Going “Hands-Free” with a Kinect

Sreya Basuroy
sreya.basuroy@gmail.com

Christopher Headley
c-headley@hotmail.com

Samuel Udotong
sudotong@gmail.com

Joseph Wang
gnawhpesoj@gmail.com

NJ Governor’s School of Engineering & Technology 2011

1 Abstract

When Microsoft released its Kinect accessory for the Xbox 360 gaming console in late 2010, not many people understood the true impact of this product. The Kinect is a motion-sensing camera capable of constructing true 3D images, complete with depth information. It is the first widely affordable camera of this kind to become available to the public; thus, research into the capabilities of such a device is now more feasible than it was before. The primary goal of our project is to investigate the Kinect’s capabilities of improving athletic and medical services. The design is such that the user would be able to interface with the computer, instead of the Xbox 360, via gestures captured by the Kinect’s camera and depth sensors. We implemented an application where the user practices a golf swing and receives diagnostic information about the strengths and weaknesses of the swing in comparison to the swing of a professional golfer. This simulation technique allows the skeleton of the user to be accurately illustrated by measuring the shoulder and elbow angles and the alignment of the shoulder, hips, and feet. The program outputs accurate and reliable results consistently. Furthermore, this application can be beneficial for rehabilitating shoulders, elbows, and other joints by pinpointing the exact locations of injuries and performing rehabilitative exercises in one golf swing. The golf swing analysis application using the Kinect is a feasible, cost-effective, and user-friendly program. This design is certainly revolutionary for future athletic and medical applications that utilize joint and body motion recognition.

2 Introduction

For years, the video game industry has been producing new technology to improve the capabilities and usability of games. The advent of the Kinect revolutionized game play since players are now able to control the games with their gestures and voice commands. However, this improvement in technology has not led to many breakthroughs in the potential functions of videogames. Apart from the usual games, the Kinect can be used to simulate various real-life applications. Thus, we decided to generate a golf swing analysis system that would help amateur golfers improve their techniques and emulate professional golfers. Unlike the current golf swing analysis systems by V1 Pro and CSwing that simply provide slow motion video capture, this
application will allow the user to directly compare his or her techniques and movements with those of the expert, along with receiving constructive advice. This direct comparison can showcase any slight difference in angle of swing and/or position. Since golf is a sport that relies on the utmost accuracy and precision, the smallest difference in angle can alter the power and direction of a golf swing. This detailed athletic analysis can be crucial in enhancing one’s skill level in a more effective manner.

Furthermore, our application can also be used for rehabilitation of the shoulder and elbow. Rehabilitation can be a very painful, expensive, and arduous process for the patient. At the Physical Therapy Department of Akron General Hospital in Akron, Ohio, one session of physical therapy costs between $79 and $170 [4]. A patient continuing these regimens for six months would accrue a charge of thousands of dollars. Moreover, physiotherapists often recommend assistive and active shoulder flexion, repetition, and extension exercises to recover complete shoulder functions. However, this slow process of rote repetition for approximately six months can cause the patient to become frustrated and unmotivated. With this Kinect application, patients will be able to perform all of the much-needed rehabilitation exercises just by playing golf. Thus, the patient will be continuing rehabilitation and enjoying a fun sport at the same time.

As of 2006, 7.5 million Americans were plagued with shoulder problems; 4.1 million of those had rotator cuff problems [5]. Since the shoulder is a ball-and-socket joint with the greatest range of motion, it is the most likely to be injured. The joint can also be unstable because the ball of the upper arm is larger than the shoulder socket that holds it. Most shoulder injuries involve the muscles, ligaments, and tendons, rather than the bones. Shoulder problems can be caused by overuse, sports injuries, and aging. Sports injuries, including abnormal twisting of the shoulder, sprains, dislocations, rotator cuff problems, frozen shoulder, fractures, and arthritis, are the single most cause of shoulder problems.

Tennis elbow (radial epicondylitis) and golfer’s elbow (ulnar epicondylitis) are some of the most common sports-related elbow injuries. Incorrect and excessive straining of the hand joint and fingers cause extreme pain in the middle of the upper arm when the hand joint is flexed or the lower arm is turned away from the body. Although the surgery procedures have a success rate of over 90 percent, many patients turn to acupuncture or physical therapy for rehabilitative purposes [1]. Thus, it is crucial to create software that can help restore proper shoulder function post surgery or other medical treatment.

The Kinect application simulating a golf swing can be used for multiple purposes. The game can mirror shoulder and elbow rehabilitation by focusing on increasing joint rotation and movement. The different stages of the golf swing parallel the rehabilitative exercises of assistive and active flexion, repetition, and extension. This cost-effective and practical application can provide accurate and constructive advice immediately. Not only will golfers be able to emulate a professional’s golf swing, but patients with shoulder and elbow injuries will also be able to receive diagnostic information about their injuries and improve their motions.
3 Background

3.1 Kinect

The Kinect (Figure 1) is a motion-sensing controller produced by Microsoft for use with the Xbox 360 video game console. This tool empowers players by letting them control the game with their own motion and voice commands. The Kinect sensor consists of an RGB (an additive color model in which red, green, and blue light is added together to reproduce a broad array of colors) camera, depth sensor, and microphones. The depth sensor includes an infrared laser projector that captures 3D video data; its range is between two and twenty feet in front of it. Its field of view is 57 degrees horizontally and 43 degrees vertically. It is possible to alter the depth sensor’s range to accommodate game play and the player’s surrounding environment. In fact, the closer the player is to the Kinect camera, the brighter the color of his or her body on the grid produced by the Kinect. The sensor outputs video at a frame rate of 30 Hz, while the RGB video stream uses 8-bit VGA (Video Graphics Array) resolution (640 X 480 pixels).

![Figure 1: The Kinect.](image-url)

Using these various motion, depth, and color sensors, the Kinect recognizes a human and tracks the body by mapping 24 joints in less than 30 seconds. In order for the Kinect camera to acquire this information, the player must strike a pose standing up with his or her hands raised up in the air at 90 degree angles. To ensure accurate calibration, this information is then translated onto the digital skeletal grid where shape, size, depth, and real-time actions are reproduced exactly. It has been tested to show that a maximum of six players can be tracked simultaneously with the Kinect; yet, only two players can be actively tracked for motion analysis.

3.2 OpenNI

Open Natural Interaction (OpenNI) is an organization that offers natural interaction devices for use across multiple tools. The natural interaction devices allow for a more user-friendly experience in which the player can communicate his/her actions with his body motions and voice commands alone. For the purpose of this project, OpenNI binary, middleware, and hardware files were downloaded to access the application programming interfaces (APIs) needed for the Kinect to function [3]. Additionally, PrimeSense middleware was also downloaded to access the open source drivers for the Kinect camera and sensors. This tool detects individual users who are within the depth range of the camera and tracks their body positions. This depth data from the Kinect camera is mapped on a grid to point out the exact locations of the user’s joints, performing a process known as “skeletonization”. Therefore, these tools allow us to handle gesture-based applications with the Kinect.

3.3 The Golf Swing

Golf is a sport that requires extreme accuracy and precision in order to hit the ball into the hole with the fewest number of strokes. Golfers begin with their bodies and
clubs facing the golf balls, which are centered between the golfers’ feet, and linear to the target hole. The feet should remain shoulder width apart, while the knees are slightly bent. A large knee flex is not only uncomfortable but also destabilizes the swing at impact. In order to ensure a proper swing, several angles need to be maintained accurately (Figure 2).

Figure 2: The golf swing demo in five stages.

3.4 Golf Swing Angles

The most crucial angles in golf are the spine and lag angles. The spine angle is the angle created by the bending of the back over the ball. However, excessive bending curves one’s spine and makes impact with the ball difficult as the swing moves off-line. Golfers should keep their spines straight and bend a little at the hips; this ensures “stacking” or alignment of the shoulder, knees, and feet. The lag angle is essential in creating swing speed and solid contact. The golfer must maintain proper wrist and elbow angles throughout the entire golf swing to keep the swing in balance and on path. This helps finish the downswing and create ball speed and spin [2].

4 Procedure/Methods

4.1 Mario Kart

The first objective of our group was to use the Kinect to control a platform other than the Xbox 360, for which it was designed. At the beginning of this project, we were not entirely familiar with the functions and capabilities of the Kinect. Thus, we worked on an introductory activity that allowed us to acquaint ourselves with the details of programming for and using the Kinect. We did not want to begin the golf swing program before we had an idea of the Kinect’s capabilities, so we decided to control simple video games to become familiar with the platform. We settled on those of the early video game consoles. We then had to decide between using the Kinect to control Mario Kart on the Nintendo 64 and using the Kinect to play Super Mario Brothers. Mario Kart is a set of go-kart racing video games developed by Ninendo in 1992. In this popular game, characters race their go-karts on various race tracks. Players can acquire power-up items by driving into item boxes or coins; these items can then be used for speeding up the vehicle and for offensive and defensive attacks. Both options were viable, but Mario Kart involved more complex gestures. The controls for Super Mario Brothers were simply pressing a few buttons to move left, right, and jump, which could each be tailored to different gestures with little difficulty. But, Mario Kart had larger issues involving accelerating, reversing, and turning. Before we created algorithms to handle these tasks, we had to find a way to edit the controls for the Nintendo 64 and sync it with the Kinect.

We eventually decided to use a Nintendo 64 emulator for this task, which is a computer program that recreated the Nintendo 64 and enabled us to modify Mario Kart. Also, to simplify the process of obtaining data from the Kinect, we downloaded OpenNI and PrimeSense, technologies created by other Kinect pioneers to serve as a medium between the computer and the Kinect. The programs work together to send the same “passcode”
that the Kinect sends to the Xbox 360 to the computer instead. These tools provide us with the necessary open source drivers to access the Kinect’s camera and sensors and create new applications for this accessory. Knowing little about how the three programs worked together, we experimented, compiled, and debugged over and over until we finally got some realistic results. Thus ensued the process of using gestures to replace the existing keyboard controls.

4.1.1 Challenges

First, we had to define several gestures for different behaviors: a wave, circle, swipe, push, and steady, which is a default that averages the user’s hand positions to prevent excessive shaking. Each gesture was mapped to a keyboard control that would normally be used to run the game. The first challenge was accelerating. Since the “A” button is held down for the entirety of the game on the actual video game console, we decided to keep the cart accelerating at all times as long as the user’s hands were being tracked properly. However, this led to excessive acceleration and the cart spun out as the time counter reached “Go!” at the beginning of each race. All of the other drivers immediately got a head start, and this was detrimental to the efficiency of the program. Therefore, we decided to use a push gesture to begin the permanent accelerator button, a simple fix to the problem. We also looked into a way to reverse for situations where the user got stuck against a wall. We eventually decided to control the reversing action by increase the distance in the x-direction between the user’s two hands.

Figure 3: Beginning of Mario Kart.

We had to figure out a way to allow for turns of different magnitudes - every turn was different and so it would not be as simple as gestures for left and right. It was vital to ensure that the turning gesture was not so sensitive to cause the car to swerve. Assuming that the user was holding his or her hands in space as he or she would a steering wheel, we made a function that put the hands on a xy-coordinate plane and set up a box that took the angle between the two hands. If the angle was equal to zero plus or minus a threshold, the hands were lined up and the cart accelerated straight ahead. Once the angle became positive (i.e. the user’s left hand was below his or her right hand), the cart started turning left (Figure 4). The more positive the angle, the more the cart would turn left. Accordingly, a negative angle made the cart go right.
Another dilemma was how to activate items while driving. If the user made a circle motion with his or her right hand to use an item, he or she might accidentally swerve right or left. We wanted to be able to easily use items while maintaining good control over the car. One consideration was tracking the user’s legs and kicking to use items; however, this would require a different program (UserTracker instead of Hands). We decided to perform a punch gesture to use the item. With correct precision, the user’s hand only moves in the z-direction and the steering is not adversely affected. This application allowed us to familiarize ourselves with the possible uses of the Kinect and OpenNI.

4.2 Golf Swing Analysis

In order to accomplish something the Kinect was not originally designed and marketed for, an alteration in the style of programming was necessary for the golf swing analysis. After having achieved a high degree of control with using the Kinect to play Mario Kart 64, thereby attaining a high degree of familiarity with the Kinect’s capabilities, we began working on the golf swing program. The Kinect’s input to simulate key presses in Mario Kart 64 was gesture-based, focusing on implementing formulas for certain motions that when triggered would correspond to a specified keyboard input. Our next endeavor would require disabling the effects of certain body configurations until all prerequisite conditions were satisfied. In doing so, we created a linear sequence of events so that relative form and technique could be compared to a preset model. The project code was based on a C++ project included with the drivers necessary to use the Kinect. This NIUserTracker program already mapped the joints and drew the limbs needed for this application on the grid; we would only have to tailor it to athletic training and rehabilitation purposes.

4.2.1 Angles

In determining how to realistically compare the motion of any two golfers, who could vary drastically in size and proportion, the approach we decided on was to measure the angles of certain key joints at predetermined positions in relation to the hip joint. Having no automatic way to determine angles and needing them in three dimensions required a creative solution that would consider three joint positions in any one of three planes as a triangle ABC where C is the joint to be measured. In order to accomplish this, the distance would have to be determined with a simple distance formula:

\[ a = \sqrt{((\text{jointb.position.X} - \text{jointc.position.X})^2 + (\text{jointb.position.Y} - \text{jointc.position.Y})^2)}; \]

With these values, we would use a derivation of the formula for the Law of Cosines that solves the equation for the cosine of angle C. To establish the actual measure we would find the inverse cosine of the cosine of C:
\[ x = \cos((a^2 + b^2 - c^2)/(2ab)) \]

return \( x \);

The formulas given all assume angle measures in xy-coordinates. To account for the depth of the golfer, measures would also have to be made from the xz-coordinates and the yz-coordinates. Simultaneously, the distances between the user's shoulders and hands also needed to be computed.

### 4.2.2 Stages

Our main goal was to compare the motion of the user with that of an experienced golfer. But, even the most minute of speed differences would render a frame by frame comparison useless. Instead, we put in place a system that takes and compares the user’s relevant data for comparison at five fixed stages (Figure 5); the first stage only initializes the user. The comparison was determined by finding the difference between the preset angle measures and the user’s angle measures. In comparing angle measures at these positions, we decided not to return a Boolean success or failure to the user, and instead established thresholds between separate levels of success to inform the user how much we quantify his or her error(s). In addition, the sign of the difference of the two angles can be read as not just an error margin, but an error direction. By observing the direction of the user’s error, we can display advice to correct his/her swing that is specific to what changes are needed the most.

<table>
<thead>
<tr>
<th>Stage Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Stage</strong></td>
</tr>
<tr>
<td><strong>Shoulder</strong></td>
</tr>
<tr>
<td><strong>Position</strong></td>
</tr>
<tr>
<td><strong>Distance</strong></td>
</tr>
<tr>
<td><strong>Left Elbow</strong></td>
</tr>
<tr>
<td><strong>Angle</strong></td>
</tr>
<tr>
<td><strong>Right Elbow</strong></td>
</tr>
<tr>
<td><strong>Hand</strong></td>
</tr>
</tbody>
</table>

Figure 5: A data table showing the different metrics used to differentiate between the five stages.

For comparison purposes, data for a baseline golfer had to be hardcoded directly into the program. Due to not starting with great knowledge in golf and the technique it requires, we initially set these values according to online golf tutorials. The result was a rigid baseline that allowed for little error, but it was an ideal starting point. From there we attempted to recreate the data (thresholding) with our own bodies which would give us data that was more organic.

As these measurements were compared to the preset values of the professional golfer, the user progressed from stage to stage. The total five stages represented the most significant positions in a golf swing (Figure 6).
4.2.3 Advice

Next, the program output constructive advice about the user’s strengths and weaknesses in each stage and throughout the entire application. Furthermore, point values were assigned to each skill level and the current performance was compared to the previous performance. This showed the trends in skill level, allowing for a more interactive and personalized learning tool. This program runs over and over again since the functions loop back. Thus, the user can continue to practice the golf swing as long as he or she wants. When the user wants to stop, he or she can simply end the application entirely.

4.2.4 Challenges

A number of obstacles barred the way to our writing a functional golf-swing program. Among these were the issues of actually determining what to use as metrics of both stage transition points and what made a swing successful at each point. Our solution for the first stage had us using the relative depth of both shoulders to determine the stages, so the user would progress as he or she rotated his or her upper body to perform a swing. With the second issue, we had tried to use joint positions as metrics to measure success; however body shape and proportion can easily vary across people. One constant however is angle measurements made at certain joint positions. From this we began to test the stages and we discovered that the user was always in one stage, and thus always having their values being recorded. A simple solution was to assign each stage a binary value initialized to zero, and as the user would pass through a stage this value would become one and deactivate for the rest of the program’s duration. While it did not create our next problem, it helped us to notice it; the stages could be completed out of the correct order. In addition to a stage being inaccessible when it had already been executed, we had to disbar it from activating when the stage immediately before it in a proper golf-swing had not been done. Lastly, a challenge we could not overcome was the ability to display a model of the ‘expert’ used in our program’s comparisons. To do so would require a programming technique called "threading" that allows for the computer to run multiple processes at a time, as compared to doing them in rapid sequence. In many programs rapid sequence is a sufficient replacement; however, simultaneous video output within one program would not be such a case. Without knowledge and experience in threading and a limited amount of time to learn, the feature had to be excluded from the most recent iteration of our program.

5 Results and Analysis

In testing the golf swing analysis program, we found it to consistently succeed in tracking the user’s golf swing. Every time a user ran the program, it took accurate measurements and compared the user’s swing to the expert swing built into the program (Figure 7). Additionally, it successfully determined when he or she reached the five different stages (Figure 8).
At the conclusion of the user’s swing, the program output constructive advice (e.g. “Stage 2: Your left arm was not straight enough.”) so that the player could work on improving his or her swing. Moreover, the user received a point value out of 100, providing a convenient way for users to gauge improvement. This value was always consistent with the user’s performance.

After each swing, the program automatically returned to stage 0, the preparatory stage, to allow the user to swing again if he or she wished to do so. This allowed for evaluation of the program’s advice, which we consistently found to be helpful in improving the user’s score.

6 Conclusions

We can draw several conclusions from our work with the Kinect with regard to the golf swing analysis. First, while we worked to determine the best method of comparison between the user’s golf swing and that of our expert, we realized that angle measures of certain joints would be the most effective and feasible way to do so. Our rationale for using angle measures instead of distance-based methods (e.g. how far away the hands are from the center of the body) for comparison was that people have differently proportioned bodies. We noted this when we tested our program in the early stages among members of our group; the program would not reliably detect or measure our golf swings. Thus, we conclude that angle measures, which remain proportional even for different people, are the optimal way of comparing one person’s golf swing to that of someone else.

Secondly, we realized that there were many angles to keep track of when comparing two golf swings. We had to make sure that we successfully tracked the most important angles. At each stage, we measured not only the user’s elbow angle, but his or her hip angle (in addition to other minor angles) as well. Through testing and experimentation, we determined what angles were necessary to compare a golf swing, learning in the process that several angles that we at first thought trivial were in fact crucial.

Thirdly, we see that this golf swing analysis tool is very practical, efficient, and cost-effective. In comparison to the hefty
price of $1000 for the V1 Pro software, the Kinect as a standalone product costs only $100 [6]. This is the only cost, since no other accessories or tools are needed to use this application. The Kinect is a very easy tool for new users to learn and is easily accessible to all people due to its commercial nature. This is a much more beneficial choice for golfers who want to improve their swing at a lower price with an application that is very reliable.

In sum, from our work with the golf swing analysis program in conjunction with the Kinect, we have determined that angles are the most accurate way of comparing golf swings, that many angles are indeed crucial when comparing golf swings, and that the Kinect can be a cost-effective alternative to expensive, traditional methods of athletic training.

7 Future Applications

Our use of the Kinect to interface with a computer has demonstrated its efficacy in comparing the user’s motions to those of a pre-set model. As of now, the pre-set model is gathered from proper golf techniques shown in the reliable literature. In the future, though, this simulation should involve the skeleton of the user to be compared to another person (either prerecorded or generated in real-time). This would allow the user’s skeleton to be overlaid on that of the professional by measuring the necessary angles and lengths. This tool would make direct comparison even easier since specific areas of strengths and weaknesses could be highlighted and shown clearly.

This use of the Kinect opens the door for further applications involving body motion and joint recognition. This utilization of the technology has already shown its worth in athletic applications. In particular, it has demonstrated its use in helping athletes correct their golf swing. It can be used in other athletic application such as tennis swings, football throws, and baseball swings. Athletes will be able to perfect their form with the help of the Kinect’s motion tracking and depth measuring capabilities, rather than relying on their own judgment or that of a coach, which might be imprecise.

Additionally, our work can be expanded to uses beyond athletic ones; the Kinect technology has the capability of being applied to medical issues. For instance, rehabilitation of injuries will be facilitated with the aid of the Kinect’s abilities for motion tracking and depth measurement. Individuals in need of recovery can mimic the prerecorded motions of a healthy individual. The Kinect can track the injured individual’s joints and then compare his or her motion to those of a healthy individual. In this way, the patient will be able to progress through rehabilitation more rapidly and accurately than would be possible through traditional methods. With what we have been able to achieve with the Kinect in four weeks, we hold high hopes that it will continue to benefit the athletic training and medical rehabilitation fields in groundbreaking ways.

8 Acknowledgements

We would like to express our sincere gratitude to our project mentors, Ivan Seskar, Roy D. Yates, Kyle Soska, Shaun Kotikalapudi, and Chris Jelesnianski for guiding us throughout the research project. We would also like to thank our project advisor, Pranav Challa, for providing us with invaluable advice and every resource we needed. We would like to recognize the New Jersey Governor’s School of
Engineering and Technology (Program Director Jean Patrick Antoine), NJ Governor’s School Board of Overseers, Dean Ilene Rosen, Head Counselor Daniel Cobar, and all our RTAs for organizing this program. Finally, this program would not have been possible without the generous contributions of our sponsors: Rutgers University, the Rutgers University School of Engineering, Morgan Stanley, the State of New Jersey, Lockheed Martin, Automated Control Concepts Inc., Silver Line Building Products, Sharon Ma and Nan Yao, the Tomasetta family, Laura Overdeck, and the families of the 2001-2010 program alumni.

9 References


10 Appendix: Golf Swing Analysis Program Code

// -------------------------------
// Includes
// -------------------------------
#include "SceneDrawer.h"
#include <cmath>
#include <iostream>

bool stage0Tracker = 0;
bool stage1Tracker = 0;
bool stage2Tracker = 0;
bool stage3Tracker = 0;
bool stage4Tracker = 0;
bool stage5Tracker = 0;
int perfectCount = 0;
int goodCount = 0;
int fairCount = 0;
int poorCount = 0;
int lastScore = 0;
int compareScore = 0;
int goodnessPoints = 0;

#if (XN_PLATFORM == XN_PLATFORM_MACOSX)
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#endif

extern xn::UserGenerator g_UserGenerator;
extern xn::DepthGenerator g_DepthGenerator;
extern XnBool g_bDrawBackground;
extern XnBool g_bDrawPixels;
extern XnBool g_bDrawSkeleton;
extern XnBool g_bPrintID;
extern XnBool g_bPrintState;

#define MAX_DEPTH 10000
float g_pDepthHist[MAX_DEPTH];

unsigned int getClosestPowerOfTwo(unsigned int n)
{
    unsigned int m = 2;
    while(m < n) m<<=1;
    return m;
}

GLuint initTexture(void** buf, int& width, int& height)
{
    GLuint texID = 0;
    glGenTextures(1,&texID);
    width = getClosestPowerOfTwo(width);
    height = getClosestPowerOfTwo(height);
    *buf = new unsigned char[width*height*4];
    glBindTexture(GL_TEXTURE_2D,texID);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    return texID;
}

GLfloat texcoords[8];

void DrawRectangle(float topLeftX, float topLeftY, float bottomRightX, float bottomRightY)
{
    GLfloat verts[8] = {    topLeftX, topLeftY,
    topLeftX, bottomRightY,
    bottomRightX, bottomRightY,
    bottomRightX, topLeftY
    };
    glVertexPointer(2, GL_FLOAT, 0, verts);
    glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
    //To Do: Maybe glFinish needed here instead - if there's some bad graphics
    glFlush();
}

void DrawTexture(float topLeftX, float topLeftY, float bottomRightX, float bottomRightY)


```c
XnFloat Colors[][3] = {
{0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.5, 0.5, 0.5, 0.5, 1.0},
{0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.5, 0.5, 0.5, 0.5, 1.0},
{0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.5, 0.5, 0.5, 0.5, 1.0},
{0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.5, 0.5, 0.5, 0.5, 1.0},
{0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.5, 0.5, 0.5, 0.5, 1.0},
{0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.5, 0.5, 0.5, 0.5, 1.0},
{0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.5, 0.5, 0.5, 0.5, 1.0},
{0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.5, 0.5, 0.5, 0.5, 1.0},
{0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.5, 0.5, 0.5, 0.5, 1.0},
{0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.5, 0.5, 0.5, 0.5, 1.0},
{0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.5, 0.5, 0.5, 0.5, 1.0},
{0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.5, 0.5, 0.5, 0.5, 1.0},
{0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.5, 0.5, 0.5, 0.5, 1.0}
};

XnUInt32 nColors = 10;
void glPrintString(void *font, char *str)
{
    int i = strlen(str);
    for(i=0; i<l; i++)
    
    glutBitmapCharacter(font, *str++);
}

// Define the getAngleXY function
float getAngleXY(XnSkeletonJointPosition ja, XnSkeletonJointPosition jb, XnSkeletonJointPosition jc)
{
    float x, a, b, c = 0;
    x = acos((a*a)+(b*b)-(c*c))/(2*a*b);
    x = (x*180)/3.14;
    return x;
}

// Define the getAngleXZ function
float getAngleXZ(XnSkeletonJointPosition ja, XnSkeletonJointPosition jb, XnSkeletonJointPosition jc)
{
    float x, a, b, c = 0;
    x = acos((a*a)+(b*b)-(c*c))/(2*a*b);
    x = (x*180)/3.14;
    return x;
}

// Define the getAngleYZ function
float getAngleYZ(XnSkeletonJointPosition ja, XnSkeletonJointPosition jb, XnSkeletonJointPosition jc)
{
    float x, a, b, c = 0;
    x = acos((a*a)+(b*b)-(c*c))/(2*a*b);
    x = (x*180)/3.14;
    return x;
}

void DrawLimb(XnUserID player, XnSkeletonJoint eJoint1, XnSkeletonJoint eJoint2)
{
    if (!g_UserGenerator.GetSkeletonCap().IsTracking(player))
    
    printf("not tracked!\n");
    return;
}
```
//printf("Joint Number(1) - %i, Joint Position (%f, %f, %f), Confidence - %f\n", eJoint1, joint1.position.X, joint1.position.Y, joint1.position.Z, joint1.fConfidence);
g_UserGenerator.GetSkeletonCap().GetSkeletonJointPosition(player, eJoint1, joint2);
//printf("Joint Number(2) - %i, Joint Position (%f, %f, %f), Confidence - %f\n", eJoint2, joint2.position.X, joint2.position.Y, joint2.position.Z, joint2.fConfidence);

// Find the hip positions
float hipAvgX = (leftHip.position.X + rightHip.position.X) / 2;
float hipAvgY = (leftHip.position.Y + rightHip.position.Y) / 2;
float hipAvgZ = (leftHip.position.Z + rightHip.position.Z) / 2;

// Define relative coordinates with respect to the hips
float relativeCoordinate1X = joint1.position.X - hipAvgX;
float relativeCoordinate1Y = joint1.position.Y - hipAvgY;
float relativeCoordinate1Z = joint1.position.Z - hipAvgZ;
float relativeCoordinate2X = joint2.position.X - hipAvgX;
float relativeCoordinate2Y = joint2.position.Y - hipAvgY;
float relativeCoordinate2Z = joint2.position.Z - hipAvgZ;

// Print the relative coordinates
//printf("Joint Number(1) - %i, Joint Position (%f, %f, %f), Confidence - %f\n", eJoint1, relativeCoordinate1X, relativeCoordinate1Y, relativeCoordinate1Z, joint1.fConfidence);
//printf("Joint Number(2) - %i, Joint Position (%f, %f, %f), Confidence - %f\n", eJoint2, relativeCoordinate2X, relativeCoordinate2Y, relativeCoordinate2Z, joint2.fConfidence);

// Declare variables for required joints (other than the hips)
g_UserGenerator.GetSkeletonCap().GetSkeletonJointPosition(player, XN_SKEL_RIGHT_ELBOW, rightElbow);
g_UserGenerator.GetSkeletonCap().GetSkeletonJointPosition(player, XN_SKEL_RIGHT_SHOULDER, rightShoulder);
g_UserGenerator.GetSkeletonCap().GetSkeletonJointPosition(player, XN_SKEL_RIGHT_HAND, rightHand);
g_UserGenerator.GetSkeletonCap().GetSkeletonJointPosition(player, XN_SKEL_LEFT_ELBOW, leftElbow);
g_UserGenerator.GetSkeletonCap().GetSkeletonJointPosition(player, XN_SKEL_LEFT_SHOULDER, leftShoulder);
g_UserGenerator.GetSkeletonCap().GetSkeletonJointPosition(player, XN_SKEL_LEFT_HAND, leftHand);

// Define necessary angles
float rightElbowXY, rightElbowXZ, rightElbowYZ;
float leftElbowXY, leftElbowXZ, leftElbowYZ;
float rightHipXY, rightHipXZ, rightHipYZ;
float leftHipXY, leftHipXZ, leftHipYZ;
float rightShoulderXY, rightShoulderXZ, rightShoulderYZ;
float leftShoulderXY, leftShoulderXZ, leftShoulderYZ;
float handDistance = rightHand.position.X - leftHand.position.X;

// Find the right elbow angle
rightElbowXY = getAngleXY(rightShoulder, rightHand, rightElbow);
rightElbowXZ = getAngleXZ(rightShoulder, rightHand, rightElbow);
rightElbowYZ = getAngleYZ(rightShoulder, rightHand, rightElbow);
//printf("Right Elbow Angle (%f, %f, %f)\n", rightElbowXY, rightElbowXZ, rightElbowYZ);

// Find the left elbow angle
leftElbowXY = getAngleXY(leftShoulder, leftHand, leftElbow);
leftElbowXZ = getAngleXZ(leftShoulder, leftHand, leftElbow);
leftElbowYZ = getAngleYZ(leftShoulder, leftHand, leftElbow);
//printf("Left Elbow Angle (%f, %f, %f)\n", leftElbowXY, leftElbowXZ, leftElbowYZ);

// Find the right hip angle
rightHipXY = getAngleXY(rightElbow, rightShoulder, rightHip);
rightHipXZ = getAngleXZ(rightElbow, rightShoulder, rightHip);
rightHipYZ = getAngleYZ(rightElbow, rightShoulder, rightHip);
//printf("Right Hip Angle (%f, %f, %f)\n", rightHipXY, rightHipXZ, rightHipYZ);

// Find the left hip angle
leftHipXY = getAngleXY(leftElbow, leftShoulder, leftHip);
leftHipXZ = getAngleXZ(leftElbow, leftShoulder, leftHip);
leftHipYZ = getAngleYZ(leftElbow, leftShoulder, leftHip);
//printf("Left Hip Angle (%f, %f, %f)\n", leftHipXY, leftHipXZ, leftHipYZ);

// Find the right shoulder angle
rightShoulderXY = getAngleXY(rightHip, rightElbow, rightShoulder);
rightShoulderXZ = getAngleXZ(rightHip, rightElbow, rightShoulder);
rightShoulderYZ = getAngleYZ(rightHip, rightElbow, rightShoulder);
//printf("Right Shoulder Angle (%f, %f, %f)\n", rightShoulderXY, rightShoulderXZ, rightShoulderYZ);

// Find the left shoulder angle
leftShoulderXY = getAngleXY(leftHip, leftElbow, leftShoulder);
leftShoulderXZ = getAngleXZ(leftHip, leftElbow, leftShoulder);
leftShoulderYZ = getAngleYZ(leftHip, leftElbow, leftShoulder);
//printf("Left Shoulder Angle (%f, %f, %f)\n", leftShoulderXY, leftShoulderXZ, leftShoulderYZ);

// Find the distance for hands touching
float handDistance = rightHand.position.X - leftHand.position.X;
//printf("Hand Distance %f\n", handDistance);
// Declare what needs to be tested in each stage
//bool stage1HandsTouching;
//bool stage1LeftArmStraight;
//bool stage1HandsBelowWaist;
//bool stage2LeftArmStraight;
//bool stage2HandsBelowWaist;
bool stage2LeftArmStraight;
bool stage2RightArmObtuse;
bool stage2HandsBelowWaist;
bool stage3LeftArmStraight;
bool stage3RightArmAcute;
bool stage3HandsBelowWaist;
bool stage4LeftArmStraight;
bool stage4RightArmStraight;
bool stage4HandsBelowWaist;
bool stage5LeftArmBent;
bool stage5RightArmBent;
bool stage5HandsBelowWaist;
float RXY[5];
float RXZ[5];
float RYZ[5];
float LXY[5];
float LXZ[5];
float LYZ[5];
RXY[0] = 160;
RXY[1] = 161;
RXY[2] = 127;
RXY[3] = 127;
RXY[4] = 157;
RXZ[0] = 168;
RXZ[1] = 168;
RXZ[2] = 178;
RXZ[3] = 178;
RXZ[4] = 151;
RYZ[0] = 165;
RYZ[1] = 161;
RYZ[2] = 127;
RYZ[3] = 160;
RYZ[4] = 163;
LXY[0] = 151;
LXY[1] = 147;
LXY[2] = 154;
LXY[3] = 122;
LXY[4] = 160;
LXZ[0] = 144;
LXZ[1] = 134;
LXZ[2] = 120;
LXZ[3] = 146;
LXZ[4] = 152;
LYZ[0] = 164;
LYZ[1] = 154;
LYZ[2] = 134;
LYZ[3] = 154;
LYZ[4] = 136;
int i = 0;

// Determine thresholds for hand and shoulder distances
float shouldersZDifference = rightShoulder.position.Z - leftShoulder.position.Z;
int stageIndicator;
if(abs(shouldersZDifference) <= 50 && handDistance <= 80)
{
    stageIndicator = 1;
}
else if(shouldersZDifference > 50 && shouldersZDifference <= 250)
{
    stageIndicator = 2;
}
else if(shouldersZDifference > 250 && shouldersZDifference < 500)
{
    stageIndicator = 3;
}
else if(shouldersZDifference < -50 && shouldersZDifference >= -250)
{
    stageIndicator = 4;
}
else if(shouldersZDifference < -250 && shouldersZDifference > -500)
{
    stageIndicator = 5;
}
else
{
    stageIndicator = 0;
}
// Check stages using CompareY and test for proper golf form by checking important distances and angles
// -1 for no stage, start with stage 0
switch (stageIndicator) {
    case 0:
        if (stage0Tracker == 0) {
            printf("****************************
******Currently in stage 0.******
****************************");
            stage0Tracker = 1;
        } else {
            return;
        }
        break;
    case 1:
        if (stage1Tracker == 0 && stage0Tracker == 1) {
            printf("****************************
******Currently in stage 1.******
****************************");
            if (handDistance <= 80) {
                stage1HandsTouching = 1;
            } else {
                stage1HandsTouching = 0;
            }
            if (leftElbowXY >= 166 && leftElbowXY <= 180) {
                stage1LeftArmStraight = 1;
            } else {
                stage1LeftArmStraight = 0;
            }
            if (rightHand.position.Y - hipAvgY < 0 && leftHand.position.Y - hipAvgY < 0) {
                stage1HandsBelowWaist = 1;
            } else {
                stage1HandsBelowWaist = 0;
            }
            rightElbowXY = rightElbowXY - RXY[0];
            if (rightElbowXY < 0) {
                rightElbowXY = rightElbowXY*-1;
            }
            rightElbowXZ = rightElbowXZ - RXZ[0];
            if (rightElbowXZ < 0) {
                rightElbowXZ = rightElbowXZ*-1;
            }
            rightElbowYZ = rightElbowYZ - RYZ[0];
            if (rightElbowYZ < 0) {
                rightElbowYZ = rightElbowYZ*-1;
            }
            leftElbowXY = leftElbowXY - LXY[0];
            if (leftElbowXY < 0) {
                leftElbowXY = leftElbowXY*-1;
            }
            leftElbowXZ = leftElbowXZ - LXZ[0];
            if (leftElbowXZ < 0) {
                leftElbowXZ = leftElbowXZ*-1;
            }
            } // end switch
leftElbowYZ = leftElbowYZ - LYZ[0];
if (leftElbowYZ < 0)
{
    leftElbowYZ = leftElbowYZ*-1;
}
printf("Right Elbow Angle (%f, %f,%f)\n", rightElbowXY, rightElbowXZ, rightElbowYZ);
printf("Left Elbow Angle (%f, %f,%f)\n", leftElbowXY, leftElbowXZ, leftElbowYZ);
//printf("Right Hip Angle (%f, %f,%f)\n", rightHipXY, rightHipXZ, rightHipYZ);
//printf("Left Hip Angle (%f, %f,%f)\n", leftHipXY, leftHipXZ, leftHipYZ);
//printf("RightShoulder Angle (%f, %f,%f)\n", rightShoulderXY, rightShoulderXZ, rightShoulderYZ);
//printf("LeftShoulder Angle (%f, %f,%f)\n", leftShoulderXY, leftShoulderXZ, leftShoulderYZ);
stage1Tracker = 1;
}
else{
    return;
}
break;
case 2:
if(stage2Tracker == 0 && stage1Tracker == 1)
{
    printf("***********************************\n"Currently in stage 2.********\n"Currently in stage 2.********\n"");
    if(leftElbowXZ >= 150 && leftElbowXZ <= 180)
    {
        stage2LeftArmStraight = 1;
    }
    else
    {
        stage2LeftArmStraight = 0;
    }
    if(rightElbowXY >= 90 && rightElbowXY <= 180)
    {
        stage2RightArmObtuse = 1;
    }
    else
    {
        stage2RightArmObtuse = 0;
    }
    if(rightHand.position.Y - hipAvgY < 0 && leftHand.position.Y - hipAvgY < 0)
    {
        stage2HandsBelowWaist = 1;
    }
    else
    {
        stage2HandsBelowWaist = 0;
    }
    rightElbowXY = rightElbowXY - RXY[1];
    if (rightElbowXY < 0)
    {
        rightElbowXY = rightElbowXY*-1;
    }
    rightElbowXZ = rightElbowXZ - RXZ[1];
    if (rightElbowXZ < 0)
    {
        rightElbowXZ = rightElbowXZ*-1;
    }
    rightElbowYZ = rightElbowYZ - RYZ[1];
    if (rightElbowYZ < 0)
    {
        rightElbowYZ = rightElbowYZ*-1;
    }
    rightElbowXY = rightElbowXY - RXY[1];
    if (rightElbowXY < 0)
    {
        rightElbowXY = rightElbowXY*-1;
    }
    rightElbowXZ = rightElbowXZ - RXZ[1];
    if (rightElbowXZ < 0)
    {
        rightElbowXZ = rightElbowXZ*-1;
    }
    rightElbowYZ = rightElbowYZ - RYZ[1];
    if (rightElbowYZ < 0)
    {
        rightElbowYZ = rightElbowYZ*-1;
    }
    leftElbowXY = leftElbowXY - LXY[1];
    if (leftElbowXY < 0)
    {
        leftElbowXY = leftElbowXY*-1;
    }
    leftElbowXZ = leftElbowXZ - LXZ[1];
    if (leftElbowXZ < 0)
    {
        leftElbowXZ = leftElbowXZ*-1;
    }
    leftElbowYZ = leftElbowYZ - LYZ[1];
    if (leftElbowYZ < 0)
    {
        leftElbowYZ = leftElbowYZ*-1;
    }
}
leftElbowYZ = leftElbowYZ * -1;
}

stage2Tracker = 1;
printf("Right Elbow Angle (%f, %f, %f)\n", rightElbowXY, rightElbowXZ, rightElbowYZ);
printf("Left Elbow Angle (%f, %f, %f)\n", leftElbowXY, leftElbowXZ, leftElbowYZ);
printf("Right Arm Performance: ");
if (abs(rightElbowXY) + abs(rightElbowXZ) + abs(rightElbowYZ) < 15) {
    printf("Perfect!\n");
    perfectCount = perfectCount + 1;
} else if (abs(rightElbowXY) + abs(rightElbowXZ) + abs(rightElbowYZ) < 30) {
    printf("Good\n");
    goodCount = goodCount + 1;
} else if (abs(rightElbowXY) + abs(rightElbowXZ) + abs(rightElbowYZ) < 60) {
    printf("Fair\n");
    fairCount = fairCount + 1;
} else {
    printf("Poor\n");
    poorCount = poorCount + 1;
}

printf("Left Arm Performance: ");
if (abs(leftElbowXY) + abs(leftElbowXZ) + abs(leftElbowYZ) < 15) {
    printf("Perfect!\n");
    perfectCount = perfectCount + 1;
} else if (abs(leftElbowXY) + abs(leftElbowXZ) + abs(leftElbowYZ) < 30) {
    printf("Good\n");
    goodCount = goodCount + 1;
} else if (abs(leftElbowXY) + abs(leftElbowXZ) + abs(leftElbowYZ) < 60) {
    printf("Fair\n");
    fairCount = fairCount + 1;
} else {
    printf("Poor\n");
    poorCount = poorCount + 1;
}
else{
    return;
}
break;
case 3:
    if (stage3Tracker == 0 && stage2Tracker == 1) {
        printf("*****************************************************************************\nCurrent in stage 3.*****************************************************************************\n");
        if (leftElbowXZ >= 150 && leftElbowXZ <= 180) {
            stage3LeftArmStraight = 1;
        } else {
            stage3LeftArmStraight = 0;
        }
        if (rightElbowXY >= 0 && rightElbowXY < 90) {
            stage3RightArmAcute = 1;
        } else {
        }
    }
stage3RightArmAcute = 0;
}
if(rightHand.position.Y - hipAvgY < 0 && leftHand.position.Y - hipAvgY < 0)
{
    stage3HandsBelowWaist = 1;
} else
{
    stage3HandsBelowWaist = 0;
}
rightElbowXY = rightElbowXY - RXY[2];
if (rightElbowXY < 0)
{
    rightElbowXY = rightElbowXY*-1;
}
rightElbowXZ = rightElbowXZ - RXZ[2];
if (rightElbowXZ < 0)
{
    rightElbowXZ = rightElbowXZ*-1;
}
rightElbowYZ = rightElbowYZ - RYZ[2];
if (rightElbowYZ < 0)
{
    rightElbowYZ = rightElbowYZ*-1;
}
leftElbowXY = leftElbowXY - LXY[2];
if (leftElbowXY < 0)
{
    leftElbowXY = leftElbowXY*-1;
}
leftElbowXZ = leftElbowXZ - LXZ[2];
if (leftElbowXZ < 0)
{
    leftElbowXZ = leftElbowXZ*-1;
}
leftElbowYZ = leftElbowYZ - LYZ[2];
if (leftElbowYZ < 0)
{
    leftElbowYZ = leftElbowYZ*-1;
}
stage3Tracker = 1;
printf("Right Elbow Angle (%f, %f,%f)\n", rightElbowXY, rightElbowXZ, rightElbowYZ);
printf("Left Elbow Angle (%f, %f,%f)\n", leftElbowXY, leftElbowXZ, leftElbowYZ);
printf("Right Arm Performance: ");
if (abs(rightElbowXY) + abs(rightElbowXZ) + abs(rightElbowYZ) < 15)
{
    printf("Perfect! \n");
    perfectCount = perfectCount+1;
} else if (abs(rightElbowXY) + abs(rightElbowXZ) + abs(rightElbowYZ) < 30)
{
    printf("Good \n");
    goodCount=goodCount+1;
} else if (abs(rightElbowXY) + abs(rightElbowXZ) + abs(rightElbowYZ) < 60)
{
    printf("Fair \n");
    fairCount=fairCount+1;
} else
{
    printf("Poor \n");
    poorCount=poorCount+1;
}
printf("Left Arm Performance: ");
if (abs(leftElbowXY) + abs(leftElbowXZ) + abs(leftElbowYZ) < 15)
{
    printf("Perfect! \n");
    perfectCount = perfectCount+1;
} else if (abs(leftElbowXY) + abs(leftElbowXZ) + abs(leftElbowYZ) < 30)
printf("Good \n");
goodCount=goodCount+1;
} else if (abs(leftElbowXY) + abs(leftElbowXZ) + abs(leftElbowYZ) < 60)
{
printf("Fair \n");
fairCount=fairCount+1;
} else
{
printf("Poor \n");
poorCount=poorCount+1;
}
} else
{
return;
}
break;

case 4:
if(stage4Tracker == 0 && stage3Tracker == 1)
{
printf("***********************************
***********************************
*******Currently in stage 4.*******
***********************************
***********************************
");
if(leftElbowXZ >= 166 && leftElbowXZ <= 180)
{
    stage4LeftArmStraight = 1;
}
else
{
    stage4LeftArmStraight = 0;
}
if(rightElbowXY >= 166 && rightElbowXY <= 180)
{
    stage4RightArmStraight = 1;
}
else
{
    stage4RightArmStraight = 0;
}
if(rightHand.position.Y - hipAvgY < 0 && leftHand.position.Y - hipAvgY < 0)
{
    stage4HandsBelowWaist = 1;
}
else
{
    stage4HandsBelowWaist = 0;
}
rightElbowXY = rightElbowXY - RXY[3];
if (rightElbowXY < 0)
{
    rightElbowXY = rightElbowXY*-1;
}
rightElbowXZ = rightElbowXZ - RXZ[3];
if (rightElbowXZ < 0)
{
    rightElbowXZ = rightElbowXZ*-1;
}
rightElbowYZ = rightElbowYZ - RYZ[3];
if (rightElbowYZ < 0)
{
    rightElbowYZ = rightElbowYZ*-1;
}
leftElbowXY = leftElbowXY - LXY[3];
if (leftElbowXY < 0)
{
    leftElbowXY = leftElbowXY*-1;
}
leftElbowXZ = leftElbowXZ - LXZ[3];
if (leftElbowXZ < 0)
{
    leftElbowXZ = leftElbowXZ*-1;
}
leftElbowYZ = leftElbowYZ - LYZ[3];
if (leftElbowYZ < 0)
{
    leftElbowYZ = leftElbowYZ*-1;
}
printf("Right Elbow Angle (%f, %f,%f)\n", rightElbowXY, rightElbowXZ, rightElbowYZ);
printf("Left Elbow Angle (%f, %f,%f)\n", leftElbowXY, leftElbowXZ, leftElbowYZ);
printf("Right Arm Performance: ");
if (abs(rightElbowXY) + abs(rightElbowXZ) + abs(rightElbowYZ) < 15)
{
    printf("Perfect! \n");
    perfectCount = perfectCount+1;
}
else if (abs(rightElbowXY) + abs(rightElbowXZ) + abs(rightElbowYZ) < 30)
{
    printf("Good \n");
    goodCount=goodCount+1;
}
else if (abs(rightElbowXY) + abs(rightElbowXZ) + abs(rightElbowYZ) < 60)
{
    printf("Fair \n");
    fairCount=fairCount+1;
}
else
{
    printf("Poor \n");
    poorCount=poorCount+1;
}
printf("Left Arm Performance: ");
if (abs(leftElbowXY) + abs(leftElbowXZ) + abs(leftElbowYZ) < 15)
{
    printf("Perfect! \n");
    perfectCount = perfectCount+1;
}
else if (abs(leftElbowXY) + abs(leftElbowXZ) + abs(leftElbowYZ) < 30)
{
    printf("Good \n");
    goodCount=goodCount+1;
}
else if (abs(leftElbowXY) + abs(leftElbowXZ) + abs(leftElbowYZ) < 60)
{
    printf("Fair \n");
    fairCount=fairCount+1;
}
else
{
    printf("Poor \n");
    poorCount=poorCount+1;
}
stage4Tracker = 1;
}
else{
    return;
}
break;
case 5:
if(stage5Tracker == 0 && stage4Tracker == 1)
{
    printf("******************************************************************************\nCurrently in stage 5******************************************************************************\n");
    if(leftElbowXY < 180)
    {
        stage5LeftArmBent = 1;
    }
    else
    {
        stage5LeftArmBent = 0;
    }
    if(rightElbowXY < 180)
    {
        stage5RightArmBent = 1;
    }
    else
    {
        stage5RightArmBent = 0;
    }
if(rightHand.position.Y - hipAvgY < 0 && leftHand.position.Y - hipAvgY < 0)
{
    stage5HandsBelowWaist = 1;
}
else
{
    stage5HandsBelowWaist = 0;
}
rightElbowXY = rightElbowXY - RXY[4];
if (rightElbowXY < 0)
{
    rightElbowXY = rightElbowXY*-1;
}
rightElbowXZ = rightElbowXZ - RXZ[4];
if (rightElbowXZ < 0)
{
    rightElbowXZ = rightElbowXZ*-1;
}
rightElbowYZ = rightElbowYZ - RYZ[4];
if (rightElbowYZ < 0)
{
    rightElbowYZ = rightElbowYZ*-1;
}
leftElbowXY = leftElbowXY - LXY[4];
if (leftElbowXY < 0)
{
    leftElbowXY = leftElbowXY*-1;
}
leftElbowXZ = leftElbowXZ - LXZ[4];
if (leftElbowXZ < 0)
{
    leftElbowXZ = leftElbowXZ*-1;
}
leftElbowYZ = leftElbowYZ - LYZ[4];
if (leftElbowYZ < 0)
{
    leftElbowYZ = leftElbowYZ*-1;
}
printf("Right Elbow Angle (%f, %f,%f)
", rightElbowXY, rightElbowXZ, rightElbowYZ);
printf("Left Elbow Angle (%f, %f,%f)
", leftElbowXY, leftElbowXZ, leftElbowYZ);
printf("Right Arm Performance: ");
if (abs(rightElbowXY) + abs(rightElbowXZ) + abs(rightElbowYZ) < 15)
{
    printf("Perfect! 
");
    perfectCount = perfectCount+1;
}
else if (abs(rightElbowXY) + abs(rightElbowXZ) + abs(rightElbowYZ) < 30)
{
    printf("Good 
");
    goodCount = goodCount+1;
}
else if (abs(rightElbowXY) + abs(rightElbowXZ) + abs(rightElbowYZ) < 60)
{
    printf("Fair 
");
    fairCount = fairCount+1;
}
else
{
    printf("Poor 
");
    poorCount = poorCount+1;
}
printf("Left Arm Performance: ");
if (abs(leftElbowXY) + abs(leftElbowXZ) + abs(leftElbowYZ) < 15)
{
    printf("Perfect! 
");
    perfectCount = perfectCount+1;
}
else if (abs(leftElbowXY) + abs(leftElbowXZ) + abs(leftElbowYZ) < 30)
if (abs(leftElbowXY) + abs(leftElbowXZ) + abs(leftElbowYZ) < 60)
{
    printf("Fair \n");
    fairCount=fairCount+1;
}
else
{
    printf("Poor \n");
    poorCount=poorCount+1;
}
}

else{
}
}
break;
default:
break;
}

// Output messages showing strengths and weaknesses
if (stage5Tracker == 1)
{
    /*if (stage1HandsTouching == 0)
    {
        printf("Stage 1: Your hands were not touching\n");
    }
    if (stage1LeftArmStraight == 0)
    {
        printf("Stage 1: Your left arm was not straight\n");
    }
    */
    if (stage1HandsBelowWaist == 1)
    {
        printf("Stage 1: Your hands were not below your waist\n");
    }
    if (stage2LeftArmStraight == 0)
    {
        printf("Stage 2: Your left arm was not straight enough\n");
    }
    /*if (stage2RightArmObtuse == 0)
    {
        printf("Stage 2: Your right elbow was not bent enough\n");
    }
    if (stage2HandsBelowWaist==1)
    {
        printf("Stage 2: Your hands were not above your waist\n");
    }
    */
    if (stage3LeftArmStraight==0)
    {
        printf("Stage 3: Your left arm was not straight enough\n");
    }
    if (stage3RightArmAcute == 0)
    {
        printf("Stage 3: Your right elbow was not bent enough\n");
    }
    if (stage3HandsBelowWaist == 0)
    {
        printf("Stage 3: Your hands were not above your waist\n");
    }
    if (stage4LeftArmStraight == 0)
    {
        printf("Stage 4: Your left arm was not straight enough\n");
    }
    if (stage4RightArmStraight == 0)
    {
        printf("Stage 4: Your right arm was not straight enough\n");
    }
}
if (stage4HandsBelowWaist == 1)
{
    printf("Stage 4: Your hands were not above your waist\n");
}
if (stage5LeftArmBent==0)
{
    printf("Stage 5: Your left arm was not bent enough\n");
}
if (stage5RightArmBent==0)
{
    printf("Stage 5: Your right arm was not bent enough\n");
}
if (stage5HandsBelowWaist==1)
{
    printf("Stage 5: Your hands were not above your waist\n");
}

// Assign points for each skill; output how well the player performed
goodnessPoints = 14*perfectCount;
printf("You did perfect (%i) times\n",perfectCount);
goodnessPoints+= 12*goodCount;
printf("You did good (%i) times\n",goodCount);
goodnessPoints+= 11*fairCount;
printf("You did fairly well (%i) times\n",fairCount);
goodnessPoints+= 6*poorCount;
printf("You did poorly (%i) times\n",poorCount);
goodnessPoints = goodnessPoints - 12;
printf("You scored (%i) out of 100\n",goodnessPoints);

// Compare current performance to previous performance
compareScore = goodnessPoints - lastScore;
if (compareScore > 0)
{
    printf("You did (%i) points better than last round\n",compareScore);
}
else if (compareScore < 0)
{
    compareScore = compareScore * -1;
    printf("You did (%i) points worse than last round\n", compareScore);
}
else
{
    printf("You performed the same as last round\n");
}
stage0Tracker = 0;
stage1Tracker = 0;
stage2Tracker = 0;
stage3Tracker = 0;
stage4Tracker = 0;
stage5Tracker = 0;
lastScore = goodnessPoints;
perfectCount = 0;
goodCount = 0;
fairCount = 0;
poorCount = 0;
} /*if (joint1.fConfidence < 0.5 || joint2.fConfidence < 0.5)
 */

XnPoint3D pt[2];
pt[0] = joint1.position;
pt[1] = joint2.position;
g_DepthGenerator.ConvertRealWorldToProjective(2, pt, pt);
glVertex3i(pt[0].X, pt[0].Y, 0);
glVertex3i(pt[1].X, pt[1].Y, 0);
}
void DrawDepthMap(const xn::DepthMetaData& dmd, const xn::SceneMetaData& smd)
{
    static bool bInitialized = false;
    static GLuint depthTexID;
    static unsigned char* pDepthTexBuf;
    /*if (joint1.fConfidence < 0.5 || joint2.fConfidence < 0.5)
    */
    return;
    XnPoint3D pt[2];
    pt[0] = joint1.position;
    pt[1] = joint2.position;
    g_DepthGenerator.ConvertRealWorldToProjective(2, pt, pt);
    glVertex3i(pt[0].X, pt[0].Y, 0);
    glVertex3i(pt[1].X, pt[1].Y, 0);
}
static int texWidth, texHeight;
float topLeftX;
float topLeftY;
float bottomRightY;
float bottomRightX;
float texXpos;
float texYpos;
if(!bInitialized)
{
    texWidth = getClosestPowerOfTwo(dmd.XRes());
texHeight = getClosestPowerOfTwo(dmd.YRes());
    // printf("Initializing depth texture: width = %d, height = %d\n", texWidth, texHeight);
depthTexID = initTexture((void**)&pDepthTexBuf,texWidth, texHeight) ;
    // printf("Initialized depth texture: width = %d, height = %d\n", texWidth, texHeight);
bInitialized = true;
topLeftX = dmd.XRes();
topLeftY = 0;
bottomRightY = dmd.YRes();
bottomRightX = 0;
texXpos = (float)dmd.XRes()/texWidth;
texYpos = (float)dmd.YRes()/texHeight;
memset(texcoords, 0, 8*sizeof(float));
}
unsigned int nValue = 0;
unsigned int nHistValue = 0;
unsigned int nIndex = 0;
unsigned int nX = 0;
unsigned int nY = 0;
unsigned int nNumberOfPoints = 0;
XnUInt16 g_nXRes = dmd.XRes();
XnUInt16 g_nYRes = dmd.YRes();
unsigned char* pDestImage = pDepthTexBuf;
const XnDepthPixel* pDepth = dmd.Data();
const XnLabel* pLabels = smd.Data();
    // Calculate the accumulative histogram
memset(g_pDepthHist, 0, MAX_DEPTH*sizeof(float));
for (nY=0; nY<g_nYRes; nY++)
{
    for (nX=0; nX < g_nXRes; nX++, nIndex++)
    {
        nValue = *pDepth;
        if (nValue != 0)
        {
            g_pDepthHist[nValue]++;
            nNumberOfPoints++;
        }
        pDepth++;
    }
}
for (nIndex=1; nIndex<MAX_DEPTH; nIndex++)
{
    g_pDepthHist[nIndex] += g_pDepthHist[nIndex-1];
}
if (nNumberOfPoints)
{
    for (nIndex=1; nIndex<MAX_DEPTH; nIndex++)
    {
        g_pDepthHist[nIndex] = (unsigned int)(256 * (1.0f - (g_pDepthHist[nIndex] / nNumberOfPoints)));
    }
}
pDepth = dmd.Data();
if (g_bDrawPixels)
{
    XnUInt32 nIndex = 0;
    // Prepare the texture map
    for (nY=0; nY< g_nYRes; nY++)
    {
        for (nX=0; nX < g_nXRes; nX++, nIndex++)
        {
            pDestImage[nIndex] = 0;
        }
    }
}
pDestImage[1] = 0;
pDestImage[2] = 0;
if (g_bDrawBackground || *pLabels != 0)
{
  nValue = *pDepth;
  XnLabel label = *pLabels;
  XnUInt32 nColorID = label % nColors;
  if (label == 0)
  {
    nColorID = nColors;
  }
  if (nValue != 0)
  {
    nHistValue = g_pDepthHist[nValue];
    pDestImage[0] = nHistValue * Colors[nColorID][0];
    pDestImage[1] = nHistValue * Colors[nColorID][1];
    pDestImage[2] = nHistValue * Colors[nColorID][2];
  }
  pDepth++;
pLabels++;
pDestImage+=3;
}
pDestImage += (texWidth - g_nXRes) * 3;
else
{
  xnOSMemSet(pDepthTexBuf, 0, 3*2*g_nXRes*g_nYRes);
}
glBindTexture(GL_TEXTURE_2D, depthTexID);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, texWidth, texHeight, 0, GL_RGB, GL_UNSIGNED_BYTE, pDepthTexBuf);
// Display the OpenGL texture map
setColor4f(0.75,0.75,0.75,1);
glEnable(GL_TEXTURE_2D);
DrawTexture(dmd.XRes(),dmd.YRes(),0,0);
glColor4f(0.75,0.75,0.75,1);
glDisable(GL_TEXTURE_2D);
for (int i = 0; i < nUsers; ++i)
{
  if (g_bPrintID)
  {
    XnP3D com;
    g_UserGenerator.GetCoM(aUsers[i], com);
    g_DepthGenerator.ConvertRealWorldToProjective(1, &com, &com);
    xnOSMemSet(strLabel, 0, sizeof(strLabel));
    if (g_bPrintState)
    {
      // Tracking
      sprintf(strLabel, "%d", aUsers[i]);
      glRasterPos2i(com.X, com.Y);
      glPrintString(GLUT_BITMAP_HELVETICA_18, strLabel);
    }
  }
  else if (g_UserGenerator.GetSkeletonCap().IsTracking(aUsers[i]))
  {
    // Tracking
    sprintf(strLabel, "%d - Tracking", aUsers[i]);
  }
  else if (g_UserGenerator.GetSkeletonCap().IsCalibrating(aUsers[i]))
  {
    // Calibrating
    sprintf(strLabel, "%d - Calibrating", aUsers[i]);
  }
  else
  {
    // Nothing
    sprintf(strLabel, "%d - Looking for pose", aUsers[i]);
  }
  gColor4f(1-Colors[i%nColors][0], 1-Colors[i%nColors][1], 1-Colors[i%nColors][2], 1);
gRasterPos2i(com.X, com.Y);
gPrintString(GLUT_BITMAP_HELVETICA_18, strLabel);
if (g_bDrawSkeleton && g_UserGenerator.GetSkeletonCap().IsTracking(aUsers[i]))
{
    glBegin(GL_LINES);
    glColor4f(1-Colors[aUsers[i]%nColors][0], 1-Colors[aUsers[i]%nColors][1], 1-Colors[aUsers[i]%nColors][2], 1);
    DrawLimb(aUsers[i], XN_SKEL_HEAD, XN_SKEL_NECK);
    DrawLimb(aUsers[i], XN_SKEL_NECK, XN_SKEL_LEFT_SHOULDER);
    DrawLimb(aUsers[i], XN_SKEL_LEFT_SHOULDER, XN_SKEL_LEFT_ELBOW);
    DrawLimb(aUsers[i], XN_SKEL_LEFT_ELBOW, XN_SKEL_LEFT_HAND);
    //DrawLimb(aUsers[i], XN_SKEL_LEFT_FINGERTIP, XN_SKEL_LEFT_HAND);
    //DrawLimb(aUsers[i], XN_SKEL_LEFT_SHOULDER, XN_SKEL_LEFT_COLLAR);
    DrawLimb(aUsers[i], XN_SKEL_NECK, XN_SKEL_RIGHT_SHOULDER);
    DrawLimb(aUsers[i], XN_SKEL_RIGHT_SHOULDER, XN_SKEL_RIGHT_ELBOW);
    DrawLimb(aUsers[i], XN_SKEL_RIGHT_ELBOW, XN_SKEL_RIGHT_HAND);
    DrawLimb(aUsers[i], XN_SKEL_RIGHT_SHOULDER, XN_SKEL_RIGHT_COLLAR);
    DrawLimb(aUsers[i], XN_SKEL_RIGHT_SHOULDER, XN_SKEL_RIGHT_ELBOW);
    DrawLimb(aUsers[i], XN_SKEL_RIGHT_ELBOW, XN_SKEL_RIGHT_COLLAR);
    DrawLimb(aUsers[i], XN_SKEL_RIGHT_SHOULDER, XN_SKEL_RIGHT_HIP);
    DrawLimb(aUsers[i], XN_SKEL_RIGHT_HIP, XN_SKEL_RIGHT_KNEE);
    DrawLimb(aUsers[i], XN_SKEL_RIGHT_KNEE, XN_SKEL_RIGHT_FOOT);
    DrawLimb(aUsers[i], XN_SKEL_LEFT_SHOULDER, XN_SKEL_RIGHT_HIP);
    DrawLimb(aUsers[i], XN_SKEL_RIGHT_HIP, XN_SKEL_RIGHT_KNEE);
    DrawLimb(aUsers[i], XN_SKEL_RIGHT_KNEE, XN_SKEL_RIGHT_FOOT);
    DrawLimb(aUsers[i], XN_SKEL_LEFT_SHOULDER, XN_SKEL_LEFT_HIP);
    glEnd();
}
}