Kinect Chapter 8. NITE Hands 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.]

This chapter is about utilizing OpenNI's gesture and hands generators with NITE's hand points to track multiple hands, and draw the tracks as 'trails' on top of the Kinect's camera image, as in Figure 1.

Figure 1. Multiple Hand Trails.

Hand point listeners were described in the previous chapter, and the coding here is similar. The main additional issues are getting NITE to recognize multiple hands (the default is only one), reporting session state changes, and rendering the hand point coordinates (which require real-world to screen coordinate mapping).

The application, called HandTrackers, consists of three classes, shown in Figure 2.

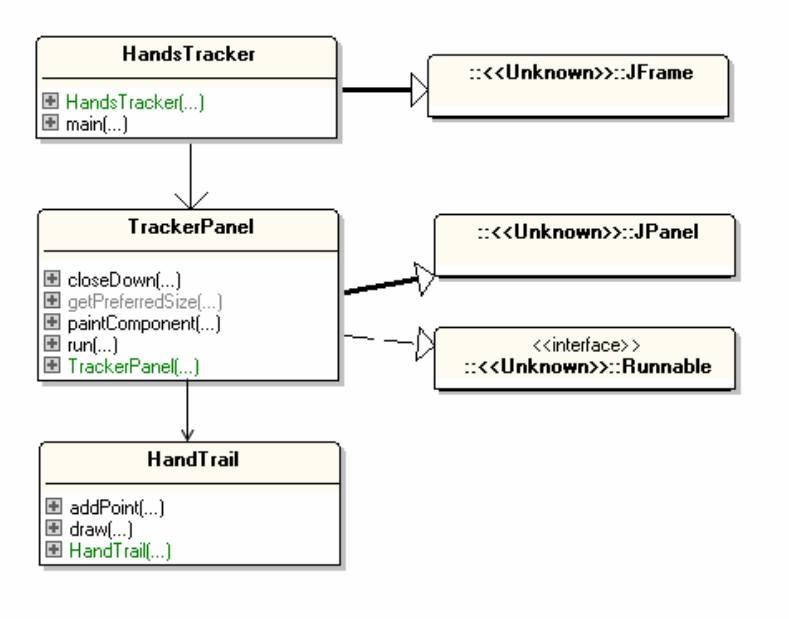

Figure 2. Class Diagrams for the HandsTracker Application.

The HandsTracker class is in charge of creating the application's JFrame, so won't be described. TrackerPanel initializes OpenNI and NITE, and creates a thread to wait for Kinect data, updates state information, and renders the Kinect's camera picture and the multiple hand trails. The trails are stored in a HashMap of HandTrail objects indexed by hand point IDs. For example, Figure 1 requires a map of four HandTrail objects, indexed by 1, 2, 4, and 6. These IDs can be seen (faintly) in the figure, drawn near the users' hands inside circles.

1. Configuring NITE

For multiple hands to be recognized by NITE, it's necessary to change a Nite.ini configuration file located in C:\Program Files\PrimeSense\NITE\Hands_1_4_0\Data (or somewhere similar). The two properties lines need to be uncommented, leaving the following:

```
[HandTrackerManager]
AllowMultipleHands=1
TrackAdditionalHands=1
```

It's a shame that these parameters can't be set programmatically from inside the HandsTracker code.

2. Configuring the Kinect

My TrackerPanel class calls configureKinect() to create four generators and several listeners.

```
// globals
private Context context;
private ImageGenerator imageGen;
```



```
private DepthGenerator depthGen;
private SessionManager sessionMan;
private SessionState sessionState;

private int imWidth, imHeight;


private void configKinect()
// set up OpenNI and NITE generators and listeners
{
  try {
    context = new Context();

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

    // set up image and depth generators
    imageGen = ImageGenerator.create(context);
         // for displaying the scene
    depthGen = DepthGenerator.create(context);
         // for converting real-world coords to screen coords

    MapOutputMode mapMode = new MapOutputMode(640, 480, 30);
    imageGen.setMapOutputMode(mapMode);
    depthGen.setMapOutputMode(mapMode);

    imageGen.setPixelFormat(PixelFormat.RGB24);

    ImageMetaData imageMD = imageGen.getMetaData();
    imWidth = imageMD.getFullXRes();
    imHeight = imageMD.getFullYRes();

    // set Mirror mode for all
    context.setGlobalMirror(true);

    // set up hands and gesture generators
    HandsGenerator hands = HandsGenerator.create(context);
    hands.SetSmoothing(0.1f);

    GestureGenerator gesture = GestureGenerator.create(context);

    context.startGeneratingAll();
    System.out.println("Started context generating...");

    // set up session manager and points listener
    sessionMan = new SessionManager(context, "Click,Wave",
                                            "RaiseHand");
    setSessionEvents(sessionMan);
    sessionState = SessionState.NOT_IN_SESSION;

    PointControl pointCtrl = initPointControl();
    sessionMan.addListener(pointCtrl);
  }
  catch (GeneralException e) {
    e.printStackTrace();
    System.exit(1);
  }
}  // end of configKinect()
```


Four generators are utilized: ImageGenerator, DepthGenerator, HandsGenerator, and GestureGenerator. ImageGenerator is used to access the Kinect's camera, while the DepthGenerator is employed for mapping read-world hand coordinates into screen positions. The HandsGenerator is required for monitoring the users' hands, and the GestureGenerator for identifying focus and refocus gestures.

The SessionManager's constructor is slightly different from the one in the last chapter. This time around, it specifies two focus gestures: clicking (similar to hand pushing) and waving, and the refocus gesture is raising the hand.

3. Tracking the Session

A NITE tracking session can move between three states: not in session, in session, and quick refocus, which are illustrated in Figure 3.

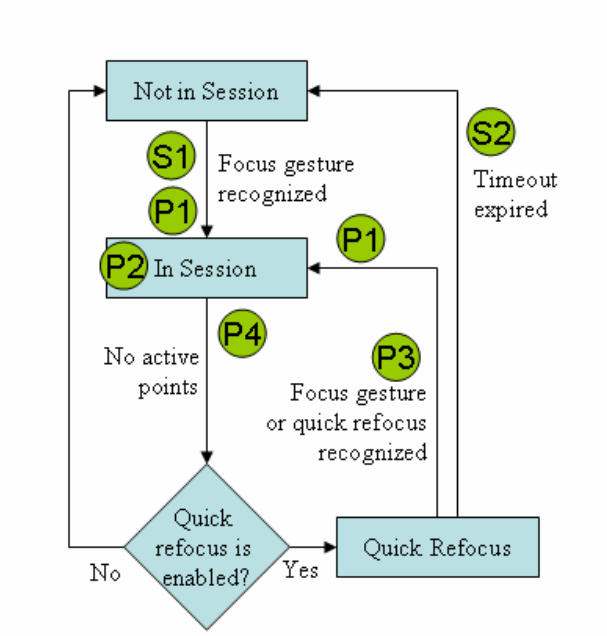

Figure 3. NITE Session States.

The circles containing S1, S2 and P1-P4 denote listeners triggered when the session states change, and are explained below.

I use the current session state to determine what message to write at the top-left of the image (see Figure 1, where it reads "Tracking 4 hands…"). The three possible states are defined as an enumerated type, and a global, sessionState, stores the current value:

```
enum SessionState {
    IN_SESSION, NOT_IN_SESSION, QUICK_REFOCUS
}

// global in TrackerPanel
private SessionState sessionState;

// in TrackerPanel.configKinect()
```



```
sessionState = SessionState.NOT_IN_SESSION;
```

The setSessionEvent() method sets up listeners to detect the start and end of a session.

```
// globals
private SessionManager sessionMan;
private SessionState sessionState;
private volatile boolean isRunning;


private void setSessionEvents(SessionManager sessionMan)
{
  try {
    // session start (S1)
    sessionMan.getSessionStartEvent().addObserver(
                          new IObserver<PointEventArgs>() {
      public void update(IObservable<PointEventArgs> observable,
                         PointEventArgs args)
      { System.out.println("Session started...");
        sessionState = SessionState.IN_SESSION;
      }
    });

    // session end (S2)
    sessionMan.getSessionEndEvent().addObserver(
                          new IObserver<NullEventArgs>() {
      public void update(IObservable<NullEventArgs> observable,
                         NullEventArgs args)
      { System.out.println("Session ended");
        isRunning = false;
        sessionState = SessionState.NOT_IN_SESSION;
      }
    });
  }
  catch (StatusException e) {
    e.printStackTrace();
  }
}  // end of setSessionEvents()
```

The listeners (labeled as S1 and S2 above, and in Figure 3) change the global sessionState variable. Also, the session end listener (S2) sets the global isRunning boolean to false, which causes the update-redraw thread in TrackerPanel to terminate.

4. Detecting Hand Points

The initPointControl() method sets up four listeners to respond to the creation of a hand point, its movement, its destruction, and the absence of active hand points.

```
// globals
private DepthGenerator depthGen;
private SessionState sessionState;
private HashMap<Integer, HandTrail> handTrails;
           // for storing multiple hand trails


private PointControl initPointControl()
{
```



```
PointControl pointCtrl = null;
try {
  pointCtrl = new PointControl();

  // create new hand point, and hand trail (P1)
  pointCtrl.getPointCreateEvent().addObserver(
                      new IObserver<HandEventArgs>() {
    public void update(IObservable<HandEventArgs> observable,
                      HandEventArgs args)
    { sessionState = SessionState.IN_SESSION;
      HandPointContext handContext = args.getHand();
      int id = handContext.getID();
      System.out.println("  Creating hand trail " + id);
      HandTrail handTrail = new HandTrail(id, depthGen);
      handTrail.addPoint( handContext.getPosition() );
      handTrails.put(id, handTrail);
    }
  });

  // hand point has moved; add to its trail (P2)
  pointCtrl.getPointUpdateEvent().addObserver(
                        new IObserver<HandEventArgs>() {
    public void update(IObservable<HandEventArgs> observable,
                      HandEventArgs args)
    { sessionState = SessionState.IN_SESSION;
      HandPointContext handContext = args.getHand();
      int id = handContext.getID();
      // System.out.println("  Extending hand trail " + id);
      HandTrail handTrail = handTrails.get(id);
      handTrail.addPoint( handContext.getPosition() );
    }
  });

  // destroy hand point and its trail (P3)
  pointCtrl.getPointDestroyEvent().addObserver(
                          new IObserver<IdEventArgs>() {
    public void update(IObservable<IdEventArgs> observable,
                      IdEventArgs args)
    { int id = args.getId();
      System.out.println("  Deleting hand trail " + id);
      handTrails.remove(id);
      if (handTrails.isEmpty())
        System.out.println("  No hand trails left...");
    }
  });

  // no active hand point, which triggers refocusing (P4)
  pointCtrl.getNoPointsEvent().addObserver(
                       new IObserver<NullEventArgs>() {
    public void update(IObservable<NullEventArgs> observable,
                       NullEventArgs args)
    { if (sessionState != SessionState.NOT_IN_SESSION) {
        System.out.println("  Lost hand point, so refocusing");
        sessionState = SessionState.QUICK_REFOCUS;
      }
    }
  });

}
catch (GeneralException e) {
  e.printStackTrace();
```



```
  }
  return pointCtrl;
}  // end of initPointControl()
```

The four listeners are labeled P1-P4 in the code above, and in Figure 3.

A new hand point causes the P1 listener to create a new hand trail for the specified hand ID. The HandTrail object is passed the ID and the DepthGenerator so hand coordinates can be mapped to the screen. Each HandTrail instance is stored in a global map, handTrails, using its hand ID as the key.

As the user moves their hand, the P2 listener will be invoked many times, and each hand coordinate added to the relevant trail.

If HandsGenerator can't find any hand points (perhaps because the users are hiding their hands), then the in-session state is left (see Figure 3), and the next state depends on whether the SessionManager has quick refocusing enabled. It is enabled in HandsTracker by having the sessionMan object employ hand raising as the refocus gesture:

```
sessionMan = new SessionManager(context, "Click,Wave",
                                         "RaiseHand");
```

The state change is noted by having the P4 listener change the session state to QUICK_REFOCUS.

The return to the session can be achieved by the user performing the quick refocus gesture ("RaiseHand") or one of the focus gestures ("Click" or "Wave"). In my tests, it was quite difficult to get the Kinect to recognize hand raising, but clicking (i.e. pushing) or waving were easily seen. In all cases, the P3 listener is called (to delete the hand trail) followed by the P1 listener which returns a new hand ID and creates a new trail.

This "refocusing" behavior is a bit unfortunate since the user's current hand ID and trail are discarded, and the hand is assigned a new ID and trail. This makes it difficult to track a hand that keeps momentarily disappearing (e.g. behind the user's body or objects in the scene). Of course, it's possible to "hack" together partial solutions by remembering the deleted ID and trail in listener P3 and later 'connect' them to the new ID obtained in listener P1. However, this approach is unreliable when multiple hands are in the scene. It's also possible to record the coordinates of deleted and new hand points, and link old and new IDs by spatial proximity, but it's easy to imagine situations where this might not work correctly.

5. Updates from the Kinect

TrackerPanel starts a thread which waits for the Context object to supply updates to the camera image and/or hand points. The camera picture is converted from a ByteBuffer into a more conventional BufferedImage, and the panel is repainted.

```
// globals
private BufferedImage image = null;
private volatile boolean isRunning;
private long totalTime = 0;
```



```
// OpenNI and NITE vars
private Context context;
private ImageGenerator imageGen;
private SessionManager sessionMan;


public void run()
{
  // wait, update, redraw cycle
  isRunning = true;
  while (isRunning) {
    try {
      context.waitAnyUpdateAll();
      sessionMan.update(context);
    }
    catch(StatusException e)
    {  System.out.println(e);
       System.exit(1);
    }
    long startTime = System.currentTimeMillis();
    updateCameraImage();
    totalTime += (System.currentTimeMillis() - startTime);
    repaint();
  }

  // close down
  try {
    context.stopGeneratingAll();
  }
  catch (StatusException e) {}
  context.release();
  System.exit(1);
}  // end of run()


private void updateCameraImage()
// update Kinect camera's image
{
  try {
    ByteBuffer imageBB = imageGen.getImageMap().createByteBuffer();
    image = bufToImage(imageBB);
    imageCount++;
  }
  catch (GeneralException e) {
    System.out.println(e);
  }
}  // end of updateCameraImage()
```

6. Painting the Panel

Panel repainting involves four elements: the camera picture, the hand trails, a user message, and some statistics.

```
// global
private BufferedImage image = null;
```



```
public void paintComponent(Graphics g)
{
  super.paintComponent(g);
  Graphics2D g2 = (Graphics2D) g;

  if (image != null)
    g2.drawImage(image, 0, 0, this);    // draw camera's image

  drawTrails(g2);
  writeMessage(g2);
  writeStats(g2);
} // end of paintComponent()
```

The hand trails are drawn by iterating through the map and delegating the drawing of each trail to its respective object.

```
// global
private HashMap<Integer, HandTrail> handTrails;


private void drawTrails(Graphics2D g2)
{
  HandTrail handTrail;
  for (Integer id : handTrails.keySet()) {
    handTrail = handTrails.get(id);
    handTrail.draw(g2);
  }
}  // end of drawTrails()
```

The choice of message depends on the current session state and the size of the handTrails map.

```
// globals
private SessionState sessionState;
private HashMap<Integer, HandTrail> handTrails;


private void writeMessage(Graphics2D g2)
{
  g2.setColor(Color.YELLOW);
  g2.setFont(msgFont);

  String msg = null;
  switch (sessionState) {
    case IN_SESSION:
      if (handTrails.size() == 1)
        msg = "Bring your second hand close to
                                    your first to track it";
      else
        msg = "Tracking " + handTrails.size() + " hands...";
      break;
    case NOT_IN_SESSION:
      msg = "Click/Wave to start tracking";
      break;
    case QUICK_REFOCUS:
      msg = "Click/Wave/Raise your hand to resume tracking";
      break;
  }
```



```
  if (msg != null)
    g2.drawString(msg, 5, 20);  // top left
}  // end of writeMessage()
```

The different messages give users some hints about how to trigger session state changes with focus and refocus gestures.

The map size information allows me to include a message about how to move beyond a single hand trail. NITE's focus/refocus gestures only work with the first hand point. Subsequent hand points are recognized differently, by bringing a 'new' hand close to the first hand. The relevant message is shown at the top-left of Figure 4 ("Bring your second hand close to your first to track it").

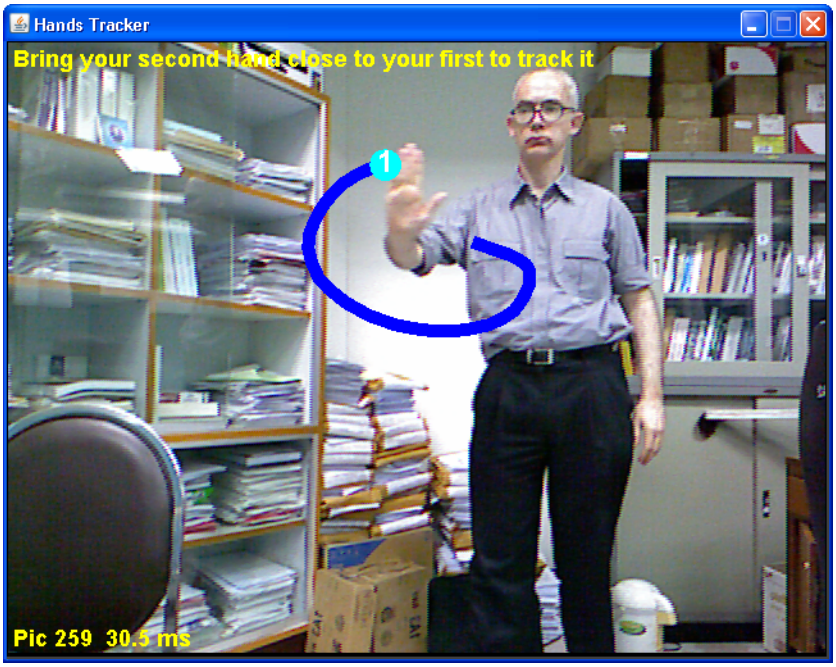

Figure 4. Tracking One Hand.

However, multiple hand tracking will only work if the Nite.ini configuration file has been modified, as explained at the start of this chapter.

7. Storing a Hand Trail

Each hand trail is managed by a HandTrail object, which draws a thick colored line based on the hand coordinates it stores. The trail nearest the user's hand ends in a larger circle with the hand ID written inside it.

A trail is limited to a maximum number of points (currently 30), and when this limit is reached, then the addition of a new point (the user's current hand position) causes the deletion of the oldest point (the one at the end of the trail).

7.1. Adding a Point

The Point3D data passed to HandTrail.addPoint() utilizes real-world values, which must be converted to screen coordinates. The easiest way of doing this is with the DepthGenerator object, which is passed to HandTrail in its constructor.

```
// globals
private static final int MAX_POINTS = 30;

private DepthGenerator depthGen;
private ArrayList<Point> coords;   // points that form the trail


public synchronized void addPoint(Point3D realPt)
// add hand coordinate to the coords list
{
  try {
    Point3D pt = depthGen.convertRealWorldToProjective(realPt);
          // convert real-world coordinates to screen form
    if (pt == null)
      return;
    coords.add( new Point((int)pt.getX(), (int)pt.getY()));
           // convert x and y to ints, and discard z coord
    if (coords.size() > MAX_POINTS) // get rid of the oldest point
      coords.remove(0);
  }
  catch (StatusException e)
  {  System.out.println("Problem converting point"); }
}  // end of addPoint()
```

Only the (x, y) parts of the screen coordinate are stored, although the z-value could be used to vary the thickness of the trail line in a fancier version of this class.

addPoint() is synchronized since it may be called by the TrackerPanel thread at the same time that Java's rendering thread is calling HandTrail.draw() to draw the trail.

7.2. Drawing the Trail

draw() passes the work of drawing the trail to drawTrail(), but places a larger circle and hand ID number at the trail's head.

```
// globals
private static final int MAX_POINTS = 30;

private static final int CIRCLE_SIZE = 25;
private static final int STROKE_SIZE = 10;

private static final Color POINT_COLORS[] = {
  Color.RED, Color.BLUE, Color.CYAN, Color.GREEN,
  Color.MAGENTA, Color.PINK, Color.YELLOW };

private Font msgFont;    // for the hand ID string
private int handID;
private ArrayList<Point> coords;  // points that form the trail


public synchronized void draw(Graphics2D g2)
/* draw trail and large circle on hand (the last point) with the
   hand ID in the center */
```



```
{
  int numPoints = coords.size();
  if (numPoints == 0)
    return;

  drawTrail(g2, coords, numPoints);

  // draw large circle on hand (the last point)
  Point pt = coords.get(numPoints-1);
  g2.setColor(POINT_COLORS[(handID+1) % POINT_COLORS.length]);
  g2.fillOval(pt.x-CIRCLE_SIZE/2, pt.y-CIRCLE_SIZE/2,
                              CIRCLE_SIZE, CIRCLE_SIZE);
  // draw the hand ID
  g2.setColor(Color.WHITE);
  g2.setFont(msgFont);
  g2.drawString("" + handID, pt.x-6, pt.y+6);   // roughly centered
}  // end of draw()



private void drawTrail(Graphics2D g2, ArrayList<Point> coords,
                                              int numPoints)
// draw (x,y) Points list as a polyline (trail)
{
  int[] xCoords = new int[numPoints];
  int[] yCoords = new int[numPoints];

  // fill the integer arrays with x and y points
  Point pt;
  for (int i=0; i < numPoints; i++) {
    pt = coords.get(i);
    xCoords[i] = pt.x;
    yCoords[i] = pt.y;
  }

  g2.setColor(POINT_COLORS[handID % POINT_COLORS.length]);
  g2.setStroke(new BasicStroke(STROKE_SIZE));
  g2.drawPolyline(xCoords, yCoords, numPoints);
}  // end of drawTrail()
```

POINT_COLOR[] is utilized to draw each trail in a different color, based on the hand ID. The ID is also employed to vary the color of the large circle at the head of the trail.

The trail is drawn as a continuous polyline by joining up the points in the coords ArrayList. Another approach would be to draw the trail as a series of dots, one for each point.

