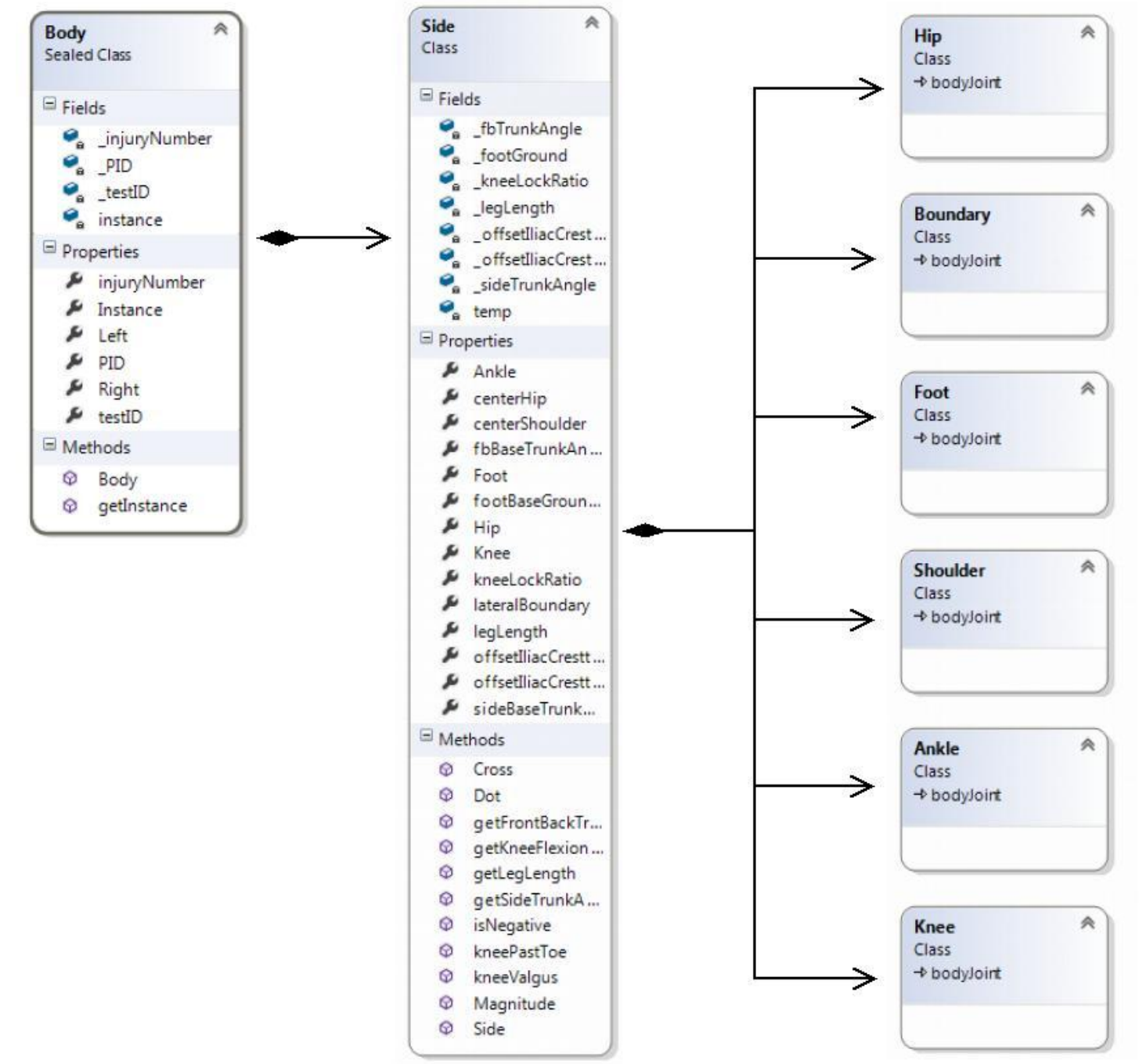

Figure 5.1 Body, Side, Joint Classes

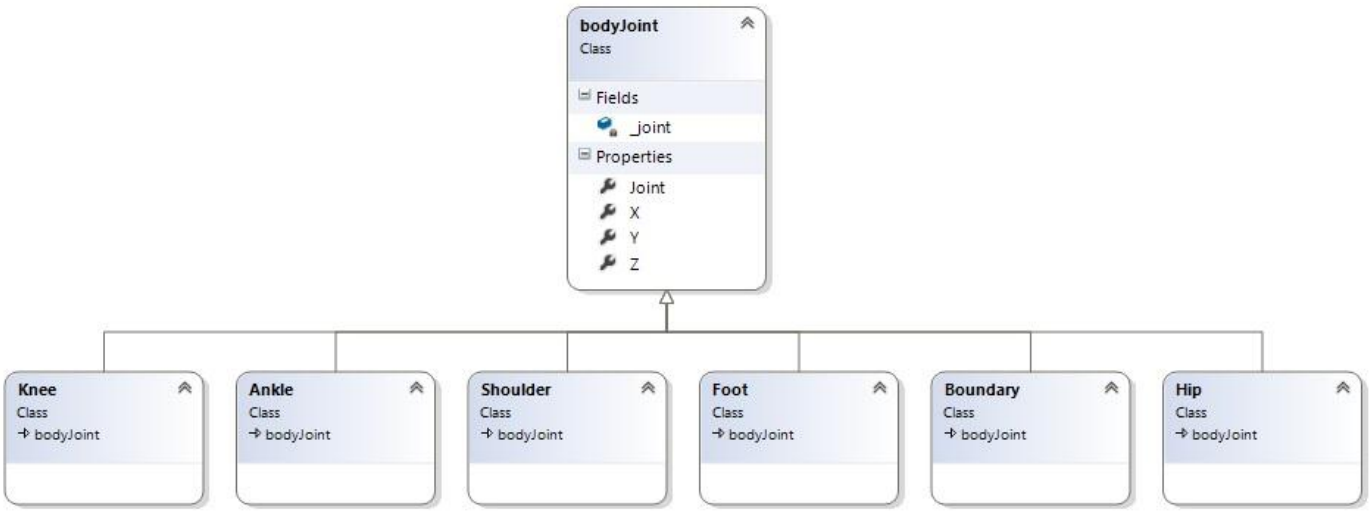

Figure 5.2 Joint Class Inheritance

5.5.2 CLASSES

These classes were designed in a way that the compiler could run the program in an optimum way and allowed us to consolidate sections of code. Also, this allowed us to be able to access these objects from the other classes and window classes without having to pass as many values.

5.5.2.1 Body

The body class is an object class that creates 2 side classes, a left and a right. The body class and side class are related by a "Has-A" composition relationship (shown in Figure 5.1). Each instance of a body that is created is composed 2 side objects which are created in the body class constructor.

5.5.2.2 Side

The side class is an object class that creates all of the joint classes. These Joint object classes are related to the side by a "Has-A" composition relationship (shown in Figure 5.1). These joint classes are created inside of the constructor of the side class. This class also contains the calculations for the various standards of each component. Because these calculations are methods in the side class, when they are called the side they calculate is based on which side is requested in the method call for the side class. For example, if you want to get the knee flexion angle for the patient's right leg you would use this line of code:

```
Patient.Right.getKneeFlexionAngle();
```

In this example Patient is an instance of the body class, Right is the desired side for the instance of the body class, and the last part is the desired method that is a part of the side class.

5.5.2.3 Joint Classes

These classes are object classes that are created with each instance of a side class. The side and joint classes are related by a "Has-A" composition relationship (as shown in Figure 5.1). Since each joint class has the same attributes we can subclass the joint classes to the bodyjoint class. The joint classes inherit the attributes of the bodyjoint class (as shown in figure 5.2). Each joint has a property for the

x,y,z coordinate values as well as a property that points to the joint object the Kinect provides. In order to access the properties of each joint you would write the method call like this:

```
Patient.Right.Knee.Z;
```

This call would access the Z-value of the right knee. In order to set this joint object to point to the joint you would need to assign the Kinect's joint object to our joint object class by doing this:

```
Patient.Right.Knee.Joint = joints[JointType.KneeRight];
```

Once you set our joint object class to point the Kinect joint, the x,y,z coordinate properties for our joints are automatically now pointing to the x,y,z coordinate values of the joint object the Kinect created.

6. Glossary of Terms

Anterior Cruciate Ligament (ACL) – One of the 4 major ligaments of the human knee.

C# - Programming language developed by Microsoft. The language is used for Kinect and general Windows Application programming.

Database (DB) - Structured collection of data that contains the patient and test information.

Dynamic knee valgus - The bending angle of the knee inward, past the big toe, towards the opposite leg during the exercise.

Knee flexion - The angle the knee makes while bending the leg.

Microsoft Kinect - Kinect is a motion sensing input device by Microsoft for the Windows PCs. Based around a webcam-style add-on peripheral, it enables users to control and interact with PC through a natural user interface using gestures and spoken commands. The Kinect also recognizes 20 joints on the human body at a capture rate of 30 Hz.

Microsoft Visual Studio 2010 - An IDE that is used for many different types of programming languages, primarily languages developed by Microsoft.

Patella extending past big toe - During the leg squat the patient cannot have their knee pass in front of their big toe during the squat.

Upright trunk – Defined by Craig Garrison as the trunk of the patient being straight up and less than 30 degrees off center in any direction.

Vail sport test - A physical therapy test, co-developed by Craig Garrison, that is used to determine if the patient has regained the proper strength in his or her ACL to return to normal physical activity. A research paper concerning the test can be found here:
http://www.texashealth.org/workfiles/THR%20System/Ben_Hogan/Garrison%202012.pdf

7. Appendix

7.1. Appendix A: Vail Sport Test

VAIL SPORT TEST™

Name: _____ Date: _____

MD: _____ DX: _____ Mo. S/P: _____

Total Points: _____/54 * Patient must score 46/54 on the test in order to pass

Single Leg Squat (goal: 3 minutes)

1. Knee flexion angle between 30 and 60°

Yes (1) No (0)

2. Patient performs repetitions without dynamic knee valgus

*knee valgus = patella falls medial to the great toe

Yes (1) No (0)

3. Patient avoids locking knee during extension

Yes (1) No (0)

4. Patient avoids patella extending past the toe during knee flexion

Yes (1) No (0)

5. Patient maintains upright trunk during knee flexion

Yes (1) No (0)

Minute 1 _____ Minute 2 _____ Minute 3 _____

Single Leg Squat Total Points: _____/15

• If patient repeats error on 3 consecutive repetitions after correction, they are not eligible to receive a point for that particular standard (within each 1 minute timeframe).

Lateral Bounding (goal: 90 seconds)

1. Knee flexion angle is 30° or greater during landing

Yes (1) No (0)

2. Patient performs repetitions without dynamic knee valgus

*knee valgus = patella falls medial to the great toe

Yes (1) No (0)

3. Patient performs repetitions within landing boundaries

Yes (1) No (0)

4. Landing phase does not exceed 1 second in duration

Yes (1) No (0)

5. Patient maintains upright trunk during knee flexion

Yes (1) No (0)

1st 30 sec _____ 2nd 30 sec _____ 3rd 30 sec _____

Lateral Bounding Total Points _____/15

• If patient repeats error on 3 consecutive repetitions after correction, they are not eligible to receive a point for that particular standard (within each 30 second timeframe).

Forward Jogging (goal: 2 minutes)

1. Knee flexion angle between 30 and 60°

Yes (1) No (0)

2. Patient performs repetitions within landing boundaries

Yes (1) No (0)

3. Patient performs repetitions without dynamic knee valgus

* knee valgus = patella falls medial to the great toe

Yes (1) No (0)

4. Patient avoids locking knee during extension

Yes (1) No (0)

5. Landing phase does not exceed 1 second in duration

Yes (1) No (0)

6. Patient maintains upright trunk during knee flexion

Yes (1) No (0)

Minute 1 _____ Minute 2 _____

Forward Jogging Total Points _____/12

• If patient repeats error on 3 consecutive repetitions after correction, they are not eligible to receive a point for that particular standard (within each 1 minute timeframe).

Backward Jogging (goal: 2 minutes)

1. Knee flexion angle between 30 and 60°

Yes (1) No (0)

2. Patient performs repetitions within landing boundaries

Yes (1) No (0)

The International Journal of Sports Physical Therapy | Volume 7, Number 1 | February 2012 | Page 30

3. Patient performs repetitions without dynamic knee valgus

* knee valgus = patella falls medial to great toe

Yes (1) No (0)

4. Patient avoids locking knee during extension

Yes (1) No (0)

5. Landing phase does not exceed 1 second in duration

Yes (1) No (0)

6. Patient maintains upright trunk during knee flexion

Yes (1) No (0)

Minute 1 _____ Minute 2 _____

Backward Jogging Total Points _____/12

• If patient repeats error on 3 consecutive repetitions after correction, they are not eligible to receive a point for that particular standard (within each 1 minute timeframe).