Kinect Chapter 13. FAAST-style Body Gestures

[**Note**: all the code for this chapter is available online at http://fivedots.coe.psu.ac.th/~ad/kinect/; only important fragments are described here.]

The ArniesTracker application from Chapter 4 illustrates how to program with OpenNI's skeletal tracking, rendering each skeleton as connected colored lines (with an attached Arnies head). Figure 1 is a reminder of how things look.

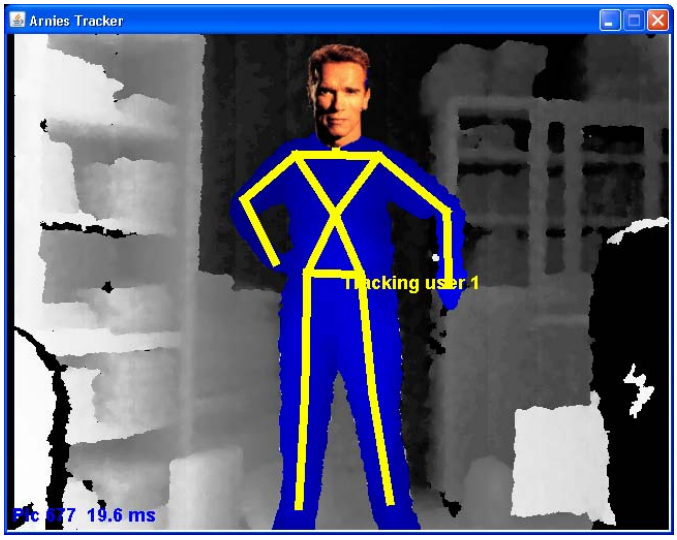

Figure 1. An Arnie Being Tracked.

The crucial data structure is a HashMap of skeletons called userSkels, which maps user IDs to skeletons made up of joints. The collection of joints for a given skeleton are represented by another HashMap which pairs joint names (SkeletonJoint values) with their (x, y, z) coordinates (SkeletonJointPosition objects). The userSkels declaration:

```
private HashMap<Integer,
          HashMap<SkeletonJoint, SkeletonJointPosition>> userSkels;
```

When a new user is detected, a ID-skeleton mapping is added to userSkels, which is updated as the user moves, and deleted when the person leaves the scene.

This data structure is utilized in the ArniesTracker application of Chapter 4 to draw the skeletons. This chapter explores its use for detecting body gestures, such as lifting an arm, bringing both hands together, and waving a hand up and down.

To give credit where it's due, this chapter was inspired by the FAAST library (http://projects.ict.usc.edu/mxr/faast/). FAAST, short for Flexible Action and Articulated Skeleton Toolkit, allows a programmer to add full-body control to games and VR applications. It's built on top of OpenNI and NITE on a Windows platform,

and makes it very easy to map gestures to keyboard and mouse controls, thereby allowing conventional software to be controlled by gesturing. FAAST also includes network support so skeletal information can be transported between machines. The application in this chapter is much less capable, only calling a print method when a body gesture is detected.

1. Extending ArniesTracker

Graphically, the new version of ArniesTracker is unchanged from Figure 1, but now prints gesture detection details to standard output. For example, Figure 2 shows the user turning their body to the left.

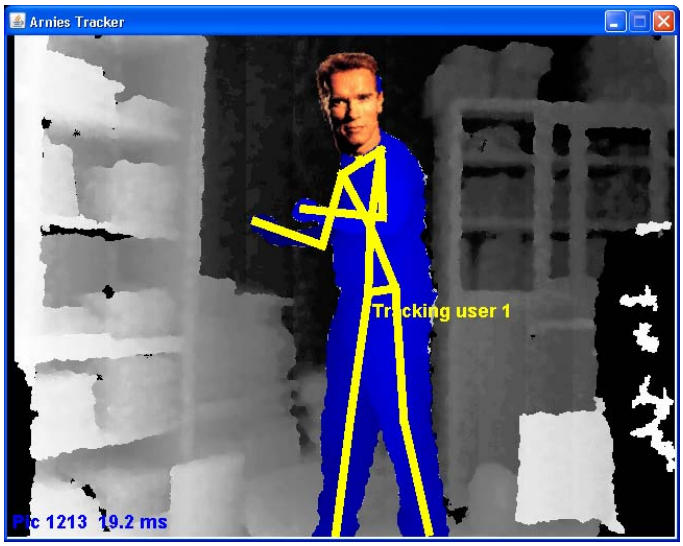

Figure 2. Turning to the Left.

The resulting print-out includes the following lines:

```
        :
TURN_LEFT 1 on
LEAN_LEFT 1 on
                        TURN_LEFT 1 off
TURN_LEFT 1 on
                        LEAN_LEFT 1 off
                        TURN_LEFT 1 off
        :
```

This text is reporting two gestures – turning left and leaning left. An "on" message is printed when a gesture starts, and the "off" messages report when the gestures ends.

The application detects *two* kinds of gesture: basic ones which correspond to the user placing their body in a certain position (e.g. a raised hand, bending to the left), and gesture *sequences* which are more complex moves made up of a series of basic gestures. An example is horizontal waving which consists of a user moving their hand to the left and right several times.

Figure 3 shows the user with their right arm lowered, which occurs at the end of a series of up and down movements.

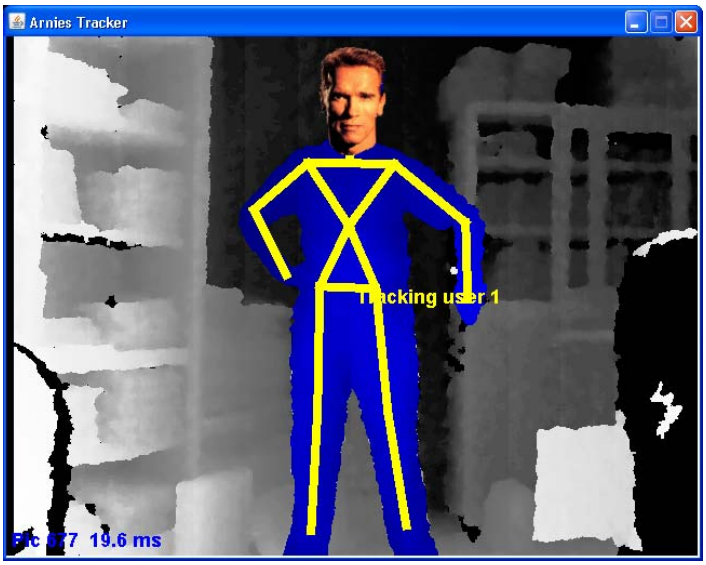

Figure 3. Vertical Waving.

The output is a mixture of basic gestures, and a vertical waving message which is reported when a sequence of raised and lowered arm gestures are observed.

```
          :
RH_DOWN 1 on
                          RH_DOWN 1 off
RH_FWD 1 on
RH_UP 1 on
                          RH_UP 1 off
                          RH_FWD 1 off
RH_DOWN 1 on
RH_FWD 1 on
                          RH_DOWN 1 off
RH_UP 1 on
                          RH_UP 1 off
                          RH_FWD 1 off
RH_DOWN 1 on
VERT_WAVE 1 on
RH_FWD 1 on
                          RH_DOWN 1 off
          :
```

The RH prefix on most of the messages stands for "Right Hand". The vertical wave gesture is represented by the VERT_WAVE output which appears after several up and down swipes of the right hand. Note that the up and down gestures are interspersed with RH_FWD messages which signal that the right hand is forward of the torso. This position often occurs as the user moves their hand up and down, but the waving gesture is still reported.

2. An Overview of the New ArniesTracker

The ArniesTracker class diagrams in Figure 4 include thick green borders around the new classes.

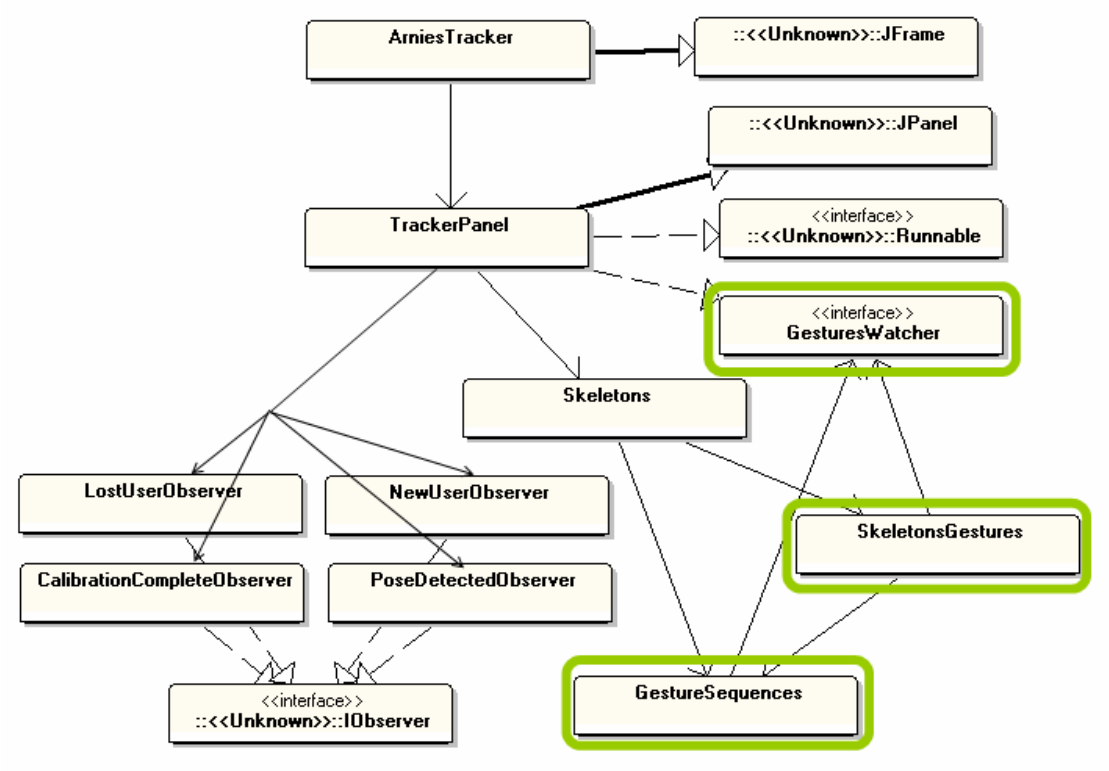

Figure 4. Class Diagrams for the New ArniesTracker.

The new classes are SkeletonsGestures and GestureSequences, and an interface called GesturesWatcher. SkeletonsGestures examines the skeletons data (i.e. the userSkels HashMap) and reports basic gestures. These gestures are also stored in a list processed by GestureSequences which looks for subsequences corresponding to complex gestures (e.g. vertical waving).

Both basic and complex gestures are reported by calling the GesturesWatcher.pose() method, which is implemented in TrackerPanel.

The Skeletons class is slightly modified from the version in the earlier ArniesTracker, and there are minor changes to the CalibrationCompleteObserver and LostUserObserver callbacks, as explained in the following subsections.

2.1. Creating the Gesture Detectors

The Skeletons constructor creates the userSkels data structure for the user skeletons, and also initializes the SkeletonsGestures and GestureSequences detectors.

```
// globals
private HashMap<Integer, HashMap<SkeletonJoint,
                                 SkeletonJointPosition>> userSkels;
/* userSkels maps user IDs --> a joints map (i.e. a skeleton)
   skeleton maps joints --> positions  */
```



```
// gesture detectors
private GestureSequences gestSeqs;
private SkeletonsGestures skelsGests;


public Skeletons(UserGenerator userGen, DepthGenerator depthGen,
                                      GesturesWatcher watcher)
{ this.userGen = userGen;
  this.depthGen = depthGen;

  headImage = loadImage(HEAD_FNM);
  configure();
  userSkels = new HashMap<Integer,
        HashMap<SkeletonJoint, SkeletonJointPosition>>();

  /* create the two gesture detectors, and tell them
     who to notify */
  gestSeqs = new GestureSequences(watcher);
  skelsGests = new SkeletonsGestures(watcher, userSkels, gestSeqs);
} // end of Skeletons()
```

The constructor's GesturesWatcher argument refers to the TracksPanel object, which implements the GesturesWatcher interface. The detectors are passed copies of this reference so they can notify TracksPanel by calling its GesturesWatcher.pose() method.

The SkeletonsGestures object is instantiated with a reference to the userSkels data structure, which is examined for body gestures. It is also assigned a reference to the GestureSequences object so detected basic gestures can be passed to it.

2.2. Updating the Skeleton (and Detectors)

A call to Skeletons.update() updates the joint positions for each user's skeleton. It's also necessary to pass the skeleton's userID to the two detectors, so that they can perform gesture analysis on that user's skeleton.

```
public void update()
{
  try {
    int[] userIDs = userGen.getUsers();
              // there may be many users in the scene
    for (int i = 0; i < userIDs.length; ++i) {
      int userID = userIDs[i];
      if (skelCap.isSkeletonCalibrating(userID))
        continue;
      if (skelCap.isSkeletonTracking(userID)) {
        updateJoints(userID);

        /* when a skeleton changes, have the
           detectors look for gestures  */
        gestSeqs.checkSeqs(userID);
        skelsGests.checkGests(userID);
      }
    }
  }
  catch (StatusException e)
  {  System.out.println(e); }
```



```
}  // end of update()
```

2.3. The Observers

The Skeletons class sets up four 'observers' (listeners) so when a new user is detected in the scene his skeleton can be calibrated in a standard pose, and then tracked. Of these observers, CalibrationCompleteObserver and LostUserObserver are modified to add and remove users to/from the detectors.

```
class CalibrationCompleteObserver implements
                   IObserver<CalibrationProgressEventArgs>
{
  public void update(
        IObservable<CalibrationProgressEventArgs> observable,
        CalibrationProgressEventArgs args)
  {
    int userID = args.getUser();
    System.out.println("Calibration status: " + args.getStatus() +
                                        " for user " + userID);
    try {
      if (args.getStatus() == CalibrationProgressStatus.OK) {
        // calibration succeeded; move to skeleton tracking
        System.out.println("Starting tracking user " + userID);
        skelCap.startTracking(userID);

        // add user to the gesture detectors
        userSkels.put(new Integer(userID),
            new HashMap<SkeletonJoint, SkeletonJointPosition>());
            // create new skeleton map for the user
        gestSeqs.addUser(userID);
      }
      else    // calibration failed; return to pose detection
        poseDetectionCap.StartPoseDetection(calibPoseName, userID);
    }
    catch (StatusException e)
    {  e.printStackTrace(); }
  }
}  // end of CalibrationCompleteObserver inner class



class LostUserObserver implements IObserver<UserEventArgs>
{
  public void update(IObservable<UserEventArgs> observable,
                     UserEventArgs args)
  { int userID = args.getId();
    System.out.println("Lost track of user " + userID);

    // remove user from the gesture detectors
    userSkels.remove(userID);
    gestSeqs.removeUser(userID);
  }
} // end of LostUserObserver inner class
```

The code shows that only the GestureSequences object is explicitly contacted. There's no need to call SkeletonsGesture since it employs the userSkels object for all its

analyses. As a consequence, only that data structure needs to be modified in order to affect the SkeletonsGestures object.

4. A Skeletal Reminder

Before I explain how basic gestures are detected by SkeletonsGestures, Figure 5 offers a quick reminder of the OpenNI skeleton joints.

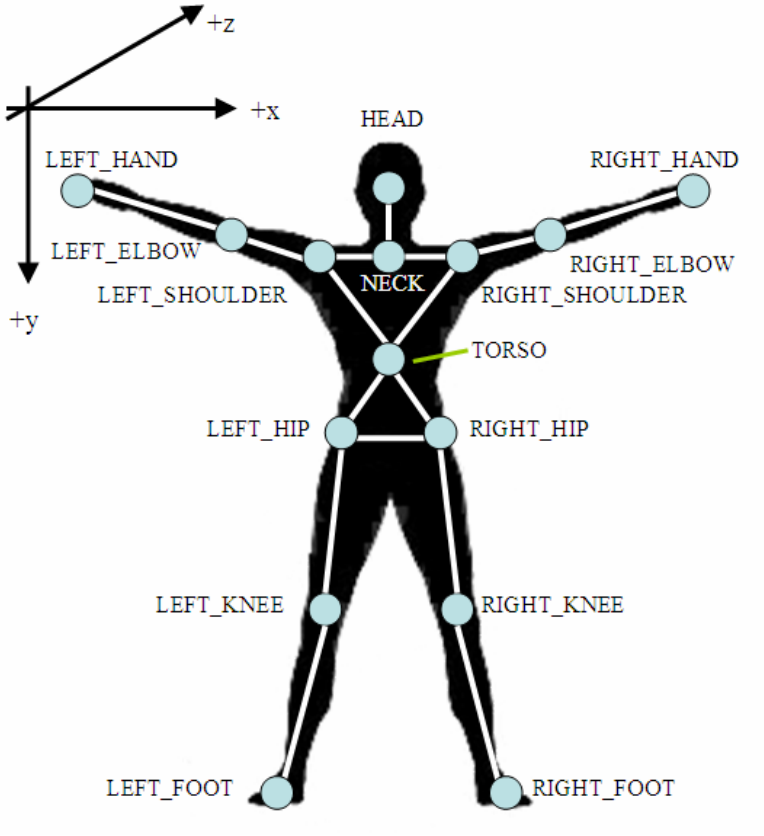

Figure 5. A Skeleton and its Joints.

OpenNI currently reports the positions for 15 joints, although the framework has labels for another nine (such as the waist and wrists).

Since ArniesTracker draws skeletons on the screen, joint positions are stored in screen coordinates, not as real-world values. As Figure 5 indicates, this means that the positive y-axis runs down the screen, and the positive z-axis is directed into the scene (i.e. away from the Kinect's camera).

The transformation from real-world to screen coordinates is carried out inside Skeletons.updateJoint() using DepthGenerator.convertRealWorldToProjective(); the crucial few lines are:

```
SkeletonJointPosition jPos = null;
       :
jPos = new SkeletonJointPosition(
     depthGen.convertRealWorldToProjective( pos.getPosition() ),
```



```
     pos.getConfidence());
       :
skel.put(joint, jPos);
```

skel is the HashMap of joints for a particular user.

Using screen coordinates for the joints has an unfortunate effect on the gesture calculations, which I'll explain shortly. It appears that the FAAST system uses real-world coordinates for its calculations, although I haven't looked at their code to check.

5. Detecting Basic Gestures

The SkeletonsGestures class carries out basic gesture detection by examining pairs of skeleton joints looking for different kinds of poses, such as right hand raised or hands close together. Each gesture is reported twice, when it first starts and when it ends. An alternative strategy would be to generate a gesture-occurring event during every update of the skeleton, until the gesture stops.

A gesture event is reported by calling the GesturesWatcher.pose() method in the designated watcher. The call includes the user ID of the skeleton, a GestureName value, and a boolean denoting if the gesture has just started or finished.

GestureName is the enum:

```
enum GestureName {
  HORIZ_WAVE, VERT_WAVE,                              // waving
  HANDS_NEAR,                                         // two hands
  LEAN_LEFT, LEAN_RIGHT, LEAN_FWD, LEAN_BACK,         // leaning
  TURN_RIGHT, TURN_LEFT,                              // turning
  LH_LHIP, RH_RHIP,                                   // touching
  RH_UP, RH_FWD, RH_OUT, RH_IN, RH_DOWN,           // righ hand position
  LH_UP                                            // left hand position
}
```

The names are divided into seven informal groups (denoted by comments in the code), and many more names could be added. For example, all the right hand positions could be duplicated for the left hand in addition to the current up position (LH_UP). A good source of ideas is the FAAST documentation at http://projects.ict.usc.edu/mxr/faast/. For instance, FAAST also recognizes a variety of left and right foot gestures.

The SkeletonsGestures constructor is passed a reference to the watcher object (TrackerPanel in this application), the user skeletons HashMap, and a GestureSequences object.

```
// globals
private GesturesWatcher watcher;
private HashMap<Integer,
           HashMap<SkeletonJoint, SkeletonJointPosition>> userSkels;
private GestureSequences gestSeqs;


public SkeletonsGestures(GesturesWatcher aw,
```



```
            HashMap<Integer,
              HashMap<SkeletonJoint, SkeletonJointPosition>> uSkels,
            GestureSequences gSeqs)
{ watcher = aw;
  userSkels = uSkels;
  gestSeqs = gSeqs;
}  // end of SkeletonsGestures()
```

The GestureSequences object will store basic gesture sequences for each user, and look for complex gestures made of sub-sequences. SkeletonsGestures sends gesture detection information to GestureSequences at run time.

5.1. Checking For Gestures

The only public method in SkeletonsGestures (aside from the constructor) is checkGests(), which is called whenever a skeleton is updated. It checks the updated skeleton joints to decide which gestures have just started or finished, and notifies the watcher.

```
public void checkGests(int userID)
{
  HashMap<SkeletonJoint, SkeletonJointPosition> skel =
                userSkels.get(userID);
  if (skel == null)
    return;

  calcSkelLengths(skel);

  // uncomment the detectors for the gestures you want...

  // twoHandsNear(userID, skel);

  leanLeft(userID, skel);
  leanRight(userID, skel);
  leanFwd(userID, skel);
  leanBack(userID, skel);

  turnLeft(userID, skel);
  turnRight(userID, skel);
/*
  leftHandTouchHip(userID, skel);
  rightHandTouchHip(userID, skel);

  rightHandUp(userID, skel);
  rightHandFwd(userID, skel);
  rightHandOut(userID, skel);
  rightHandIn(userID, skel);
  rightHandDown(userID, skel);

  leftHandUp(userID, skel);
*/
}  // end of checkGests()
```

checkGests() is passed the user ID of the skeleton that was just updated in the Skeletons object, and uses it to access the user's HashMap of joints. Useful skeleton

lengths are calculated in calcSkelLengths(), and then a lengthy series of methods are called, each one checking for a different gesture.

If I uncomment all the gesture detecting methods, the output from ArniesTracker can become overwhelming. Therefore, most of the methods are commented out in the code apart from those looking for leaning and turning gestures.

5.2. Calculating Skeleton Lengths

calcSkelLengths() calculate lengths between three joint pairs in the skeleton. These lengths are used later by some of the detectors to judge the distances between joints.

Unfortunately, these length calculations have to be done repeatedly since the on-screen dimensions of a skeleton will change as the user moves closer to or further away from the Kinect. This overhead would disappear if the joints information was stored in real-world coordinates, since the real-world lengths between joint pairs don't vary.

```
// globals
// standard skeleton lengths
private static final float NECK_LEN = 50.0f;
private static final float LOWER_ARM_LEN = 150.0f;
private static final float ARM_LEN = 400.0f;

private float neckLength = NECK_LEN;        // neck to shoulder len
private float lowerArmLength = LOWER_ARM_LEN;  // hand to elbow len
private float armLength = ARM_LEN;          // hand to shoulder len


private void calcSkelLengths(HashMap<SkeletonJoint,
                             SkeletonJointPosition> skel)
{ Point3D neckPt = getJointPos(skel, SkeletonJoint.NECK);
  Point3D shoulderPt = getJointPos(skel,
                                   SkeletonJoint.RIGHT_SHOULDER);
  Point3D handPt = getJointPos(skel, SkeletonJoint.RIGHT_HAND);
  Point3D elbowPt = getJointPos(skel, SkeletonJoint.RIGHT_ELBOW);

  if ((neckPt != null) && (shoulderPt != null) &&
      (handPt != null) && (elbowPt != null)) {
    neckLength = distApart(neckPt, shoulderPt);
    armLength = distApart(handPt, shoulderPt);
    lowerArmLength = distApart(handPt, elbowPt);
  }
}  // end of calcSkelLengths()


private float distApart(Point3D p1, Point3D p2)
// the Euclidian distance between two points
{
  float dist = (float) Math.sqrt(
           (p1.getX() - p2.getX())*(p1.getX() - p2.getX()) +
           (p1.getY() - p2.getY())*(p1.getY() - p2.getY()) +
           (p1.getZ() - p2.getZ())*(p1.getZ() - p2.getZ()) );
  return dist;
}  // end of distApart()
```


Lengths are calculated between the neck and shoulder, hand and elbow, and hand and shoulder, and stored in the globals neckLength, lowerArmLength, and armLength. These measurements were chosen since they offer a range of human-specific lengths ranging from a short amount (neck to shoulder) to long (hand to shoulder).

getJointPos() is called by calcSkelLengths() to return the current screen coordinates of a joint as a Point3D object, or null if the joint hasn't a value.

```
private Point3D getJointPos(
          HashMap<SkeletonJoint, SkeletonJointPosition> skel,
                                                  SkeletonJoint j)
{ SkeletonJointPosition pos = skel.get(j);
  if (pos == null)
    return null;

  if (pos.getConfidence() == 0)
    return null;

  return pos.getPosition();
}  // end of getJointPos()
```

Common reasons for a null joint position is if the joint is out of range of the Kinect camera (e.g. too near to it) or is obscured by another part of the body or an object in the scene.

5.3. Checking a Gesture

As will become clear as I explain some of the gesture checking methods, they're all coded in a similar way. A gesture is detected by comparing the positions of two joints. If the positions are sufficiently close, then a global boolean is set to true and the watcher is notified that the gesture has started.

If the positions aren't close enough, and the global boolean is true, then the boolean is set to false, and the watcher is notified that the gesture has ended.

For example, the "hands together" gesture is shown in Figure 6.

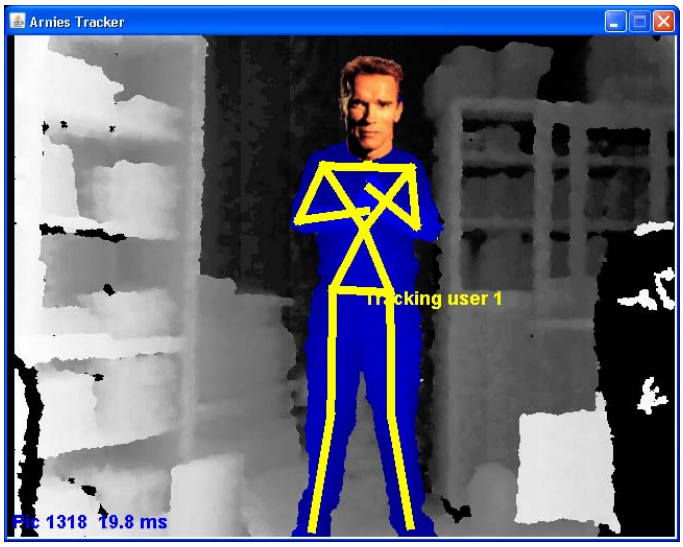

Figure 6. The "Hands Together" Gesture.

If I assume that the gesture is only performed while the user is facing the camera, then the left and right hands will be "close together" when the distance between their x-coordinates is small. This is implemented in the twoHandsNear() method:

```
// global
private float neckLength;            // neck to shoulder length
private boolean areHandsNear = false;
private GesturesWatcher watcher;


private void twoHandsNear(int userID,
         HashMap<SkeletonJoint, SkeletonJointPosition> skel)
// are the user's hand close together on the x-axis?
{
  Point3D leftHandPt = getJointPos(skel, SkeletonJoint.LEFT_HAND);
  Point3D rightHandPt = getJointPos(skel, SkeletonJoint.RIGHT_HAND);
  if ((leftHandPt == null) || (rightHandPt == null))
    return;

  float xDiff = rightHandPt.getX() - leftHandPt.getX();
  if (xDiff < neckLength) {    // near
    if (!areHandsNear) {
      watcher.pose(userID, GestureName.HANDS_NEAR, true);  //started
      areHandsNear = true;
    }
  }
  else {   // not near
    if (areHandsNear) {
      watcher.pose(userID, GestureName.HANDS_NEAR, false); //stopped
      areHandsNear = false;
    }
  }
}  // end of twoHandsNear()
```

The xDiff value should be a small positive value since I'm subtracting the left hand's x-coordinate from the right and, as shown in Figure 5, the left hand has a smaller x-value Of course the hands may be crossed but that will only make xDiff even smaller (i.e. negative).

The global boolean for twoHandsNear() is areHandsNear, which is used as the third argument of GesturesWatcher.pose(). pose() could do something complicated with the gesture information, but TrackerPanel only prints it out:

```
// in the TrackerPanel class
public void pose(int userID, GestureName gest, boolean isActivated)
// called by the gesture detectors
{
  if (isActivated)
    System.out.println(gest + " " + userID + " on");
  else
    System.out.println("         " + gest + " " + userID + " off");
}  // end of pose()
```


twoHandsNear() compares the x-axis distance between the hands using a short human-specific length, the neck-to-shoulder distance in neckLength.

I can justify my assumption about a forward facing user in twoHandsNear() by considering what would happen in other situations. For instance, if the user tries to perform the gesture while turned away from the Kinect, then it's very likely that one or more of the joint positions will return null, causing twoHandsNear() to do nothing. A null position is returned because the joints are obscured by part of the user's body.

This body-hiding problem makes it quite difficult to implement touching gestures, such as "hand touches head" because if the hand moves in front of the head then the head joint will be hidden, and its position will be returned as null. The most reliable forms of touching are when both joints remain visible to the Kinect, such as "hand to hip". Of course, this assumes that the user is facing the camera.

5.4. Turning to the Left

Turning to the left is illustrated in Figure 2, and is implemented using similar code, and similar assumptions, as in twoHandsNear(). If the user is facing the Kinect, then a turn can be detected by examining the relative z-axis positions of the two hips. A left turn involves the left hip moving further away from the camera, and the right hip moving towards it. Since the positive z-axis runs into the scene, this offset will be represented by a positive difference between the left and right hips, which is tested in turnLeft():

```
// globals
private float lowerArmLength;   // hand to elbow length
private boolean isTurnLeft = false;
private GesturesWatcher watcher;


private void turnLeft(int userID,
         HashMap<SkeletonJoint, SkeletonJointPosition> skel)
// has the user's right hip turned forward of his left hip?
{
  Point3D rightHipPt = getJointPos(skel, SkeletonJoint.RIGHT_HIP);
  Point3D leftHipPt = getJointPos(skel, SkeletonJoint.LEFT_HIP);
  if ((rightHipPt == null) || (leftHipPt == null))
    return;

  float zDiff = leftHipPt.getZ() - rightHipPt.getZ();
  if (zDiff > lowerArmLength) {    // right hip is forward
    if (!isTurnLeft) {
      watcher.pose(userID, GestureName.TURN_LEFT, true); //started
      isTurnLeft = true;
    }
  }
  else {   // not forward
    if (isTurnLeft) {
      watcher.pose(userID, GestureName.TURN_LEFT, false); //stopped
      isTurnLeft = false;
    }
  }
}  // end of turnLeft()
```


The difference between the hips' z-coordinates is stored in zDiff and compared against the lower arm length as a way of judging if the offset is large enough. The global boolean used to record the gesture's state is isTurnLeft.

This implementation may fail if the user turns so they are side-on to the Kinect since their left side will be hidden from the camera; the joints on that side, such as the left hip, will return a null position. Also, if the user turns even further to the left, so their back is towards the Kinect, then the zDiff value will start to decrease. When it drops below the lowerArmLength value, the gesture will no longer be detected.

5.5. Moving the Right Hand Up and Down

Before talking about right hand vertical waving, a so-called complex gesture, I need to explain its component basic gestures – raising and lowering the hand.

A simple way of detecting a raised right hand is to compare it's y-axis position with the user's head, as in rightHandUp():

```
// globals
private boolean isRightHandUp = false;
private GesturesWatcher watcher;

private GestureSequences gestSeqs;
       /* stores gesture sequences for each user,
          and looks for complex gestures */


private void rightHandUp(int userID,
       HashMap<SkeletonJoint, SkeletonJointPosition> skel)
// is the user's right hand at head level or above?
{
  Point3D rightHandPt = getJointPos(skel, SkeletonJoint.RIGHT_HAND);
  Point3D headPt = getJointPos(skel, SkeletonJoint.HEAD);
  if ((rightHandPt == null) || (headPt == null))
    return;

  if (rightHandPt.getY() <= headPt.getY()) {    // above
    if (!isRightHandUp) {
      watcher.pose(userID, GestureName.RH_UP, true);  // started
      gestSeqs.addUserGest(userID, GestureName.RH_UP);
                              // add to gesture sequence
      isRightHandUp = true;
    }
  }
  else {   // not above
    if (isRightHandUp) {
      watcher.pose(userID, GestureName.RH_UP, false);  // stopped
      isRightHandUp = false;
    }
  }
}  // end of rightHandUp()
```

This time it's not necessary to employ a body length to judge the difference between two coordinates. Simply testing if the hand is at the same height (or higher) than the head is sufficient. Care must be taken to remember that the positive y-axis runs down the screen, so a raised hand has a smaller y-value than the head.

This gesture uses the global boolean isRightHandUp to record its status.

The new element in this code is the GestureSequences object. When a right-hand-up gesture is started (labeled as GestureName.RH_UP), it's passed to GestureSequences where it's added to the gesture sequences list for that userID, and more complex gestures are detected.

All the gesture methods, such as twoHandsNear() and turnLeft(), should contain similar calls to GestureSequences.addUserGest() as rightHandUp(), but I've only implemented code for finding vertical and horizontal waving as yet. As a consequence, only the gesture methods for right hand positioning (rightHandUp(), rightHandFwd(), rightHandOut(), rightHandIn(), and rightHandDown()) call addUserGest().

An example of a lowered right hand is shown in Figure 3, and suggests a simple test – comparing the hand's y-axis position with the right hip. This is implemented in rightHandDown():

```
// globals
private boolean isRightHandDown = false;
private GesturesWatcher watcher;
private GestureSequences gestSeqs;


private void rightHandDown(int userID,
        HashMap<SkeletonJoint, SkeletonJointPosition> skel)
// is the user's right hand at hip level or below?
{
  Point3D rightHandPt = getJointPos(skel, SkeletonJoint.RIGHT_HAND);
  Point3D hipPt = getJointPos(skel, SkeletonJoint.RIGHT_HIP);
  if ((rightHandPt == null) || (hipPt == null))
    return;

  if (rightHandPt.getY() >= hipPt.getY()) {    // below
    if (!isRightHandDown) {
      watcher.pose(userID, GestureName.RH_DOWN, true);  // started
      gestSeqs.addUserGest(userID, GestureName.RH_DOWN);
                              // add to gesture sequence
      isRightHandDown = true;
    }
  }
  else {   // not below
    if (isRightHandDown) {
      watcher.pose(userID, GestureName.RH_DOWN, false);  // stopped
      isRightHandDown = false;
    }
  }
}  // end of rightHandDown()
```

In this method, GestureName.RH_DOWN is passed to the GestureSequences object when the gesture starts.

6. Complex Gestures

GestureSequences stores basic gesture sequences for each user, and detects complex gestures by looking for specified sub-sequences.

The constructor creates the main data structure – a HashMap that maps user IDs to gesture sequence lists:

```
// globals
private GesturesWatcher watcher;
private HashMap<Integer, ArrayList<GestureName>> userGestSeqs;


public GestureSequences(GesturesWatcher gw)
{ watcher = gw;
  userGestSeqs = new HashMap<Integer, ArrayList<GestureName>>();
}
```

The Skeletons class' observers add and remove users to/from the map: CalibrationCompleteObserver calls GestureSequences.addUser() and LostUserObserver calls GestureSequences.removeUser().

```
public void addUser(int userID)
// create a new empty gestures sequence for a user
{  userGestSeqs.put(new Integer(userID),
                   new ArrayList<GestureName>());  }


public void removeUser(int userID)
// remove the gesture sequence for this user
{  userGestSeqs.remove(userID); }
```

A particular user's sequence is extended by SkeletonsGestures calling addUserGest():

```
public void addUserGest(int userID, GestureName gest)
// add an gesture to the end of a user's sequence
{
  ArrayList<GestureName> gestsSeq = userGestSeqs.get(userID);
  if (gestsSeq == null)
    System.out.println("No gestures sequence for user " + userID);
  else
    gestsSeq.add(gest);
}  // end of addUserGest()
```

As I mentioned in the last section, addUserGest() is currently only called by the right hand position methods in SkeletonsGestures, so only sequences of right hand gestures are stored in the GestureSequences object.

6.1. Finding a Complex Gesture

Skeletons.update() calls GestureSequences.checkSeq() to search for complex gestures. At present, only vertical and horizontal waving with the right hand is detected, but the class' functionality could be easily extended.

The two forms of waving are defined as arrays of basic gestures:


```
// globals
private final static GestureName[] HORIZ_WAVE =
                 { GestureName.RH_OUT, GestureName.RH_IN,
                   GestureName.RH_OUT, GestureName.RH_IN };
      // a horizontal wave is two out-in moves of the right hand


private final static GestureName[] VERT_WAVE =
                 { GestureName.RH_UP, GestureName.RH_DOWN,
                   GestureName.RH_UP, GestureName.RH_DOWN };
      // a vertical wave is two up-down moves of the right hand
```

checkSeq() looks for the arrays' values in a user's gestures sequence. If a matching sub-sequence is found, then that part of the user's gesture sequence is deleted.

```
private void checkSeq(int userID, ArrayList<GestureName> gestsSeq)
{
  int endPos = findSubSeq(gestsSeq, HORIZ_WAVE);
                        // look for a horizontal wave
  if (endPos != -1) {   // found it
    watcher.pose(userID, GestureName.HORIZ_WAVE, true);
    purgeSeq(gestsSeq, endPos);
  }

  endPos = findSubSeq(gestsSeq, VERT_WAVE);
                       // look for a vertical wave
  if (endPos != -1) {  // found it
    watcher.pose(userID, GestureName.VERT_WAVE, true);
    purgeSeq(gestsSeq, endPos);
  }
}  // end of checkSeq()
```

checkSeq() only calls GesturesWatcher.pose() with the isActivated boolean argument set to true. I decided not to notify the watcher when a complex gesture finishes.

findSubSeq() looks for all the array values inside a list, and returns the position after the last value, or -1. The array elements don't have to be stored contiguously in the list.

```
private int findSubSeq(ArrayList<GestureName> gestsSeq,
                       GestureName[] gests)
{ int pos = 0;
  for(GestureName gest : gests) {   // iterate through array
    while (pos < gestsSeq.size()) {  // find gesture in list
      if (gest == gestsSeq.get(pos))
        break;
      pos++;
    }
    if (pos == gestsSeq.size())
      return -1;
    else
      pos++;   // carry on, starting with next gesture in list
  }
  return pos;
}  // end of findSubSeq()
```


The non-contiguous condition allows a complex gesture to be mixed in amongst other gestures. For example, the output for vertical waving shown near to Figure 3 shows that up and down moves are interspersed with right-hand-forward (RH_FWD) gestures when the user's hand is forward of the torso. This arm position often occurs as the user moves their hand up and down, but the waving gesture is still observed.

If a complex gesture sub-sequence is found, it's deleted from the user's sequence by purgeSeq(), including any other gestures mixed in among the sub-sequence.

```
private void purgeSeq(ArrayList<GestureName> gestsSeq, int pos)
/* remove all the elements in the seq between the positions
   0 and pos-1  */
{
  for (int i=0; i < pos; i++) {
    if (gestsSeq.isEmpty())
      return;
    gestsSeq.remove(0);
  }
}  // end of purgeSeq()
```

Deletion means that the user's sequence doesn't keep growing without end, but it also means that a basic gesture can only form part of one complex gesture, after which it is deleted.

It may be a better coding strategy not to delete all the intervening basic gestures since this means that it isn't possible to detect two complex gestures (such as right hand vertical waving and left hand horizontal waving) that are occurring at the same time.

