Kinect Chapter 4. Arnies Tracker

[**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 next two chapters looks at the skeletal tracking features offered by the Kinect, displaying them in 2D in this chapter, and in 3D (with the aid of Java 3D) in the next.

This chapter's ArniesTracker application is shown in action in Figure 1.

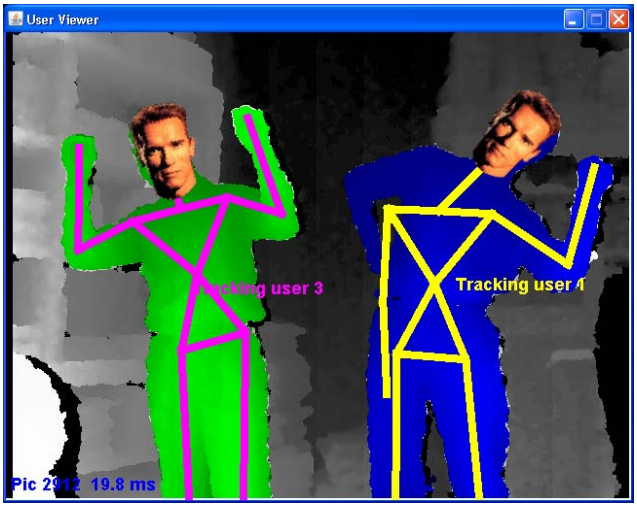

Figure 1. Two Arnies Being Tracked.

The background is a grayscale depth map, created in much the same way as in chapter 2. The additional features are:

- each body outline is colored;
- each outline comes with a colored 'skeleton', consisting of lines linking the skeleton's joint positions;
- each head has an image drawn over it, which rotates around the z-axis (perpendicular to the screen), so it stays aligned with the head-neck line;
- each body is labeled with status information, including a user ID.

It's somewhat hard to see but the two users in Figure 1 have IDs 1 and 3. User 2 (which was me) has just walked out of the camera's field of view in order to take the screenshot.

Figure 2 shows more of user 1's skeleton.

Figure 2. A Dancing Arnie.

Class diagrams for ArniesTracker are given in Figure 3.

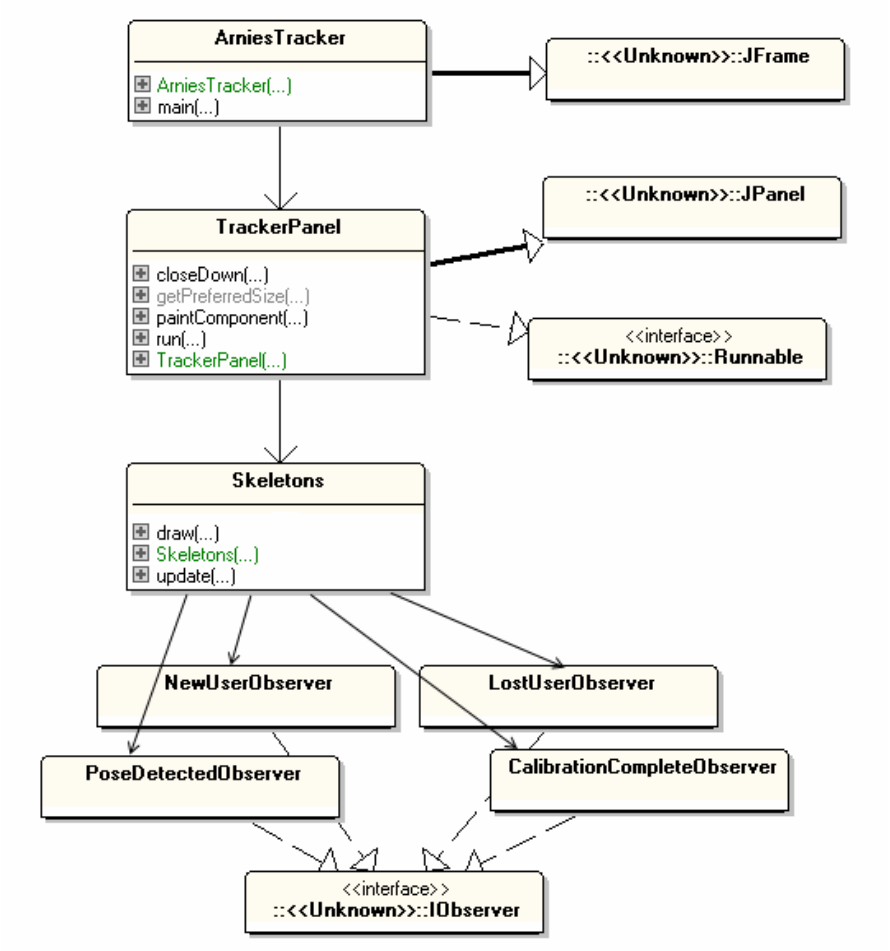

Figure 3. The ArniesTracker Class Diagrams.

The ArniesTracker class instantiates the TrackerPanel and places it in a window.

TrackerPanel creates the OpenNI production nodes, and its main update/drawing task is to display the grayscale depth map with colored user outlines.

The user skeletons are maintained, updated, and drawn by the Skeletons class. Skeletons sets up four 'observers' (listeners) so that when a new user is detected in the scene, a skeleton can be calibrated and tracked. This requires that the UserGenerator node supports pose and skeleton detection capabilities.

ArniesTracker is based on the UserTracker example from the OpenNI Java samples.

1. Configuring the Tracker

TrackerPanel's configOpenNI() creates a context and populates it with production nodes. It also initializes the Skeleton object.

```
// globals
private Context context;
private DepthMetaData depthMD;
private SceneMetaData sceneMD;

private Skeletons skels;   // the users' skeletons


private void configOpenNI()
{
  try {
    context = new Context();

    // add the NITE License
    License = new License("PrimeSense",
                          "0KOIk2JeIBYClPWVnMoRKn5cdY4=");
    context.addLicense(license);

    DepthGenerator depthGen = DepthGenerator.create(context);
    MapOutputMode mapMode = new MapOutputMode(640, 480, 30);
    depthGen.setMapOutputMode(mapMode);

    context.setGlobalMirror(true);         // set mirror mode

    depthMD = depthGen.getMetaData();
         // use depth metadata to access depth info

    UserGenerator userGen = UserGenerator.create(context);
    sceneMD = userGen.getUserPixels(0);
       // used to return a map containing user IDs
       // (or 0) at each depth location

    skels = new Skeletons(userGen, depthGen);

    context.startGeneratingAll();
    System.out.println("Started context generating...");
  }
  catch (Exception e) {
    System.out.println(e);
    System.exit(1);
  }
```



```
}  // end of configOpenNI()
```

As in previous examples, a DepthMetaData object is employed to create the depth map.

The UserGenerator production node is passed to the Skeletons object, where it will generate joint information for the bodies detected in the scene. It's also used to create a SceneMetaData object, which will be used later to access a different form of depth map (a so-called *scene map*), where each pixel holds a user ID (1, 2, etc.) if it's part of the user's body, or 0 to mean it is part of the background. This scene map will be used to colorize the bodies in the depth map.

1.1. Updating the Scene

TrackerPanel.run() executes a wait-update-redraw loop:

```
// globals
private volatile boolean isRunning;
private Context context;
private Skeletons skels;   // the users' skeletons


public void run()
{
  isRunning = true;
  while (isRunning) {
    try {
      context.waitAnyUpdateAll();
    }
    catch(StatusException e)
    {  System.out.println(e);
       System.exit(1);
    }
    long startTime = System.currentTimeMillis();
    updateUserDepths();
    skels.update();
    imageCount++;
    totalTime += (System.currentTimeMillis() - startTime);
    repaint();
  }
  // close down
  try {
    context.stopGeneratingAll();
  }
  catch (StatusException e) {}
  context.release();
  System.exit(0);
}  // end of run()
```

Depth map creation and coloring is performed by updateUserDepths(), and the updating of the skeletons is delegated to Skeletons.update().

updateUserDepths() starts in a similar way to my other depth map examples, by calling a function to build a histogram[] array of depths. What's new is that it utilizes the scene map to colorize the resulting depth map.


```
// globals
private Color USER_COLORS[] = {
  Color.RED, Color.BLUE, Color.CYAN, Color.GREEN,
  Color.MAGENTA, Color.PINK, Color.YELLOW, Color.WHITE};
    /* colors used to draw each user's depth image, except
       the last (white) which is for the background */

private byte[] imgbytes;
private float histogram[]; // for the depth values


private void updateUserDepths()
{
  ShortBuffer depthBuf = depthMD.getData().createShortBuffer();
  calcHistogram(depthBuf);
  depthBuf.rewind();

  // use user IDs to color the depth map
  ShortBuffer usersBuf = sceneMD.getData().createShortBuffer();

  while (depthBuf.remaining() > 0) {
    int pos = depthBuf.position();
    short depthVal = depthBuf.get();
    short userID = usersBuf.get();

    imgbytes[3*pos] = 0;   // default color is black
    imgbytes[3*pos + 1] = 0;
    imgbytes[3*pos + 2] = 0;

    if (depthVal != 0) {   // there is depth data
      // convert userID to index into USER_COLORS[]
      int colorIdx = userID % (USER_COLORS.length-1);
                                        // skip last color
      if (userID == 0)    // not a user; actually the background
        colorIdx = USER_COLORS.length-1;
            // last index is white's position in USER_COLORS[]

      // convert histogram value (0.0-1.0f) to a RGB color
      float histValue = histogram[depthVal];
      imgbytes[3*pos] =
          (byte) (histValue * USER_COLORS[colorIdx].getRed());
      imgbytes[3*pos + 1] =
          (byte) (histValue * USER_COLORS[colorIdx].getGreen());
      imgbytes[3*pos + 2] =
          (byte) (histValue * USER_COLORS[colorIdx].getBlue());
    }
  }
}  // end of updateUserDepths()
```

I won't explain calcHistogram() in detail since it's almost unchanged from previous examples. It stores the number of different depths, converts them into a cumulative depth count, and then into fractions between 0 and 1.

The while loop inside updateUserDepths() cycles through the depth map (stored in depthBuf) and the scene map (stored in usersBuf). The two maps are the same size, but the scene map stores integers in each pixel location – either a user ID (e.g. 1, 2, 3), or 0 to denote the background. The integers in the scene map are treated as indices

into a USER_COLORS[] colors array, with 0 (the background ID) being mapped to the last color in the array (WHITE)

The colors are stored as RGB bytes in the imgbytes[] array after being scaled by the corresponding histogram[] value, which makes the pixels darker if they are further from the Kinect sensor.

1.2. Drawing the Scene

TrackerPanel.paintComponent() draw the colored depth image, and statistics info, but delegates skeletons rendering to the Skeletons instance.

```
// global
private Skeletons skels;   // the users' skeletons


public void paintComponent(Graphics g)
{
  super.paintComponent(g);
  Graphics2D g2d = (Graphics2D) g;

  drawUserDepths(g2d);
  g2d.setFont(msgFont);    // for the status and stats text
  skels.draw(g2d);
  writeStats(g2d);
} // end of paintComponent()
```

drawUserDepths() creates an 8-bit RGB channel color model, representing how the colors are stored inside imgbytes[], fills a raster with the contents of imgbytes[], and combines the color model and raster into a BufferedImage.

```
// global
private byte[] imgbytes;


private void drawUserDepths(Graphics2D g2d)
{
  // define an 8-bit RGB channel color model
  ColorModel colorModel = new ComponentColorModel(
              ColorSpace.getInstance(ColorSpace.CS_sRGB),
              new int[] { 8, 8, 8}, false, false,
              ComponentColorModel.OPAQUE, DataBuffer.TYPE_BYTE);

  // fill the raster with the depth image bytes
  DataBufferByte dataBuffer =
        new DataBufferByte(imgbytes, imWidth*imHeight*3);
  WritableRaster raster =
        Raster.createInterleavedRaster(dataBuffer, imWidth,
            imHeight, imWidth*3, 3, new int[] { 0, 1, 2}, null);

  // combine color model and raster to create a BufferedImage
  BufferedImage image =
         new BufferedImage(colorModel, raster, false, null);

  g2d.drawImage(image, 0, 0, null);
}  // end of drawUserDepths()
```


2. Managing Skeletons

The key data structure in the Skeletons object is a table of tables called userSkels, defined as:

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

The first table (HashMap) maps user IDs to skeletons, with each skeleton represented by a table that maps joint labels (e.g. SkeletonJoint.HEAD, SkeletonJoint.NECK) to (x, y, z) positions. userSkels is illustrated in Figure 4.

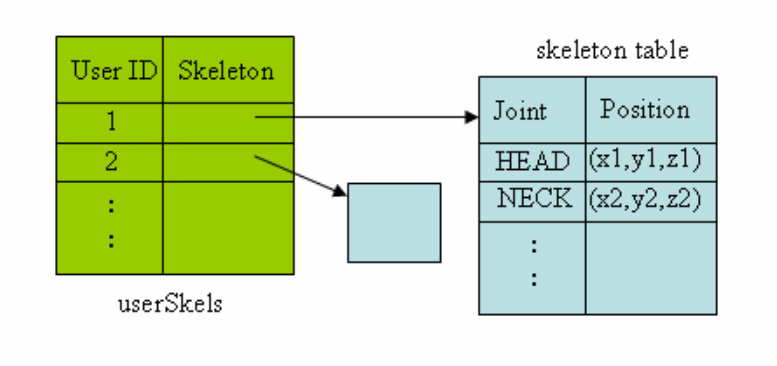

Figure 4. The userSkels Table of Tables

SkeletonJoint and SkeletonJointPosition are OpenNI classes, with SkeletonJoint an enum class containing the names of 24 joints for a body, shown in Figure 5.

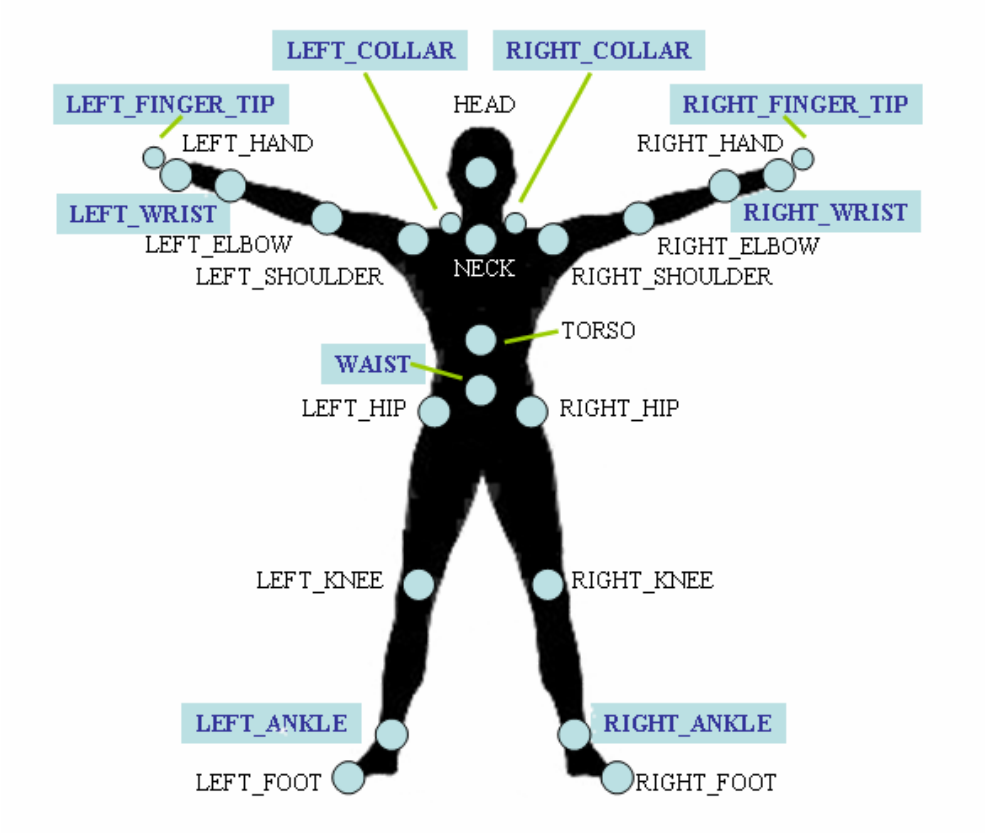

Figure 5. The SkeletonJoint Labels.

The figure is shown as a user appears on-screen after mirroring mode has been switched on. This means that the "left" and "right" labelling correspond to a user's left and right hand sides. For example, the RIGHT_HAND joint is the user's right hand.

Nine of the joint are 'unavailable' in this version of the OpenNI API (those with blue labels in Figure 5), which means that SkeletonCapability.isJointAvailable(SkeletonJoint j) returns false when j is assigned one of those labels. Any API calls using these joint labels will return false or may fail.

Skeleton tracking requires the pose and skeleton detection capabilities of UserGenerator, which are accessed in Skeletons.configure().

```
// globals
private UserGenerator userGen;
private DepthGenerator depthGen;

// capabilities used by UserGenerator
private SkeletonCapability skelCap;
     // to output skeletal data, including location of joints

private PoseDetectionCapability poseDetectionCap;
    // to recognize when the user is in a specific position

private String calibPoseName = null;


private void configure()
/* create pose and skeleton detection capabilities for the
```



```
   user generator, and set up observers (listeners)  */
{
  try {
    // set up UserGenerator pose and skeleton detection capabilities;
    poseDetectionCap = userGen.getPoseDetectionCapability();
    skelCap = userGen.getSkeletonCapability();

    calibPoseName = skelCap.getSkeletonCalibrationPose();
    skelCap.setSkeletonProfile(SkeletonProfile.ALL);

    // set up four observers
    userGen.getNewUserEvent().addObserver(new NewUserObserver());
              // new user found

    userGen.getLostUserEvent().addObserver(new LostUserObserver());
              // lost a user

    poseDetectionCap.getPoseDetectedEvent().addObserver(
                                new PoseDetectedObserver());
              // for when a pose is detected

    skelCap.getCalibrationCompleteEvent().addObserver(
                                new CalibrationCompleteObserver());
              // for when skeleton calibration is completed,
              // and tracking starts
  }
  catch (Exception e) {
    System.out.println(e);
    System.exit(1);
  }
}  // end of configure()
```

2.1. Listening for Users

Four listeners are set up by configure(), two for the user generator, and one each for the pose and skeleton capabilities. Listeners are necessary because the process of finding a new user, pose detection, skeleton calibration, and skeleton tracking can be lengthy, and we don't want the Skeletons object to wait for it to complete. In addition, the listener approach means that several users can be tracked at once.

The process of setting up skeleton tracking for a new user is shown in Figure 6.

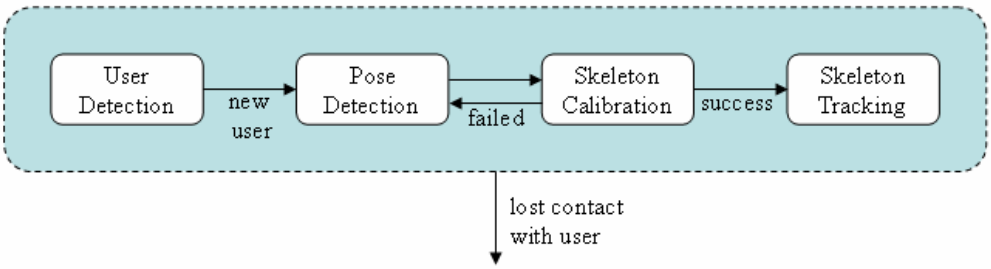

Figure 6. Steps in Tracking a New User.

The NewUserObserver class below corresponds to the "User Detection" box in Figure 6, and starts pose detection by calling PoseDetectionCapability.StartPoseDetection().

It looks for OpenNI's standard "psi" pose (a body with both arms raised, as in the left hand use in Figure 1).

```
// global
private PoseDetectionCapability poseDetectionCap;
private String calibPoseName;


class NewUserObserver implements IObserver<UserEventArgs>
{
  public void update(IObservable<UserEventArgs> observable,
                                          UserEventArgs args)
  {
    System.out.println("Detected new user " + args.getId());
    try {
      // try to detect a pose for the new user
      poseDetectionCap.StartPoseDetection(calibPoseName,
                                              args.getId());
    }
    catch (StatusException e)
    { e.printStackTrace(); }
  }
}  // end of NewUserObserver inner class
```

The PoseDetectedObserver class corresponds to the "Pose Detection" box in Figure 6. Its update() method is called when the "psi" pose is detected (which may take several seconds) – pose detection is terminated and skeleton calibration begun by calling SkeletonCapability.requestSkeletonCalibration ().

```
// globals
private SkeletonCapability skelCap;
private PoseDetectionCapability poseDetectionCap;


class PoseDetectedObserver implements
                               IObserver<PoseDetectionEventArgs>
{
  public void update(IObservable<PoseDetectionEventArgs> observable,
                             PoseDetectionEventArgs args)
  {
    int userID = args.getUser();
    System.out.println(args.getPose() +
                     " pose detected for user " + userID);
    try {
      // finish pose detection; switch to skeleton calibration
      poseDetectionCap.StopPoseDetection(userID);
      skelCap.requestSkeletonCalibration(userID, true);
    }
    catch (StatusException e)
    {  e.printStackTrace(); }
  }
}  // end of PoseDetectedObserver inner class
```

When the calibration is completed, the update() method in CalibrationCompleteObserver is invoked, corresponding to the end of the "Skeleton Calibration" in Figure 6. Calibration may succeed or fail (perhaps because not enough

of the user is visible to the Kinect), and so an if-test inside update() decides whether to continue onto tracking or to return to pose detection.

```
// globals
private SkeletonCapability skelCap;
private PoseDetectionCapability poseDetectionCap;

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


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

If user tracking is about to commence then a user entry is added to the userSkels data structure, which links the user's ID (e.g. 1, 2) to an empty skeleton table. The table will be filled with joint positions during updates to the Skeletons object.

The steps leading from the detection of a new user to the tracking of his skeleton in Figure 6 can be seen in the sequence of prints made by the listeners in Figure 7.

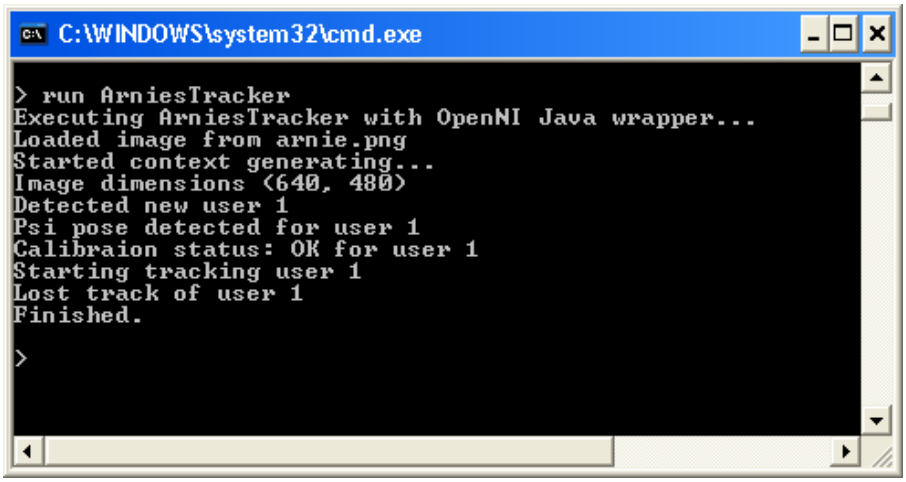

Figure 7. Outputs from the Listeners.

A user may be lost at any time, perhaps because they walk out of the range of the Kinect sensor, and another listener deals with that:

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

class LostUserObserver implements IObserver<UserEventArgs>
{
  public void update(IObservable<UserEventArgs> observable,
                     UserEventArgs args)
  {
    System.out.println("Lost track of user " + args.getId());
    userSkels.remove(args.getId());    // remove user from userSkels
  }
} // end of LostUserObserver inner class
```

The output from this listener can be seen in Figure 8, which occurs about 10 seconds after the user walks out of range of the Kinect sensor.

It's necessary to watch for a disappearing user so their userSkels entry can be removed.

3. Other Listeners

OpenNI supports numerous kinds of listeners (observers), all of which implement the IObserver interface and its update() method. A quick look back at the four listeners from before shows their general coding style:

```
class Foo implements IObserver<EventArgs-subclass>
{
  public void update(IObservable<EventArgs-subclass> observable,
                     EventArgs-subclass args)
  {
    int userID = args.getUser();  // or sometimes getID()
```



```
    System.out.println(User: " + userID);
    try {
      // do something with args values
    }
    catch (StatusException e)
    {  e.printStackTrace(); }
  }
}  // end of Foo class
```

The details of the listener depends on which subclass of EventArgs it uses; there are twelve, shown in Figure 8.

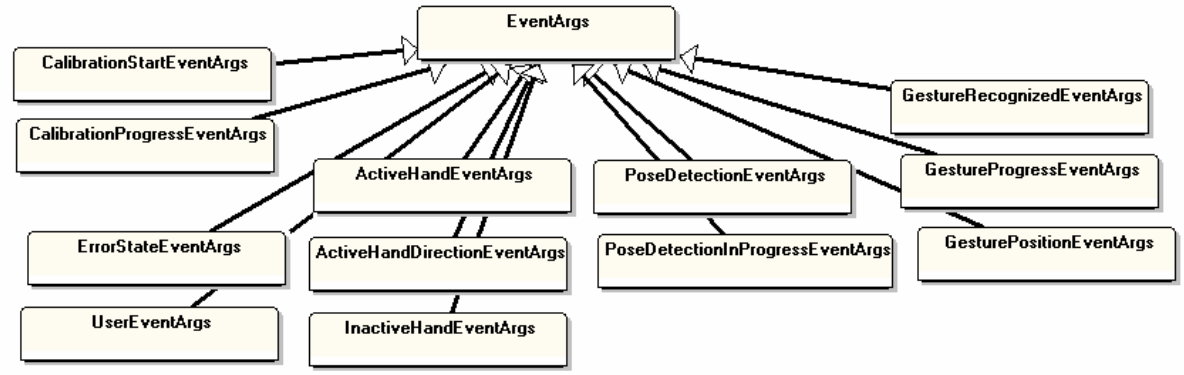

Figure 8. EventArgs and its Subclasses.

For example, the UserGenerator class maintains four listeners, all of which use UserEventArgs:

```
private Observable<UserEventArgs> newUserEvent;
private Observable<UserEventArgs> lostUserEvent;
private Observable<UserEventArgs> userExitEvent;
private Observable<UserEventArgs> userReenterEvent;
```

They're attached to a UserGenerator object with the methods getNewUserEvent(), getLostUserEvent(), getUserExitEvent(), and getUserReenterEvent(). In Skeletons, I've only used the listeners for watching for new and lost users.

4. Updating the Skeletons

TrackerPanel executes a wait-update-draw loop inside its run() method, which calls Skeletons.update() as part of its update stage.

```
// the Skeletons class
// globals
private UserGenerator userGen;
private SkeletonCapability skelCap;


public void update()
{
```



```
  try {
    int[] userIDs = userGen.getUsers();
    for (int i = 0; i < userIDs.length; ++i) {
      int userID = userIDs[i];
      if (skelCap.isSkeletonCalibrating(userID))
        continue;
      if (skelCap.isSkeletonTracking(userID))
        updateJoints(userID);
    }
  }
  catch (StatusException e)
  {  System.out.println(e); }
}  // end of update()
```

The UserGenerator is queried about the detected users (i.e. those that have left the "User Detection" stage of Figure 6 (but haven't been lost). For instance, in the case of Figure 1, UserGenenator.getUsers() returns a two-element array containing the IDs 1 and 3.

It's necessary to determine whether a user has reached the tracking stage, before updating their skeleton. That should be possible by simply calling SkeletonCapability.isSkeletonTracking() with the user's ID. However, in the current API, this can occasionally cause the application to crash if the skeleton is still being calibrated. To avoid that problem, I added a SkeletonCapability. isSkeletonCalibrating() test to update().

If a user is being tracked, then updateJoints() is called, which makes multiple calls to updateJoint(), once for each of the available joints (the ones without blue labels in Figure 5.

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


private void updateJoints(int userID)
{
  HashMap<SkeletonJoint, SkeletonJointPosition> skel =
                      userSkels.get(userID);   // the user's skeleton

  updateJoint(skel, userID, SkeletonJoint.HEAD);
  updateJoint(skel, userID, SkeletonJoint.NECK);

  updateJoint(skel, userID, SkeletonJoint.LEFT_SHOULDER);
  updateJoint(skel, userID, SkeletonJoint.LEFT_ELBOW);
  updateJoint(skel, userID, SkeletonJoint.LEFT_HAND);

  updateJoint(skel, userID, SkeletonJoint.RIGHT_SHOULDER);
  updateJoint(skel, userID, SkeletonJoint.RIGHT_ELBOW);
  updateJoint(skel, userID, SkeletonJoint.RIGHT_HAND);

  updateJoint(skel, userID, SkeletonJoint.TORSO);

  updateJoint(skel, userID, SkeletonJoint.LEFT_HIP);
  updateJoint(skel, userID, SkeletonJoint.LEFT_KNEE);
  updateJoint(skel, userID, SkeletonJoint.LEFT_FOOT);

  updateJoint(skel, userID, SkeletonJoint.RIGHT_HIP);
```



```
  updateJoint(skel, userID, SkeletonJoint.RIGHT_KNEE);
  updateJoint(skel, userID, SkeletonJoint.RIGHT_FOOT);
}  // end of updateJoints()
```

A somewhat briefer way of coding updateJoints() is to iterate through the SkeletonJoint enumeration:

```
private void updateJoints(int userID)
{
  HashMap<SkeletonJoint, SkeletonJointPosition> skel =
                                        userSkels.get(userID);
  for(SkeletonJoint j : SkeletonJoint.values())
    updateJoint(skel, userID, j);
}  // end of updateJoints()
```

The drawback is that updateJoint() will be called to examine the nine non-implemented joints, which is a waste of time (and generates a multitude of warning messages). Also, this approach assumes that tracking is enabled for the entire skeleton, which it is in this case because of the configure() call :

```
skelCap.setSkeletonProfile(SkeletonProfile.ALL);
```

However, there are other skeletal profiles which use a subset of the joints, including SkeletonProfile.UPPER_BODY, SkeletonProfile.LOWER_BODY, and SkeletonProfile.HEAD_HANDS. Iterating through all the joints for these profiles is pointless.

updateJoint() updates the 3D position of the specified user's joint by querying the SkeletonCapability object.

```
private void updateJoint(
       HashMap<SkeletonJoint, SkeletonJointPosition> skel,
       int userID, SkeletonJoint joint)
{
  try {
    // report unavailable joints
    if (!skelCap.isJointAvailable(joint) ||
        !skelCap.isJointActive(joint)) {
      System.out.println(joint + " not available for updates");
      return;
    }

    SkeletonJointPosition pos =
            skelCap.getSkeletonJointPosition(userID, joint);
    if (pos == null) {
      System.out.println("No update for " + joint);
      return;
    }

    SkeletonJointPosition jPos = null;
    if (pos.getPosition().getZ() != 0)   // has a depth position
      jPos = new SkeletonJointPosition(
        depthGen.convertRealWorldToProjective(pos.getPosition()),
                                          pos.getConfidence());
    else  // no info found for that user's joint
      jPos = new SkeletonJointPosition(new Point3D(), 0);
    skel.put(joint, jPos);
  }
```



```
  catch (StatusException e)
  {  System.out.println(e); }
}  // end of updateJoint()
```

The code is complicated by the need to detect unavailable joints.

The position returned by SkeletonCapability.getSkeletonJointPosition() uses the coordinate system of the 3D scene, which needs to be converted into the 3D projection coordinates employed by the panel.

The SkeletonJointPosition objects stored in the HashMap contain a tiny bit more information than just the 3D coordinates I drew in Figure 4. Each object holds a 3D point *and* a confidence value between 0 and 1, where 0 means that there's no confidence in the accuracy of the coordinate.

A More Complex Skeleton

The userSkels data structure in Figure 4 is defined as:

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

The joints in each user skeleton are represented by 3D positions (with confidence values). The OpenNI Java API offers a more powerful way of representing joints, by employing the SkeletonJointTransformation class. The new version of userSkels would be:

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

SkeletonJointTransformation is a wrapper around two objects – a SkeletonJointPosition instance for a joint's position and a SkeletonJointOrientation object for its orientation. SkeletonJointOrientation can be thought of as 3x3 matrix representing the joint's x-, y-, and z- axis orientations, although the class encodes the matrix as nine floats internally.

Unfortunately, I was unable to use SkeletonJointTransformation because accessing a joint's orientation via a SkeletonCapability instance causes my version of the OpenNI API to crash Java. For example, if the following code fragment is added to updateJoint():

```
SkeletonJointOrientation ori =
        skelCap.getSkeletonJointOrientation(userID, joint);
if (ori == null)
  System.out.println("No orientation for " + joint);
else
  System.out.println("Orientation for " + joint + ": " + ori);
```

Then a fatal error occurs when SkeletonCapability.getSkeletonJointOrientation() is called.

It is possible to calculate a joints orientation without utilizing SkeletonJointOrientation, as I'll show when I draw a skeleton's head image.

5. Drawing the Skeletons

TrackerPanel calls Skeletons.draw() from paintComponent() during its rendering phase.

Skeletons.draw() draws a skeleton for each user currently being tracked. 'Skeleton' is a fancy name for a series of thick colored lines between the joint positions in Figure 9.

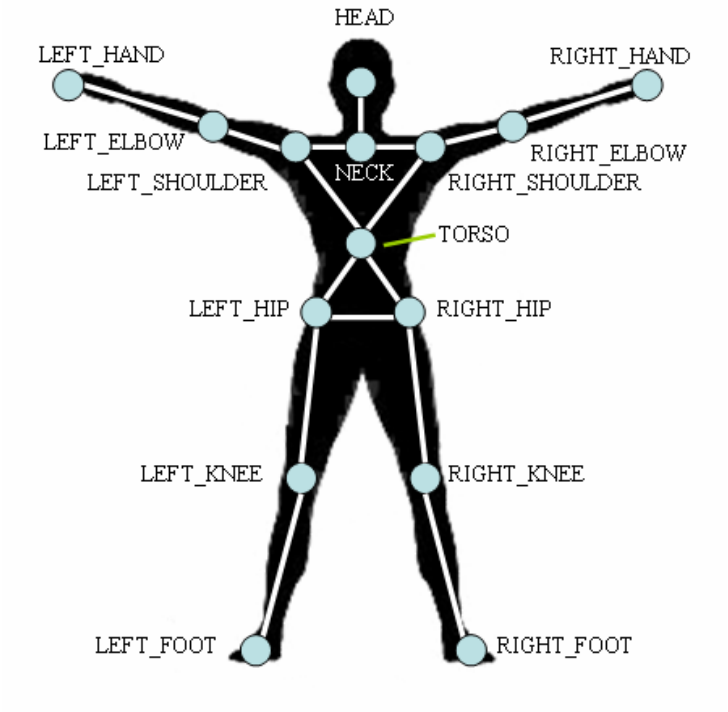

Figure 9. The Skeletal Limbs.

The lines in Figure 9 should be compared with those drawn on the users in Figures 1 and 2.

A head image is drawn centered over the head joint, rotated around that point so its vertical axis remains parallel with the line from the neck to the head. Also, status information is drawn at the center-of-mass of the body.

Skeletons.draw() iterates through all the users, but only draws those that are in the tracking stage.

```
// globals
private SkeletonCapability skelCap;
private HashMap<Integer,
     HashMap<SkeletonJoint, SkeletonJointPosition>> userSkels;


public void draw(Graphics2D g2d)
// draw skeleton of each user, with a head image, and user status
{
  g2d.setStroke(new BasicStroke(8));

  try {
    int[] userIDs = userGen.getUsers();
    for (int i = 0; i < userIDs.length; ++i) {
```



```
      setLimbColor(g2d, userIDs[i]);
      if (skelCap.isSkeletonCalibrating(userIDs[i]))
        {}  // avoid occasional crashes with isSkeletonTracking()
      else if (skelCap.isSkeletonTracking(userIDs[i])) {
        HashMap<SkeletonJoint, SkeletonJointPosition> skel =
                              userSkels.get(userIDs[i]);
        drawSkeleton(g2d, skel);
        drawHead(g2d, skel);
      }
      drawUserStatus(g2d, userIDs[i]);
    }
  }
  catch (StatusException e)
  {  System.out.println(e); }
}  // end of draw()
```

5.1. Drawing a Skeleton

Each skeleton is drawn as a collection of lines like those in Figure 9.

```
private void drawSkeleton(Graphics2D g2d,
             HashMap<SkeletonJoint, SkeletonJointPosition> skel)
{
  drawLine(g2d, skel, SkeletonJoint.HEAD, SkeletonJoint.NECK);

  drawLine(g2d, skel, SkeletonJoint.LEFT_SHOULDER,
                          SkeletonJoint.TORSO);
  drawLine(g2d, skel, SkeletonJoint.RIGHT_SHOULDER,
                          SkeletonJoint.TORSO);

  drawLine(g2d, skel, SkeletonJoint.NECK,
                          SkeletonJoint.LEFT_SHOULDER);
  drawLine(g2d, skel, SkeletonJoint.LEFT_SHOULDER,
                          SkeletonJoint.LEFT_ELBOW);
  drawLine(g2d, skel, SkeletonJoint.LEFT_ELBOW,
                          SkeletonJoint.LEFT_HAND);

  drawLine(g2d, skel, SkeletonJoint.NECK,
                          SkeletonJoint.RIGHT_SHOULDER);
  drawLine(g2d, skel, SkeletonJoint.RIGHT_SHOULDER,
                          SkeletonJoint.RIGHT_ELBOW);
  drawLine(g2d, skel, SkeletonJoint.RIGHT_ELBOW,
                          SkeletonJoint.RIGHT_HAND);

  drawLine(g2d, skel, SkeletonJoint.LEFT_HIP, SkeletonJoint.TORSO);
  drawLine(g2d, skel, SkeletonJoint.RIGHT_HIP, SkeletonJoint.TORSO);
  drawLine(g2d, skel, SkeletonJoint.LEFT_HIP,
                          SkeletonJoint.RIGHT_HIP);

  drawLine(g2d, skel, SkeletonJoint.LEFT_HIP,
                          SkeletonJoint.LEFT_KNEE);
  drawLine(g2d, skel, SkeletonJoint.LEFT_KNEE,
                          SkeletonJoint.LEFT_FOOT);

  drawLine(g2d, skel, SkeletonJoint.RIGHT_HIP,
                          SkeletonJoint.RIGHT_KNEE);
  drawLine(g2d, skel, SkeletonJoint.RIGHT_KNEE,
                          SkeletonJoint.RIGHT_FOOT);
```



```
}  // end of drawSkeleton()
```

drawLine() draws a line between two specified joints (if they have positions):

```
private void drawLine(Graphics2D g2d,
         HashMap<SkeletonJoint, SkeletonJointPosition> skel,
                          SkeletonJoint j1, SkeletonJoint j2)
{
  Point3D p1 = getJointPos(skel, j1);
  Point3D p2 = getJointPos(skel, j2);
  if ((p1 != null) && (p2 != null))
    g2d.drawLine((int) p1.getX(), (int) p1.getY(),
                         (int) p2.getX(), (int) p2.getY());
}  // end of drawLine()
```

getJointPos() reads a joint's position from the user's skeleton table. However, if the point's confidence level is 0, then a line isn't drawn.

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

  // don't draw line to joint with a zero-confidence pos
  if (pos.getConfidence() == 0)
    return null;

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

5.2. Drawing a Head Image

The head image is drawn so it's centered over the head joint, and rotated to stay parallel to the head-neck line. These criteria are illustrated by Figure 10.

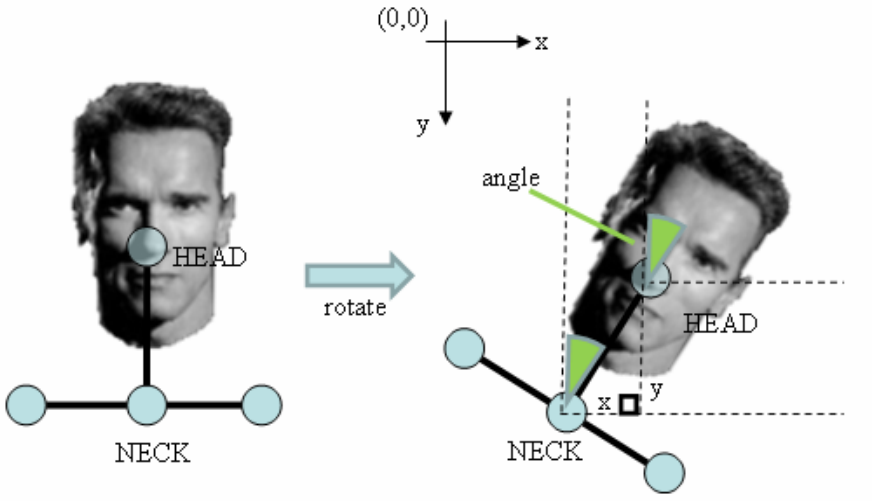

Figure 10. Rotating the Head.

Joint orientation information isn't available (at least in this version of the API), but it isn't hard to calculate the relative orientation of a joint relative to another. Assuming that the head joint starts directly above the neck joint, then the head's rotation relative to the neck is a matter of some trigonometry with their x- and y- axis values (remembering that the y-axis runs down the screen).

```
private void drawHead(Graphics2D g2d,
           HashMap<SkeletonJoint, SkeletonJointPosition> skel)
{
  if (headImage == null)
    return;

  Point3D headPt = getJointPos(skel, SkeletonJoint.HEAD);
  Point3D neckPt = getJointPos(skel, SkeletonJoint.NECK);
  if ((headPt == null) || (neckPt == null))
    return;
  else {
    int angle = 90 - ((int) Math.round( Math.toDegrees(
                 Math.atan2(neckPt.getY()-headPt.getY(),
                            headPt.getX()- neckPt.getX()) )));
    drawRotatedHead(g2d, headPt, headImage, angle);
  }
}  // end of drawHead()
```

Drawing a rotated image can be achieved by temporarily rotating the graphic context's drawing axes around the image's center.

```
private void drawRotatedHead(Graphics2D g2d, Point3D headPt,
                             BufferedImage headImage, int angle)
{
  AffineTransform origTF = g2d.getTransform();
                    // store original orientation
  AffineTransform newTF = (AffineTransform)(origTF.clone());

  // center of rotation is the head joint
  newTF.rotate( Math.toRadians(angle),
                  (int)headPt.getX(), (int)headPt.getY());
  g2d.setTransform(newTF);

  // draw image centered at head joint
  int x = (int)headPt.getX() - (headImage.getWidth()/2);
  int y = (int)headPt.getY() - (headImage.getHeight()/2);
  g2d.drawImage(headImage, x, y, null);

  g2d.setTransform(origTF);    // reset original orientation
}  // end of drawRotatedHead()
```

5.3. User Status Information

The status details printed at the center-of-mass (CoM) of each body include the user's ID and the current processing stage (i.e. whether OpenNI is in the "Pose Detection", "Skeleton Calibration" or "Skeleton Tracking" box of Figure 6).


```
private void drawUserStatus(Graphics2D g2d, int userID)
                                        throws StatusException
{
  Point3D massCenter = depthGen.convertRealWorldToProjective(
                                      userGen.getUserCoM(userID));
  String label = null;
  if (skelCap.isSkeletonTracking(userID))     // tracking
    label = new String("Tracking user " + userID);
  else if (skelCap.isSkeletonCalibrating(userID))  // calibrating
    label = new String("Calibrating user " + userID);
  else    // pose detection
    label = new String("Looking for " + calibPoseName +
                                    " pose for user " + userID);

  g2d.drawString(label, (int) massCenter.getX(),
                        (int) massCenter.getY());
}  // end of drawUserStatus()
```

The center-of-mass is obtained from the UserGenerator, and converted from its scene coordinates to those used by the panel. The skeleton processing stage is discovered by querying the SkeletonCapability object.

