1. Year and Semester : 2013 SRING

2. Course Number : CS-492

3. Course Title : Kinect Programming

4. Work Number : LK-04

5. Work Name : Kinect Laboratory Report

6. Work Version : Version 1

7. Long Date : Saturday, 3, March, 2013

8. Author(s) Name(s) : Jake Waffle

Chapter 4 – Working With Skeleton Data

Introductory Examples:

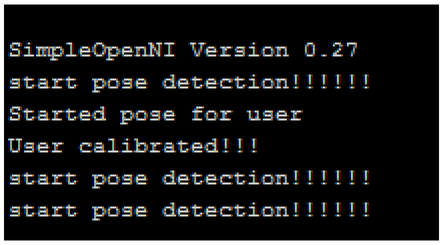
Code Snippet 1:

The first example taught me how to track a single joint using SimpleOpenNI's skeleton tracking (Borenstein pp191-192.) But I couldn't get the code to track my hand for a while. That was because I was doing the calibration pose incorrectly. I figured it out though. (The pose that is being used in this example is the "Psi" pose, but it looks like different poses can be used for calibration.)

In order to do skeleton tracking you must follow these steps. (This will just cover new information found in Chapter 4 and following along with the code is recommended.):

1. First SimpleOpenNI needs to have its enableUser() method called and that will enable skeleton tracking (different arguments can be passed to enableUser() depending on what you want tracked.) This only needs to happen once in the setup for the program.

2. Then each drawing iteration we must create a vector of integers that will store the active users visible to the Kinect. And we must pass that vector as an argument to SimpleOpenNI's getUsers() method (which adds the users into the vector.)
3. If there is a user visible to the Kinect, then we'll grab the first user's Id and see if the Kinect is tracking its skeleton. And if it is being tracked, that means that the user has been calibrated.
4. The calibration is done by SimpleOpenNI accessing some specific functions that are defined in our code (onNewUser(), onEndCalibration(), onStartPose().) I don't have to call those functions, probably because SimpleOpenNI has a better idea of when a new user has entered the space than I. Then the SimpleOpenNI.startPoseDetection() method within onNewUser() will indirectly trigger onStartPose() and SimpleOpenNI.requestCalibrationSkeleton() within onStartPose() indirectly triggers onEndCalibration() (Figure 1 shows the print statements that I used to understand the calibration process.)



```
SimpleOpenNI Version 0.27
start pose detection!!!!!!
Started pose for user
User calibrated!!!
start pose detection!!!!!!
start pose detection!!!!!!
```

Figure 1. – The console after tracking my left hand with the code from the book (Borenstein pp191-192.)

5. Then once we know the user is being tracked, we can query the real world vector location for the “left” hand (from the point of view of the Kinect, my right) with SimpleOpenNI.getJointPositionSkeleton(). But SimpleOpenNI.convertRealWorldToProjective() is needed in order to get the vector to be accurate with respect to the 2d depth image.
6. And the result is that we can track the user's right hand with respect to the 2d depth image. The example in the book simply draws an ellipse over the right hand.

Skeleton Anatomy Lesson:

Code Snippet #2:

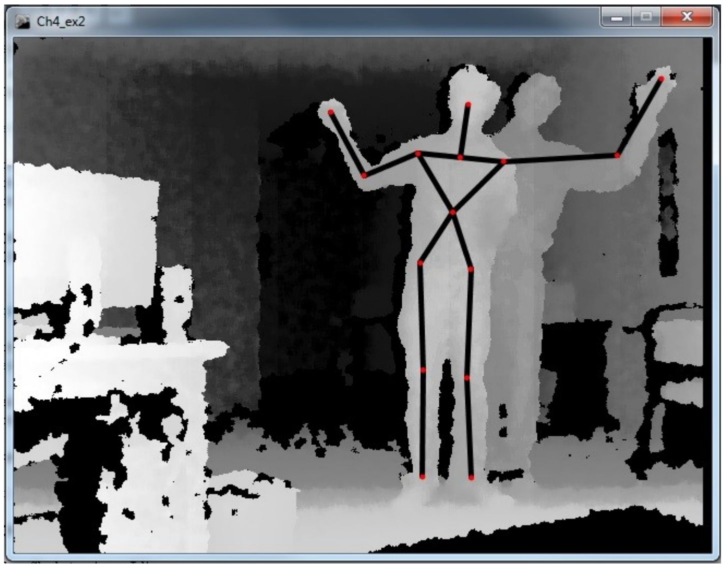
This code taught me a couple different things about the skeleton tracking. But it was very similar to the first one in structure and function. You can find the code in Borenstein's book on page 223-224.

1. First is that there are built-in methods for the SimpleOpenNI library that draws the line between two joints with respect to the 2d depth image. That method's parameters are simple a user's id and two of SimpleOpenNI's enumeration values associated with the joints that you want the line drawn between.
2. The other thing is that you can get a confidence percentage (a float between 0 and 1) associated with a joint when you are querying for its location (that could prevent some inaccurate readings from messing with your software.)

Experiment:

When experimenting with this project, Daniel and I found that the Kinect can be calibrated so that it thinks that two people are part of the same body (check out Figure 1. for a visual.) But to do this, we had to get close together and make it look like (to the Kinect) that Daniel and I were apart of the same body during calibration. It took a while to calibrate, but we eventually succeeded and the Kinect worked pretty fine thinking that we were one person.

Figure 2. - Daniel and I are tricking the kinect into thinking we're one person.



Measuring the Distance Between Two Joints

Code Snippet #3:

This code uses the same basic structure as in the first two code snippets. But there were a few things that I learned from it about PVectors and drawing text onto the screen (Borenstein pp 223-224).

1. The first is that the dataType “PVector” has some built-in methods that make vector math a lot easier. For example, getting the difference vector between two vectors is as easy as calling PVector.sub(firstVector, secondVector). There is also a method that will return a floating point number that represents the magnitude of a vector (which is the length of the hypotenuse of a triangle visually) and that method is just called mag(). Finally, the other method is called normalize() and that essentially will divide each xyz component by the magnitude of the vector (so that each xyz component will be at most of length 1.)
2. The second thing I learned was about drawing text onto the screen with the text() function. It's a pretty simple and easy to use, but seems like it can be pretty useful. The function essentially takes in three arguments: the first is the string that will be drawn onto the window, the second and third is the x- and y-position the string will be drawn on. I also learned in a later code snippet that you can change the font/size of the text within the setup() function by creating a font and setting that font as the general font that is used (which can be done like so, “textFont(createFont(“Verdana”, 40))”.)

Transferring Orientation in 3D

Code Snippet #4:

This example code taught be how to query the orientation of a joint instead of its position (Borenstein pp 230-232). And this is done in a similar way to what is being done in previous examples.

Overview:

This example essentially gets the orientation of the user's torso (i.e. the rotation of the torso is what is important, not the position) and saves that into a 3-dimensional matrix. Then OpenGL's current matrix is updated by that matrix. And when the axis' lines are drawn, they are going to be drawn with respect to the torso's orientation (because the current matrix isn't an identity matrix anymore.)

Procedure:

1. Getting the orientation of a joint is done in pretty much the same way as the position of a joint (so while the user is being tracked.) Before querying the orientation, we must initialize a matrix that will store the orientation.

   ```
   >PMatrix3d orientation = new PMatrix3D();
   ```

   Then we can query the orientation of a user's joint and also get the confidence of the orientation. Note that a pointer to the orientation is passed to the method.

   ```
   >float confidence = kinect.getJointOrientationSkeleton(userId,
                                           SimpleOpenNI.SKEL_TORO,
                                           orientation);
   ```

2. Now since we have a matrix that holds the orientation, we can apply that to OpenGL's current drawing matrix (the matrix on the top of OpenGL's stack.)

   >applyMatrix(orientation);

3. Then anything that is drawn using the OpenGL matrix that we just altered will be rotated so that it matches the torso of the user.

Code Snippet #5:

This bit of code essentially does the same things as the previous one. But it is different in that it draws a Kinect model using the orientation of the user's right elbow. I did learn one thing from it however.

1. The one thing I learned from this is that SimpleOpenNI has a method that makes it so that querying for the position/orientation of the right elbow is in fact the user's right elbow (where it would've been the user's left elbow without it.) This method does this by reflecting the kinect's view about the y-axis.

   >kinect.setMirror(true);

Project #10: Exercise Measurement

Overview:

In this project, I learned how to record the exact position (with respect to the Kinect) of a joint over a period of time. And also how to compare the recorded positions with the user's current position. (Note that the project uses OpenGL to draw the point where the left hand is in a 3d space, but it appears to be 2d because 2d shapes are being used to represent the joint.)

Procedure:

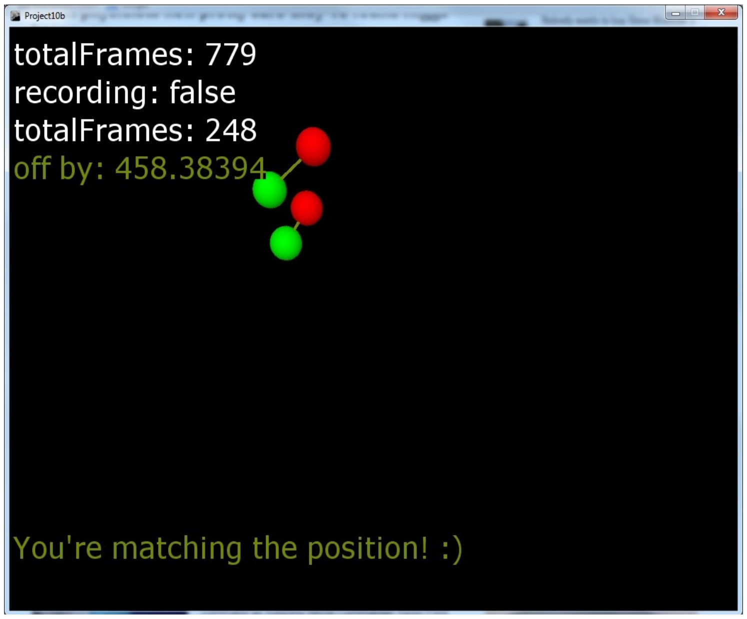
1. Copy the code from the book (Borenstein pp266-268, 275).
2. Analyze the code after running it.
3. The setup() function sets up OpenGL in the window, SimpleOpenNI, skeleton tracking and an instance of the SkeletonRecorder class (that will store the positions of the user's left hand over the recorded time period.)
4. Then after everything important is setup, the program is on stand-by until the user is calibrated, which triggers the recording to begin(by setting the “recording” variable to true.) (The calibrated user is also set to be recorded with SkeletonRecorder's setUser() method.) And that means that the SkeletonRecorder class will have its recordFrame() method called each draw() iteration for the calibrated user (Borenstein p275). Then within the recordFrame() method, SkeletonRecorder will add the position of the user's left-hand to the end of an ArrayList called “frames.”
5. The recording of the calibrated user's left-hand will go on until a key is pressed, which then triggers the keyPressed() function. And that will turn off the recording (by setting the “recording” variable to false.)

6. Once the recording process has stopped, the program then will compare the user's current hand position with the position recorded by SkeletonRecorder. First the difference vector of the recorded and current hand position is calculated (the sub() method is used to find the difference vector.) Then with the magnitude of the difference vector, we now know the distance between the two hand positions (the mag() method is used to find the magnitude().)
7. Throughout this entire program, there is always text on the screen that displays information about various parts of the program. One of the important bits of information is the distance between the recorded and current hand positions (which is used to determine the error of your exercise.) The rest of the text just gives information about the recording: is the program recording, how long is the recording, the frame of the recording the program is currently on.

Modification:

My modification essentially just added in another joint into the recording and comparison process. And I did this by altering the SkeletonRecorder class so that it would keep track of two joints instead of just one. I also added in some text to tell the user if s/he is close enough to the recording's position to be considered correct (Figure 1. shows the text at the bottom of the window.)

Figure 3. - This is a picture of the modification to project 10 (Borenstein pp266-268, 275).



```
totalFrames: 779
recording: false
totalFrames: 248
off by: 458.38394

You're matching the position! :)
```

Code Explanation:

(Note that the highlighted code is what is changed and the comments with "///" will explain what has changed in the code.)

Main:

```
void setup()
{
  size(1028, 768, OPENGL);
  kinect = new SimpleOpenNI(this);
  kinect.enableDepth();
  kinect.enableUser(SimpleOpenNI.SKEL_PROFILE_ALL);
  kinect.setMirror(true);
  ///This initializes and sets up the Skeleton Recorder to work with the left hand and elbow.

  recorder = new SkeletonRecorder(kinect, SimpleOpenNI.SKEL_LEFT_HAND,

                                  SimpleOpenNI.SKEL_LEFT_ELBOW);

  PFont font = createFont("Verdana", 40);

  textFont(font);

}


void draw()

{

  background(0);

  kinect.update();

  lights();

  noStroke();

  fill(255);


  text("totalFrames: " + recorder.frames.size()/recorder.numJoints, 5, 50);

  text("recording: " + recording, 5, 100);

  text("currentFrames: " + recorder.currentFrame, 5, 150);


  float c = map(offByDistance, 0, 1000, 0, 255);

  fill(c, 255-c, 0);

  text("off by: " + offByDistance, 5, 200);
```

```
 ///This whole block of code didn't exist before and it is for giving feedback to the user on how
 ///       close s/he is to the recorded position.
 //offByDistance should only be 0.0 while the user is recording/calibrating.
 if(offByDistance == 0.0){
   text("Record your exercise!", 5, 700);
 }
 //This sees if the user is within 500 units of the recorded position.
 else if (offByDistance < 500.0){
   text("You're matching the position! :)", 5, 700);
 }
 //This sees if the user is more than 500 units away from the recorded position.
 else{
   text("You're not matching the position! :(", 5, 700);
 }
 translate(width/2, height/2, 0);
 rotateX(radians(180));

 IntVector userList = new IntVector();
 kinect.getUsers(userList);

 if (userList.size() > 0){
   int userId = userList.get(0);
   recorder.setUser(userId);
```

```
  if (kinect.isTrackingSkeleton(userId)){
   ///This following highlighted block simply draws spheres where the left hand and elbow are
   ///   for the user currently (the elbow was added in basically.)
   PVector currentPositionA = new PVector();
   kinect.getJointPositionSkeleton(userId,
                                   SimpleOpenNI.SKEL_LEFT_HAND,
                                   currentPositionA);
   //This sphrere will be drawn at the current hand position.
   pushMatrix();
     fill(255,0,0);
     translate(currentPositionA.x, currentPositionA.y, currentPositionA.z);
     sphere(80);
   popMatrix();
   PVector currentPositionB = new PVector();
   kinect.getJointPositionSkeleton(userId,
                                   SimpleOpenNI.SKEL_LEFT_ELBOW,
                                   currentPositionB);
   //This sphere represents the left elbow
   pushMatrix();
     fill(255,0,0);
     translate(currentPositionB.x, currentPositionB.y, currentPositionB.z);
     sphere(80);
   popMatrix();
```

```
//If we are recording we, then we'll catch the current frame.

if (recording){

  recorder.recordFrame();

}

else{

  //Otherwise, we'll play the recording and compare with the current hand position.

  ///The variable name for the recorded hand position was changed and a parameter

  ///             for querying the left hand's position is being used.

  PVector recordedPositionA = recorder.getPosition(0);

  //This represents the recorded hand position

  pushMatrix();

    fill(0,255,0);

    translate(recordedPositionA.x,

              recordedPositionA.y,

              recordedPositionA.z);

    sphere(80);

  popMatrix();
```

```
///This entire following block of highlighted code was added. And it works exactly like the
///		previous block (draws the recorded position for the elbow,) except it uses the
///		elbow's position instead (by using a “1” for getPosition()'s argument.)
PVector recordedPositionB = recorder.getPosition(1);
//This represents the recorded elbow position
pushMatrix();
  fill(0,255,0);
  translate(recordedPositionB.x,
            recordedPositionB.y,
            recordedPositionB.z);
  sphere(80);
popMatrix();


//This draws a line between the recorded hand position and the current hand position.
stroke(c,255-c,0);
strokeWeight(20);
line(currentPositionA.x, currentPositionA.y,
     currentPositionA.z, recordedPositionA.x,
     recordedPositionA.y, recordedPositionA.z);
```

```
        ///This draws an additional line between the recorded elbow position and the current elbow
        ///          position.
        line(currentPositionB.x, currentPositionB.y,
           currentPositionB.z, recordedPositionB.x,
           recordedPositionB.y, recordedPositionB.z);

        ///Calculates the distance between the recorded and observed positions (for the hand and
        ///          elbow.)
        currentPositionA.sub(recordedPositionA);
        currentPositionB.sub(recordedPositionB);

        ///This adds the magnitude of the two difference vectors previously calculated to get the
        ///          cumulative distance away from the recorded position.
        offByDistance = currentPositionA.mag() + currentPositionB.mag();
        recorder.nextFrame();
      }
    }
  }
}
```

SkeletonRecorder:

```
class SkeletonRecorder{

  SimpleOpenNI context;

  ///These two variables hold the ids of the TWO joints of interest.

  int jointIDA, jointIDB;

  ///This variable is just there so that I could later add more joints into this class (wasn't required.)

  int numJoints;

  int userID;

  //Each element does not correspond with the frame number!

  //Every numJoints elements will represent a single frame.

  ArrayList<PVector> frames;

  int currentFrame = 0;

  ///Another parameter was added for the second joint (the elbow.)

  SkeletonRecorder(SimpleOpenNI tempContext, int tempJointIDA, int tempJointIDB){

    context = tempContext;

    ///Both joint Ids get stored in the class variables.

    jointIDA = tempJointIDA;

    jointIDB = tempJointIDB;

    ///The number of joints is initialized to two.

    numJoints = 2;

    frames = new ArrayList();

  }
```

```
void setUser(int tempUserID){
  userID = tempUserID;
}


void recordFrame(){
  ///Everything was duplicated for the second joint basically.
  PVector positionA = new PVector();
  PVector positionB = new PVector();
  context.getJointPositionSkeleton(userID, jointIDA, positionA);
  context.getJointPositionSkeleton(userID, jointIDB, positionB);
  frames.add(positionA);
  frames.add(positionB);
}
///A parameter to differentiate between the two joints is used
PVector getPosition(int jointNum){
  ///An equation is used to determine the frame element, because two elements of frames makes
  ///    up a single frame.
  return frames.get(currentFrame*numJoints + jointNum);
}


void nextFrame(){
  currentFrame++;
```

```
    ///The maximum size of the frames List has been doubled.
    if (currentFrame*numJoints == frames.size() ){
      currentFrame = 0;
    }
  }
}
```

Project #11: Dance Move Triggers MP3

Overview:

This project is similar to the last in that it compares a recorded pose to the user's current position. But the position is compared in a relative fashion rather than an exact comparison. The recorded pose is also hard-coded into the program instead of allowing the user to record the pose and match it. And when I said that the position is compared relatively, I meant that the program checks to see where the user's joints are in relation to the others. When the user strikes the correct pose, the program simply plays a song.

Procedure:

1. I copied the second bit of code (I'm going to skip the first code snippet, because it works in a very similar way to that of the advanced version) from the book (Borenstein pp288-290, 294-295).
2. Then ran and analyzed the code.
3. It starts off in the setup() function by initializing the window, SimpleOpenNI, Minim and the SkeletonPoser class.

   ```
   >kinect = new SimpleOpenNI(this);
   >minim = new Minim(this);
   >pose = new SkeletonPoser(kinect);
   ```

   Then the skeleton tracking and depth image are enabled within the SimpleOpenNI object.

   ```
   >kinect.enableUser(SimpleOpenNI.SKEL_PROFILE_ALL);
   >kinect.enableDepth();
   ```

   A music file is loaded into the Minim object. And finally the SkeletonPoser object gets rules added to it that define the recorded pose that I mentioned in the overview.

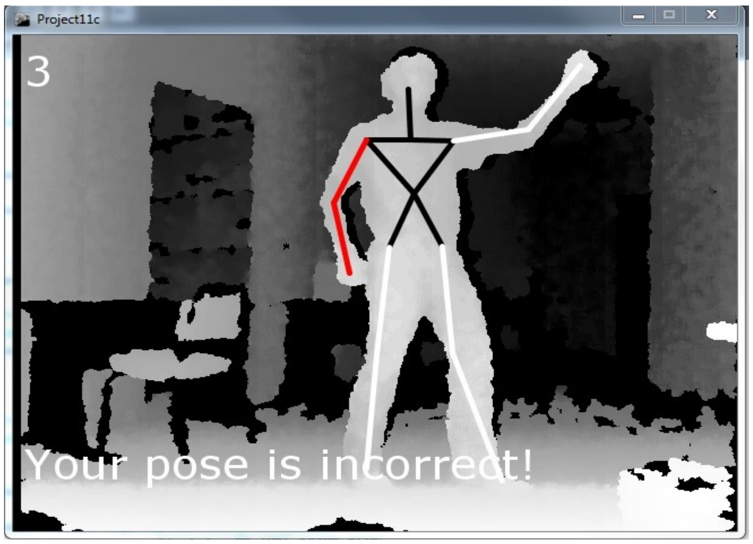
4. When a rule is added to the SkeletonPoser instance, it is defined as one joint in a position in relation to another joint (e.g. the right hand being to the right of the right elbow or the right hand being above the right elbow.) And these rules are saved within a PoseRule class instance within the SkeletonPoser instance.
5. From there the program doesn't really do anything until the user is calibrated for skeleton tracking.
6. Then once the user is calibrated the SkeletonPoser object's check() method is called. And that method will check to see if the specified user's skeleton follows the rules that were defined in the setup() function. If the check() method returns true, then the music that was loaded will play.
7. To check to see if the user's skeleton is following the defined rules, the SkeletonPoser class will utilize another class called PoseRule within its check() method. Since the SkeletonPoser stores each rule as a separate PoseRule instance, each rule just needs to be fetched from the SkeletonPoser's ArrayList of rules and then each needs its check() method to be called. Then after running through all of the rules, if all of the check() methods for the rules returned true, the SkeletonPoser's check() method will return true (meaning that the pose is correct.)
8. The PoseRule check() method essentially will check to see if the one of the specified joints is above/below/right of/left of the other. And all of the rules are separated into their own PoseRule instance within the SkeletonPoser instance.
9. Other than the poses being checked each iteration of the draw() function, the skeleton of the user is also drawn. But depending on if the user's pose is correct or not, the entire

skeleton will be one of two colors (white for the correct pose and red for an incorrect pose.)

Modification:

1. When doing modifications, I started out making it so that each limb of the skeleton is checked individually to see if it is correct (Figure 4. shows how each limb is independently being checked.) That was done by using an instance of the SkeletonPoser class for each limb. I did that because the previous program made it hard to tell which limb was incorrect.
2. Next I made it so that the color of each of those limbs is colored independently. That was done by checkinsg each limb as the skeleton is being drawn within the drawSkeleton() function.
3. And Finally I printed some information to the screen so that the user has more information about whether their pose is correct/incorrect. If you'll look at Figure 4., the three represents the number of correct joints and the message says whether the overall pose is correct or incorrect.

Figure 4. - This is a picture of the modified version of Project #11.



Your pose is incorrect!

Code Explanation:

The following code is where I've made changes to Project #11 and more specifically the highlighted regions. Comments with “///” before the highlighted regions will be for explaining the code block below it. And the SkeletonPoser class was unaltered in this modification.

```
///Here I declared four SkeletonPoser instances instead of one, one for each limb.
SkeletonPoser pose1, pose2, pose3, pose4;


void setup(){
  size(640,480);
  kinect = new SimpleOpenNI(this);
  kinect.enableDepth();
  kinect.enableUser(SimpleOpenNI.SKEL_PROFILE_ALL);
  kinect.setMirror(true);

  minim = new Minim(this);

  //player = minim.loadFile("aFile.mp3");
```

```
 ///The declarations all need to be initialized.

 pose1 = new SkeletonPoser(kinect);

 pose2 = new SkeletonPoser(kinect);

 pose3 = new SkeletonPoser(kinect);

 pose4 = new SkeletonPoser(kinect);


 /// Each limb's rules are added into separate SkeletonPoser instances

 //rules for the right arm

 pose1.addRule(SimpleOpenNI.SKEL_RIGHT_HAND,

              PoseRule.ABOVE,

              SimpleOpenNI.SKEL_RIGHT_ELBOW);

 pose1.addRule(SimpleOpenNI.SKEL_RIGHT_HAND,

              PoseRule.RIGHT_OF,

              SimpleOpenNI.SKEL_RIGHT_ELBOW);

 pose1.addRule(SimpleOpenNI.SKEL_RIGHT_ELBOW,

              PoseRule.ABOVE,

              SimpleOpenNI.SKEL_RIGHT_SHOULDER);

 pose1.addRule(SimpleOpenNI.SKEL_RIGHT_ELBOW,

              PoseRule.RIGHT_OF,

              SimpleOpenNI.SKEL_RIGHT_SHOULDER);
 //rules for the left arm
 pose2.addRule(SimpleOpenNI.SKEL_LEFT_ELBOW,
             PoseRule.BELOW,
             SimpleOpenNI.SKEL_LEFT_SHOULDER);
 pose2.addRule(SimpleOpenNI.SKEL_LEFT_ELBOW,
             PoseRule.LEFT_OF,
             SimpleOpenNI.SKEL_LEFT_SHOULDER);
 pose2.addRule(SimpleOpenNI.SKEL_LEFT_HAND,
             PoseRule.LEFT_OF,
             SimpleOpenNI.SKEL_LEFT_ELBOW);
 pose2.addRule(SimpleOpenNI.SKEL_LEFT_HAND,
             PoseRule.BELOW,
             SimpleOpenNI.SKEL_LEFT_ELBOW);


 //rules for the right leg
 pose3.addRule(SimpleOpenNI.SKEL_RIGHT_KNEE,
             PoseRule.BELOW,
             SimpleOpenNI.SKEL_RIGHT_HIP);
 pose3.addRule(SimpleOpenNI.SKEL_RIGHT_KNEE,
             PoseRule.RIGHT_OF,
             SimpleOpenNI.SKEL_RIGHT_HIP);
```

```
  //rules for the left leg
  pose4.addRule(SimpleOpenNI.SKEL_LEFT_KNEE,
              PoseRule.BELOW,
              SimpleOpenNI.SKEL_LEFT_HIP);
  pose4.addRule(SimpleOpenNI.SKEL_LEFT_KNEE,
              PoseRule.LEFT_OF,
              SimpleOpenNI.SKEL_LEFT_HIP);
  pose4.addRule(SimpleOpenNI.SKEL_LEFT_FOOT,
              PoseRule.BELOW,
              SimpleOpenNI.SKEL_LEFT_KNEE);
  pose4.addRule(SimpleOpenNI.SKEL_LEFT_FOOT,
              PoseRule.LEFT_OF,
              SimpleOpenNI.SKEL_LEFT_KNEE);


  strokeWeight(5);


  PFont font = createFont("Verdana", 40);
  textFont(font);
}
```

```
void draw(){
  background(0);
  kinect.update();
  image(kinect.depthImage(), 0, 0);


  IntVector userList = new IntVector();
  kinect.getUsers(userList);
  if(userList.size() > 0){
    int userId = userList.get(0);


    if (kinect.isTrackingSkeleton(userId)){
      ///Everything in this area was moved into the drawSkeleton() function besides the call to
      ///   drawSkeleton(). That allows the color of the limbs to change as the skeleton is being
      ///   drawn.
      drawSkeleton(userId);
    }
  }
}
```

```
void drawSkeleton(int userId){
  ///All of the lines that don't matter will be black. Specifically that's the lines belonging to the
  ///       torso and head.
  stroke(0);

  ///This variable keeps track of how many limbs are correct and will be displayed on the
  ///       window.
  int correctLimbs = 0;
  //Here the unimportant lines are drawn.
  drawLimb(userId, SimpleOpenNI.SKEL_HEAD,
                        SimpleOpenNI.SKEL_NECK);
  drawLimb(userId, SimpleOpenNI.SKEL_NECK,
                        SimpleOpenNI.SKEL_LEFT_SHOULDER);
  drawLimb(userId, SimpleOpenNI.SKEL_NECK,
                        SimpleOpenNI.SKEL_RIGHT_SHOULDER);
  drawLimb(userId, SimpleOpenNI.SKEL_LEFT_SHOULDER,
                        SimpleOpenNI.SKEL_TORSO);
  drawLimb(userId, SimpleOpenNI.SKEL_RIGHT_SHOULDER,
                        SimpleOpenNI.SKEL_TORSO);
  drawLimb(userId, SimpleOpenNI.SKEL_TORSO,
                        SimpleOpenNI.SKEL_LEFT_HIP);
  drawLimb(userId, SimpleOpenNI.SKEL_TORSO,
                        SimpleOpenNI.SKEL_RIGHT_HIP);
```

```
///This is where the left arm is checked to see if it is correct and if it is correct, then we'll
///     increment the correctLimbs variable and change the color of the limbs to white.
if (pose2.check(userId)){
  //The left arm will be colored white
  stroke(255);
  correctLimbs++;
}
else{
  //The left arm will be colored red.
  stroke(255,0,0);
}


//And here is where we'll draw the left arm with the color that was just set.
drawLimb(userId, SimpleOpenNI.SKEL_LEFT_SHOULDER,
                 SimpleOpenNI.SKEL_LEFT_ELBOW);
drawLimb(userId, SimpleOpenNI.SKEL_LEFT_ELBOW,
                 SimpleOpenNI.SKEL_LEFT_HAND);
```

```
///This is where the right arm is checked to see if it is correct and if it is correct, then we'll
///      increment the correctLimbs variable and change the color of the limbs to white.
if (pose1.check(userId)){
  //The right arm will be colored white
  stroke(255);
  correctLimbs++;
}
else{
  //The right arm will be colored red.
  stroke(255,0,0);
}


///And here is where we'll draw the right arm with the color that was just set.
drawLimb(userId, SimpleOpenNI.SKEL_RIGHT_SHOULDER,
                        SimpleOpenNI.SKEL_RIGHT_ELBOW);
drawLimb(userId, SimpleOpenNI.SKEL_RIGHT_ELBOW,
                        SimpleOpenNI.SKEL_RIGHT_HAND);
```

```
///This is where the left leg is checked to see if it is correct and if it is correct, then we'll
///      increment the correctLimbs variable and change the color of the limbs to white.
if (pose4.check(userId)){
  //The left leg will be colored white
  stroke(255);
  correctLimbs++;
}
else{
  //The left leg will be colored red.
  stroke(255,0,0);
}


///And here is where we'll draw the left leg with the color that was just set.
drawLimb(userId, SimpleOpenNI.SKEL_LEFT_HIP,
                        SimpleOpenNI.SKEL_LEFT_KNEE);
drawLimb(userId, SimpleOpenNI.SKEL_LEFT_KNEE,
                        SimpleOpenNI.SKEL_LEFT_FOOT);
```

```
///This is where the right leg is checked to see if it is correct and if it is correct, then we'll
///      increment the correctLimbs variable and change the color of the limbs to white.
if (pose3.check(userId)){
  //The right leg will be colored white
  stroke(255);
  correctLimbs++;
}
else{
  //The right leg will be colored red.
  stroke(255,0,0);
}


///And here is where we'll draw the right leg with the color that was just set.
drawLimb(userId, SimpleOpenNI.SKEL_RIGHT_HIP,
                         SimpleOpenNI.SKEL_RIGHT_KNEE);
drawLimb(userId, SimpleOpenNI.SKEL_RIGHT_KNEE,
                         SimpleOpenNI.SKEL_RIGHT_FOOT);
  ///Here we'll change the color that stuff is to be drawn with, then display the amount of correct
  ///      limbs in the top-left part of the screen.
  stroke(0,200,0);
  text(correctLimbs, 10, 50);

  ///Finally we'll check to see if all of the limbs were correct. And depending on how many limbs
  ///      were correct, we'll display a message that tells the user if the pose is correct or incorrect
  ///      overall. If I was using music to tell the user that, then I would trigger the music to be
  ///      played here.
  if (correctLimbs == 4){
    text("Your pose is correct!", 10, 430);
  }
  else{
    text("Your pose is incorrect!", 10, 430);
  }
}
```