Kinect Chapter 7. NITE Hand 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.]

This chapter is about utilizing OpenNI's gesture and hands generators with NITE's hand gesture detectors. I'll be looking at how to create wave, push, swipe, circle, and steady detectors, and a hand point listener.

The application, GestureDetect.java, doesn't have a graphical interface; instead it prints user gesture information to the command line, as shown below:

```
Started context generating...
Wave settings -- no. of flips: 4; min length: 50mm
Push settings -- min velocity: 0.3 m/s; min duration: 240.0 ms;
                 max angle to z-axis: 30.0 degs
Swipe setting -- min motion time: 350 ms
Circle setting -- min-max radius: 40.0 - 1200.0 mm
Steady settings -- min duration: 250 ms
                   max movement: 0.010 mm

Make a click gesture to start the session
Gesture "Click" recognized at (-144, 336, 1665);
                     ended at (-143, 336, 1670)
Session started at (-144, 336, 1665)
Hand 1 located at (-143, 336, 1670), at 3 secs
Hand Point 1 at (-143, 336, 1670) at 3 secs
Setting resolution to QQVGA
Push: velocity 0.6 m/s, angle 28.7 degs
  Hand Point 1 at (-192, 236, 1699) at 4 secs
Swipe UP: velocity 0.4 m/s, angle 3.9 degs
  Hand Point 1 at (-143, 434, 1585) at 4 secs
Hand 1 is steady: movement 0.009
  Hand Point 1 at (-200, 219, 1703) at 8 secs
Swipe RIGHT: velocity 0.5 m/s, angle 2.2 degs
  Hand Point 1 at (130, 238, 1741) at 8 secs
Hand 1 is steady: movement 0.000
  Hand Point 1 at (-96, 256, 1704) at 9 secs
Wave detected
  Hand Point 1 at (-170, 319, 1727) at 13 secs
Wave detected
  Hand Point 1 at (-31, 306, 1734) at 14 secs
Push: velocity 0.8 m/s, angle 18.8 degs
  Hand Point 1 at (-114, 303, 1640) at 15 secs
    :
```

GestureDetect begins by initializing five gesture detectors (wave, push, swipe, circle, and steady) and a hands point listener, and printing some of the detectors' default settings. After the user prompt, "Make a click gesture…", the program enters a "focus detection" phase where it tries to identify a hand by having the user perform a "click"

gesture. If this is successful recognized, then the application starts a "gesture tracking" session where the user's wave, push, swipe, circle, and steady gestures are reported.

Actually, the program never reports on circle gestures, due to a minor error in the NITE API, as explained in section 10.

1. Focus Detection

During focus detection, the Kinect waits for the user to perform a "click" focus gesture which lets NITE identify the person's hand and start a gesture tracking session. It's also possible to configure NITE to look for a "wave" focus gesture.

A "click" involves the user holding the palm of their hand towards the Kinect, then straightening their arm. Focus detection is more likely to succeed by having the user follow a few rules:

> The user's hand should be kept away from their body and other objects, but within the Kinect's field-of-view.
>
> The palm of the user's hand should be open, facing the sensor, with their fingers pointing upwards.
>
> The user should try to stand around 2m from the sensor.

If the Kinect loses touch with a hand during gesture tracking, the session enters a "quick refocus" state, which tries to relocate the hand before a timeout terminates the session. The quick refocus gesture typically involves the user raising their hand.

2. Gesture Tracking

NITE gestures are derived from a stream of hand points which record how a hand moves through space over time. Each hand point is the real-world 3D coordinate of the center of the hand, measured in millimeters.

Gesture detectors are sometimes called point listeners (or point controls) since they analyze the points stream looking for a gesture. Currently all the detectors (except SteadyDetector) only analyze the hand that performed the focus or refocus gesture. In that case, the hand point is called the *primary* point.

If you're happy to do without gesture interpretation, and instead manipulate point streams directly, then it is possible to track multiple hands at the same time (as I'll show in chapter 8).

The superclass of all the gesture tracking classes is PointControl (see Figure 1), a class for watching a stream of hand points.

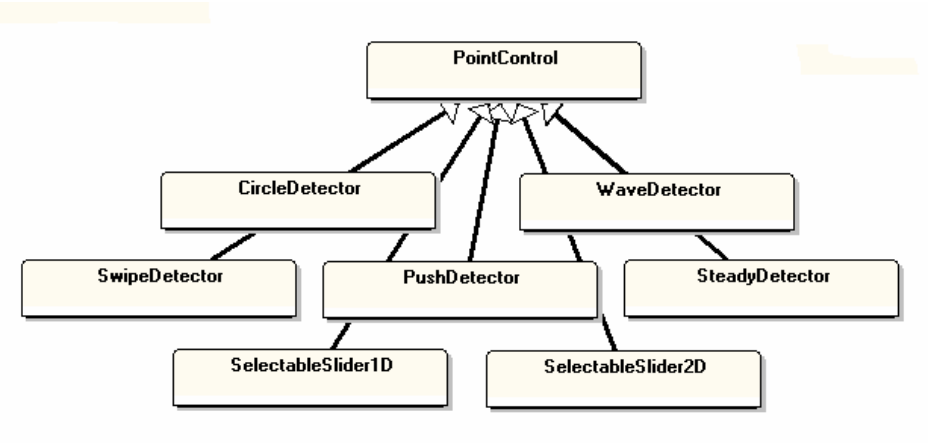

Figure 1. PointControl and its Gesture Detector Subclasses.

This chapter will use the following gesture detectors:

WaveDetector interprets hand movement as left-right waving, carried out at least four times in a row.

PushDetector recognizes hand movement as pushes towards and away from the sensor. Pushes are the same as focus detection clicks, but occur during tracking.

SwipeDetector looks for swiped hand movement, either up, down, left or right, followed by the hand resting momentarily.

CircleDetector sees hand movement as circular motion. It needs to observe a full circle in either direction before identifying the gesture. Clockwise rotation is positive, anti-clockwise negative.

SteadyDetector recognizes a hand that hasn't moved for some time

I won't be utilizing the SelectableSlider1D or SelectableSlider2D detectors in this chapter.

SelectableSlider1D treats hand movement as adjustments to an invisible slider, aligned to the x-, y-, or z- axes. The slider can include selectable areas, which allow it to be used to implement menus, with each area acting as a menu item. The NITE sample, Boxes.java, utilizes SelectableSlider1D.

SelectableSlider2D treats hand behavior as (x, y) movement across a predefined spatial area. It's possible to define multiple areas, spread along the z-axis.

3. Detecting Gestures

My GestureDetector program starts by setting up a range of OpenNI nodes and NITE detectors, each with event listeners. Then it enters a loop which passes Kinect events arriving at OpenNI's context object to NITE's SessionManager.

```
// globals
private Context context;
private SessionManager sessionMan;
```



```
private boolean isRunning = true;


public GestureDetect()
{
  try {
    configOpenNI();    // initialize OpenNI and NITE
    configNITE();

    System.out.println();
    System.out.println("Make a click gesture to start the session");
    while (isRunning) {
      context.waitAnyUpdateAll();
      sessionMan.update(context);
    }
    context.release();
  }
  catch (GeneralException e) {
    e.printStackTrace();
  }
}  // end of GestureDetect()
```

The global Context and SessionManager objects are created in configOpenNI() and configNITE() respectively, as shown below.

4. Initializing OpenNI

configOpenNI() initializes the OpenNI parts of the gesture detector by creating HandsGenerator and GestureGenerator objects:

```
// globals
private Context context;


private void configOpenNI()
// set up the Gesture and Hands Generators in OpenNI
{
  try {
    context = new Context();

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

    HandsGenerator handsGen = HandsGenerator.create(context);
    handsGen.SetSmoothing(0.1f);
             // 0-1: 0 means no smoothing, 1 means 'infinite'
    setHandEvents(handsGen);

    GestureGenerator gestureGen = GestureGenerator.create(context);
    setGestureEvents(gestureGen);

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



```
  catch (GeneralException e) {
    e.printStackTrace();
    System.exit(1);
  }
}  // end of configOpenNI()
```

HandsGenerator supports hand detection and tracking, sending events to user-defined listeners when a hand point is first detected, moves, and is finally destroyed.

GestureGenerator sends events to its listeners during focus detection. However, during a gesture tracking session, gesture processing is handled by NITE detectors with their own callback methods.

4.1. Processing Hand Events

As you'll see, my GestureDetect application consists of many listeners for different kinds of gesture events. The good news is that the listeners all look very similar since they follow the same coding style as in chapter 4 (where I used listeners for user tracking).

A typical listener class implements the generic IObserver interface, instantiated for a particular kind of event (represented by a subclass of EventArg). The class contains a single update() method, which is called whenever an event arrives, and utilizes get methods to access the event's information:

```
class Foo implements IObserver<EventArgs-subclass>
{
  public void update(IObservable<EventArgs-subclass> observable,
                     EventArgs-subclass args)
  {
    int id = args.getId();
    // access other args information
    try {
      // do something with information; usually just print it out
    }
    catch (StatusException e)
    {  e.printStackTrace(); }
  }
}  // end of Foo class
```

In OpenNI there are twelve EventArg subclasses, but NITE has this beat with sixteen! Documentation on the classes is rather scanty, and so I employed the JD-GUI decompiler (http://java.decompiler.free.fr/?q=jdgui) to examine the decompiled OpenNI and NITE Jar files at C:\Program Files\OpenNI\Bin\org.OpenNI.jar and C:\Program Files\PrimeSense\NITE\Bin\ com.primesense.NITE.jar.

The HandsGenerator class supports three listeners, which is easiest to see by looking at its decompiled code. The class includes three private listener variables:

```
// in HandsGenerator
private Observable<ActiveHandEventArgs> handCreateEvent;
private Observable<ActiveHandEventArgs> handUpdateEvent;
```



```
private Observable<InactiveHandEventArgs> handDestroyEvent;
```

These can be accessed outside the class via corresponding get methods:

```
// in HandsGenerator
public IObservable<ActiveHandEventArgs> getHandCreateEvent()
{  return handCreateEvent;}

public IObservable<ActiveHandEventArgs> getHandUpdateEvent()
{  return handUpdateEvent  }

public IObservable<InactiveHandEventArgs> getHandDestroyEvent()
{  return handDestroyEvent;  }
```

This coding style is used consistently across OpenNI and NITE.

It's necessary to look at the EventArgs subclass (in this case, ActiveHandEventArgs and InactiveHandEventArgs) to see what information they manage, and how to call their get methods. For example, ActiveHandEventArgs contains three suitable methods: getId(), getPosition(), and getTime().

Another good way of finding out about a class is to search the OpenNI documentation (a CHM file) for the C++ API in its Class Lists section. There's a close relationship between the C++ and Java versions of most of the classes in OpenNI (and NITE). The document usually explains the purpose of class methods, information lacking from the decompiled Java source.

My GestureDetect.setHandEvents() method implements two anonymous listeners for when a hand is first identified (a so-called hand creation event) and when a hand disappears (a hand destroy event):

```
private void setHandEvents(HandsGenerator handsGen)
{
  try {
    // when a hand is first identified
    handsGen.getHandCreateEvent().addObserver(
                        new IObserver<ActiveHandEventArgs>() {
      public void update(
              IObservable<ActiveHandEventArgs> observable,
              ActiveHandEventArgs args)
      { int id = args.getId();
        Point3D pt = args.getPosition();
        float time = args.getTime();
        System.out.printf("Hand %d located at
           (%.0f, %.0f, %.0f), at %.0f secs\n",
            id, pt.getX(), pt.getY(), pt.getZ(), time);
      }
    });

    // when a hand is lost to tracking
    handsGen.getHandDestroyEvent().addObserver(
                        new IObserver<InactiveHandEventArgs>() {
      public void update(
                  IObservable<InactiveHandEventArgs> observable,
                  InactiveHandEventArgs args)
      { int id = args.getId();
        float time = args.getTime();
```



```
        System.out.printf("Hand %d destroyed at %.0f secs \n",
                                                  id, time);
      }
    });
  }
  catch (StatusException e) {
    e.printStackTrace();
  }
}  // end of setHandEvents()
```

Hand creation is reported as a "Hand %d located…" message, which can be seen in the listing at the start of this chapter:

```
Hand 1 located at (-143, 336, 1670), at 3 secs
```

The point uses real-world coordinates, measured in millimeters, with the positive x-, y-, and z- axes as shown in Figure 2.

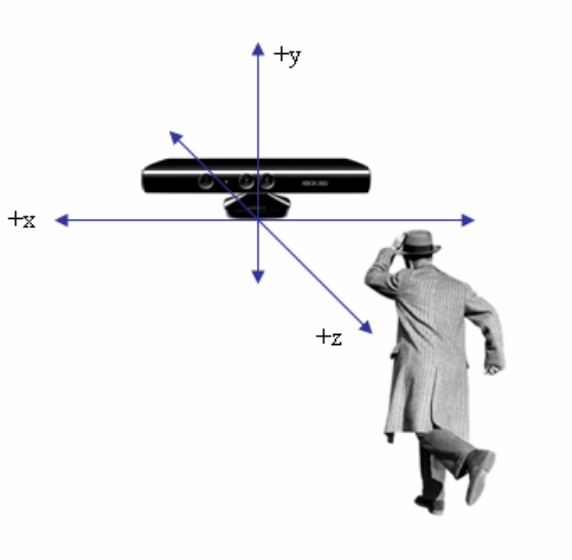

Figure 2. Hand Point Coordinate System.

4.2. Processing Gesture Focus Events

GestureGenerator supports five listeners, represented in the decompiled GestureGenerator class by five private variables:

```
private Observable<GestureRecognizedEventArgs>
                                          gestureRecognizedEvent;
private Observable<GestureProgressEventArgs> gestureProgressEvent;
private Observable<GesturePositionEventArgs>
                            gestureIntermediateStageCompletedEvent;
private Observable<GesturePositionEventArgs>
                         gestureReadyForNextIntermediateStageEvent;
private StateChangedObservable gestureChangedEvent;
```

Useful information is also available in the OpenNI documentation for the C++ version of GestureGenerator. Strangely, there's no C++ equivalent for the last listener, gestureChangedEvent.

The most useful listener is probably the first, gestureRecognizedEvent, which is fired when a gesture is recognized. The generated event is a subclass of GestureRecognizedEventArgs which supports three get methods: getGesture(), getIdPosition(), and getEndPosition(). A simple implementation of a suitable listener appears in my setGestureEvents() method:

```
private void setGestureEvents(GestureGenerator gestureGen)
{
  try {
    // callback for gesture focus
    gestureGen.getGestureRecognizedEvent().addObserver(
                    new IObserver<GestureRecognizedEventArgs>() {
      public void update(
             IObservable<GestureRecognizedEventArgs> observable,
             GestureRecognizedEventArgs args)
      {
        String gestureName = args.getGesture();
        Point3D idPt = args.getIdPosition();
           // hand position when gesture was identified
        Point3D endPt = args.getEndPosition();
            // hand position at the end of the gesture

        System.out.printf("Gesture \"%s\" recognized at
             (%.0f, %.0f, %.0f); ended at (%.0f, %.0f, %.0f)\n",
            gestureName, idPt.getX(), idPt.getY(), idPt.getZ(),
                        endPt.getX(), endPt.getY(), endPt.getZ() );
      }
    });
  }
  catch (StatusException e) {
    e.printStackTrace();
  }
}  // end of setGestureEvents()
```

OpenNI's GesturesGenerator class is not that useful since its gesture listener is only called for focus events. I really want data on gestures detected during the tracking session, which are handled by NITE detector classes.

When the gesture focus is obtained, typically output from GestureGenerator's listener is:

```
Gesture "Click" recognized at (-75, 304, 1506);
                     ended at (-74, 304, 1511)
```

This message appears immediately after a hand has been identified by HandsGenerator. A gesture tracking session, handled by NITE, can now commence.

5. Initializing NITE

My GestureDetect.configNITE() method creates a SessionManager object to supply a stream of hand points, and attaches several NITE detectors to it.

```
// globals
private Context context;
private SessionManager sessionMan;
```



```
private void configNITE()
{
  try {
    sessionMan = new SessionManager(context, "Click", "RaiseHand");
                            // focus gesture(s), refocus gesture(s)
    setSessionEvents(sessionMan);

    // point, wave, push, swipe, circle, steady NITE detectors;
    // connect them to the session manager
    PointControl pointCtrl = initPointControl();
    sessionMan.addListener(pointCtrl);

    WaveDetector wd = initWaveDetector();
    sessionMan.addListener(wd);

    PushDetector pd = initPushDetector();
    sessionMan.addListener(pd);

    SwipeDetector sd  = initSwipeDetector();
    sessionMan.addListener(sd);

    CircleDetector cd = initCircleDetector();
    sessionMan.addListener(cd);

    SteadyDetector sdd  = initSteadyDetector();
    sessionMan.addListener(sdd);
  }
  catch (GeneralException e) {
    e.printStackTrace();
    System.exit(1);
  }
}  // end of configNITE()
```

The SessionManager's input arguments specify the focus and refocus gestures: a "click" initiates a tracking session, while "RaiseHand" is used to resume a stalled session.

configNITE() creates six detectors for processing the hand points output by the SessionManager, as shown in Figure 3.

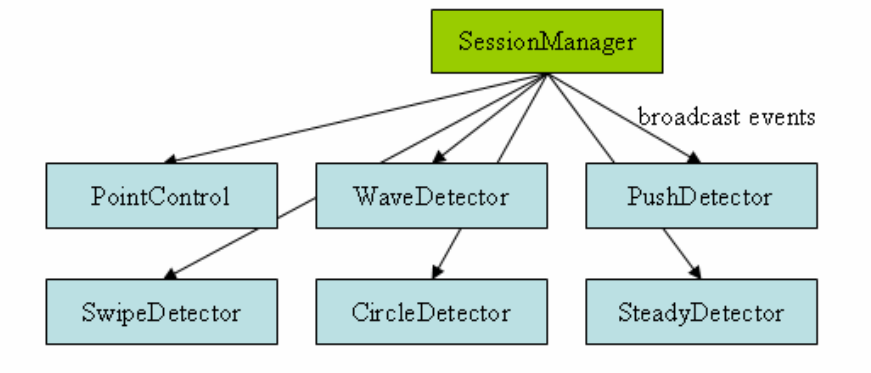

Figure 3. SessionManager and its Detectors.

SessionManager.addListener() connects the detectors to the manager so that each one receives the broadcast hand points.

For more complicated routing, NITE includes the FlowRouter and Broadcaster classes. FlowRouter acts as a switch so that data can be routed to one detector rather than another. The choice of node can be changed at run time, thereby allowing an application to switch between detectors. The NITE Boxes.java sample uses FlowRouter to switch between PointControls and a SelectableSlider1D.

The Broadcaster class may seem somewhat superfluous, but is useful as a way of grouping detectors together as a single switch node beneath a FlowRouter.

5.1. Processing Session Events

SessionManager supports listeners due to its subclassing of SessionGenerator. The decompiled SessionGenerator class contains three listener variables:

```
private Observable<PointEventArgs> sessionStartEvent;
private Observable<NullEventArgs> sessionEndEvent;
private Observable<StringPointValueEventArgs>
                                    sessionFocusProgressEvent;
```

This information is also available in the NITE API documentation for the C++ XnVSessionGenerator class.

As you might expect, a sessionStartEvent listener is woken when a gesture tracking session starts, and sessionEndEvent is notified when the session ends.

A sessionFocusProgress event may be fired during gesture focusing, before the session's start. Part of the event is a progress variable, a value between 0 and 1, which indicates how close the focusing is to finishing. It seems that a sessionFocusProgress event isn't always generated, especially if the focus is obtained quickly.

gestureDetect.setSessionEvents() implements listeners for the three types of session event:

```
// global
private boolean isRunning = true;


private void setSessionEvents(SessionManager sessionMan)
{
  try {
    // processing of focus gesture is in progress
    sessionMan.getSessionFocusProgressEvent().addObserver(
                        new IObserver<StringPointValueEventArgs>()
      public void update(
            IObservable<StringPointValueEventArgs> observable,
            StringPointValueEventArgs args)
      {
        Point3D focusPt = args.getPoint();
        float progress = args.getValue();
        String focusName = args.getName();
        System.out.printf("Session focused at
            (%.0f, %.0f, %.0f) on %s [progress %.2f]\n",
              focusPt.getX(), focusPt.getY(), focusPt.getZ(),
```



```
                                        focusName, progress);
      }
    });

    // session started
    sessionMan.getSessionStartEvent().addObserver(
                                new IObserver<PointEventArgs>() {
      public void update(
                  IObservable<PointEventArgs> observable,
                  PointEventArgs args)
      { Point3D focusPt = args.getPoint();
        System.out.printf("Session started at
                        (%.0f, %.0f, %.0f)\n",
                focusPt.getX(), focusPt.getY(), focusPt.getZ());
      }
    });


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

The sessionEndEvent listener (the last of the anonymous listeners in setSessionEvents()) sets a global boolean isRunning to false. This causes the processing loop back in GestureDetect's constructor to stop (see section 3), and the program terminates.

6. Detecting Hand Points

GestureDetect's use of a PointControl object may seem a bit unusual because the gesture detectors subclass PointControl and so contain all of its functionality. Why bother with a PointControl listener, when I can implement the same code in one of the detectors?

One reason is to have a way of monitoring hand positions separately from gesture processing. This gives me the option of carrying out additional forms of processing on the hand points stream. This stream isn't available to a detector, such as a PushDetector, which only processes hand data related to its gesture.

In fact, I won't be doing anything fancy with the hand positions stream in this chapter, but chapter 8 utilizes PointControl listeners to visualize multiple hand tracks.

The decompiled PointControl class contains eight listener variables:


```
private Observable<HandEventArgs> pointCreateEvent;
private Observable<HandEventArgs> pointUpdateEvent;
private Observable<IdEventArgs> pointDestroyEvent;

private Observable<HandPointEventArgs> primaryPointCreateEvent;
private Observable<HandEventArgs> primaryPointUpdateEvent;
private Observable<IdEventArgs> primaryPointDestroyEvent;
private Observable<HandIdEventArgs> primaryPointReplaceEvent;

private Observable<NullEventArgs> noPointsEvent;
```

Four of the listeners refer to the primary point, which is associated with the hand that performed the focus gesture There can be other hand points, linked to other hands in the Kinect's field of view, but they won't trigger the primary point listeners.

If the primary point is unavailable (e.g. because its hand has disappeared from view), then one of the other hand points will take over as the primary point. This will trigger a primaryPointReplace event.

All the hand points, including the primary point, are monitored by the pointCreateEvent, pointUpdateEvent. pointDestroyEvent listeners, and so I won't need primary point listeners in my code.

initPointControl() sets up three listeners for hand point creation, updates (i.e. hand movement), and point destruction (i.e. hand disappearance):

```
// global
private PositionInfo pi = null;  // for current hand point info


private PointControl initPointControl()
{
  PointControl pointCtrl = null;
  try {
    pointCtrl = new PointControl();

    // create new hand point
    pointCtrl.getPointCreateEvent().addObserver(
                 new IObserver<HandEventArgs>() {
      public void update(
                    IObservable<HandEventArgs> observable,
                    HandEventArgs args)
      { pi = new PositionInfo( args.getHand() );
        System.out.println(pi);
      }
    });


    // hand point has moved
    pointCtrl.getPointUpdateEvent().addObserver(
                new IObserver<HandEventArgs>() {
      public void update(
                  IObservable<HandEventArgs> observable,
                  HandEventArgs args)
      { HandPointContext handContext = args.getHand();
        if (pi == null)
          pi = new PositionInfo(handContext);
        else
```



```
          pi.update(handContext);
        // System.out.println(pi); //comment out to reduce output
      }
    });

    // destroy hand point
    pointCtrl.getPointDestroyEvent().addObserver(
                new IObserver<IdEventArgs>() {
      public void update(
                    IObservable<IdEventArgs> observable,
                    IdEventArgs args)
      { int id = args.getId();
        System.out.printf("Point %d destroyed:\n", id);
        if (pi.getID() == id)
          pi = null;
      }
    });

  }
  catch (GeneralException e) {
    e.printStackTrace();
  }
  return pointCtrl;
}  // end of initPointControl()
```

These point control listeners create and manipulate a global PositionInfo object, which I'll utilize to supply hand position information to the other detectors.

PositionInfo stores the latest data on a hand point : the ID of its hand, its (x, y, z) coordinate, and the time when the point was observed.

```
public class PositionInfo
{
  private int id;         // of the hand
  private Point3D pos;    // in real-world coords (mm)
  private float time;     // in secs


  public PositionInfo(HandPointContext hpc)
  {
    id = hpc.getID();
    pos = hpc.getPosition();
    time = hpc.getTime();
  }  // end of PositionInfo()


  public synchronized void update(HandPointContext hpc)
  {
    if (id == hpc.getID()) {
      pos = hpc.getPosition();
      time = hpc.getTime();
    }
  }  // end of update()


  public int getID()  //no need to synch since ID doesn't change
  {  return id;  }


  public synchronized Point3D getPosition()
```



```
  {  return pos;  }

  public synchronized float getTime()
  {  return time;  }


  public synchronized String toString()
  {
    return String.format("Hand Point %d at (%.0f, %.0f, %.0f)
                                          at %.0f secs",
              id, pos.getX(), pos.getY(), pos.getZ(), time);
  }  // end of toString()

}  // end of PositionInfo class
```

The get and set methods are synchronized because PositionInfo is updated by the PointControl listener, which may be running in a separate thread from a detector listeners reading the PositionInfo.

7. The Wave Detector

A wave detector tries to interpret hand point movement as left-right waving, consisting of a number of direction changes (flips) of a certain length, carried out within a fixed time period. It has one listener variable (in addition to those inherited from PointControl):

```
private Observable<NullEventArgs> waveEvent;
```

initWaveDetector() and the other detector initialization methods in the following sections have much the same structure. First a detector object is created, some of its parameters are printed, then a listener (or listeners) is attached to the detector.

initWaveDetector() creates a WaveDetector object, prints two default settings, and specifies a listener for the wave event.

```
// global
private PositionInfo pi = null;
          // for storing current hand point info


private WaveDetector initWaveDetector()
{
  WaveDetector waveDetector = null;
  try {
    waveDetector = new WaveDetector();

    // print 2 wave settings
    int flipCount = waveDetector.getFlipCount();
    int flipLen = waveDetector.getMinLength();
    System.out.println("Wave settings -- no. of flips: " +
             flipCount + "; min length: " + flipLen + "mm");

    // create callback for when a wave is detected
    waveDetector.getWaveEvent().addObserver(
                        new IObserver<NullEventArgs>() {
      public void update(
```



```
                  IObservable<NullEventArgs> observable,
                  NullEventArgs args)
      {
        System.out.println("Wave detected");
        System.out.println("  " + pi);  // show current hand point
      }
    });
  }
  catch (GeneralException e) {
    e.printStackTrace();
  }
  return waveDetector;
}  // end of initWaveDetector()
```

initWaveDetector() prints out the default flip count and minimum length of a wave, which appear as:

```
Wave settings -- no. of flips: 4; min length: 50mm
```

WaveDetector has other setting, such as the maximum permitted angular deviation of the wave from the horizontal plane, before the movement stops being interpreted as a wave.

initWaveDetector() only reports the parameters; there are also WaveDetector set methods for changing their values.

The listener prints out a rather uninformative "Wave detected" message, but supplements it with the current hand position, obtained from the global PositionInfo object:

```
Wave detected
  Hand Point 1 at (-164, 302, 1635) at 10 secs
```

8. The Push Detector

PushDetector interprets a moving hand point as a push when the point reaches a specified velocity at an angle close to the Kinect's z-axis for a certain period of time. Not surprisingly, PushDetector's parameters include settings for the minimum velocity, maximum angle, and push duration. The class also has methods to compare the current push's velocity, angle, and duration with the previous push. There are two listener variables:

```
private Observable<VelocityAngleEventArgs> pushEvent;
private Observable<ValueEventArgs> stabilizedEvent;
```

The first listener detects a push event, while the second manages a stabilized event, which occurs when the hand point stops moving at the end of a push. A 'new' push can't be detected until a stabilized event for the current push has been detected.

initPushDetector() creates a PushDetector object, prints three settings, and sets up a single listener for detecting pushes:

```
// global
private PositionInfo pi = null;
```



```
private PushDetector initPushDetector()
{
  PushDetector pushDetector = null;
  try {
    pushDetector = new PushDetector();

    // print 3 push settings
    float minVel = pushDetector.getPushImmediateMinimumVelocity();
          // min push speed, in m/s

    float duration = pushDetector.getPushImmediateDuration();
          // min time used to detect a push, in ms

    float angleZ =
         pushDetector.getPushMaximumAngleBetweenImmediateAndZ();
         // max angle between direction and z-axis, in degrees

    System.out.printf("Push settings -- min velocity: %.1f m/s;
          min duration: %.1f ms; max angle to z-axis: %.1f degs \n",
                      minVel, duration, angleZ);

    // callback for push detection
    pushDetector.getPushEvent().addObserver(
               new IObserver<VelocityAngleEventArgs>() {
      public void update(
               IObservable<VelocityAngleEventArgs> observable,
               VelocityAngleEventArgs args)
      { System.out.printf("Push: velocity %.1f m/s,
                                    angle %.1f degs \n",
                     args.getVelocity(), args.getAngle());
        System.out.println("  " + pi);  // show current hand point
      }
    });
  }
  catch (GeneralException e) {
    e.printStackTrace();
  }
  return pushDetector;
}  // end of initPushDetector()
```

initPushDetector () prints out the push's minimum velocity, maximum angle, and duration:

```
Push settings -- min velocity: 0.3 m/s; min duration: 240.0 ms;
                 max angle to z-axis: 30.0 degs
```

The listener reports the push's velocity and angle to the z-axis, and the current hand position obtained from the global PositionInfo object:

```
Push: velocity 0.5 m/s, angle 19.6 degs
  Hand Point 2 at (-78, 663, 1698) at 13 secs
```


9. The Swipe Detector

A swipe motion is a hand movement up, down, left or right, followed by a short pause. SwipeDetector's parameters include the speed threshold for the gesture to be deemed a swipe, its minimum duration, and the maximum permitted angular derivations from the x-and y-axes.

The class has five listener variables, one which responds to all swipe directions (up, down, left, right), while the other listeners report on a specific type:

```
private Observable<DirectionVelocityAngleEventArgs> swipeEvent;

private Observable<VelocityAngleEventArgs> swipeUpEvent;
private Observable<VelocityAngleEventArgs> swipeDownEvent;
private Observable<VelocityAngleEventArgs> swipeLeftEvent;
private Observable<VelocityAngleEventArgs> swipeRightEvent;
```

initSwipeDetector() creates a SwipeDetector object, prints the minimum swipe duration, and specifies two listeners – one for capturing all swipe directions, and one only for left swipes.

```
// global
private PositionInfo pi = null;


private SwipeDetector initSwipeDetector()
{
  SwipeDetector swipeDetector = null;
  try {
    swipeDetector = new SwipeDetector();

    // print 1 swipe setting
    System.out.println("Swipe setting -- min motion time: " +
                swipeDetector.getMotionTime() + " ms");

    // general swipe callback
    swipeDetector.getGeneralSwipeEvent().addObserver(
              new IObserver<DirectionVelocityAngleEventArgs>() {
      public void update(
           IObservable<DirectionVelocityAngleEventArgs> observable,
           DirectionVelocityAngleEventArgs args)
      { System.out.printf("Swipe %s: velocity %.1f m/s,
                                          angle %.1f degs \n",
        args.getDirection(), args.getVelocity(), args.getAngle());
        System.out.println("  " + pi);  // show current hand point
      }
    });

    // callback for left swipes only
    swipeDetector.getSwipeLeftEvent().addObserver(
                 new IObserver<VelocityAngleEventArgs>() {
      public void update(
                IObservable<VelocityAngleEventArgs> observable,
                VelocityAngleEventArgs args)
      { System.out.printf("*Left* Swipe: velocity %.1f m/s,
                                    angle %.1f degs \n",
                    args.getVelocity(), args.getAngle());
      }
```



```
    });
  }
  catch (GeneralException e) {
    e.printStackTrace();
  }
  return swipeDetector;
}  // end of initSwipeDetector()
```

The swipe listeners report velocity and angle details, while position data comes from the global PositionInfo object:

```
Swipe UP: velocity 0.5 m/s, angle 15.8 degs
  Hand Point 1 at (-74, 401, 1718) at 9 secs
Swipe DOWN: velocity 0.3 m/s, angle 177.1 degs
  Hand Point 1 at (-99, 368, 1750) at 11 secs
```

and

```
*Left* Swipe: velocity 0.3 m/s, angle 178.9 degs
Swipe LEFT: velocity 0.3 m/s, angle 178.9 degs
  Hand Point 1 at (-285, 272, 1708) at 7 secs
Swipe RIGHT: velocity 0.3 m/s, angle 10.5 degs
  Hand Point 1 at (-169, 231, 1690) at 8 secs
```

Note that a left swipe is reported twice, first by the left swipe listener, then by the general swipe listener.

Figure 4 summarizes how swipe directions are represented as positions and angles along the x- and y- axes.

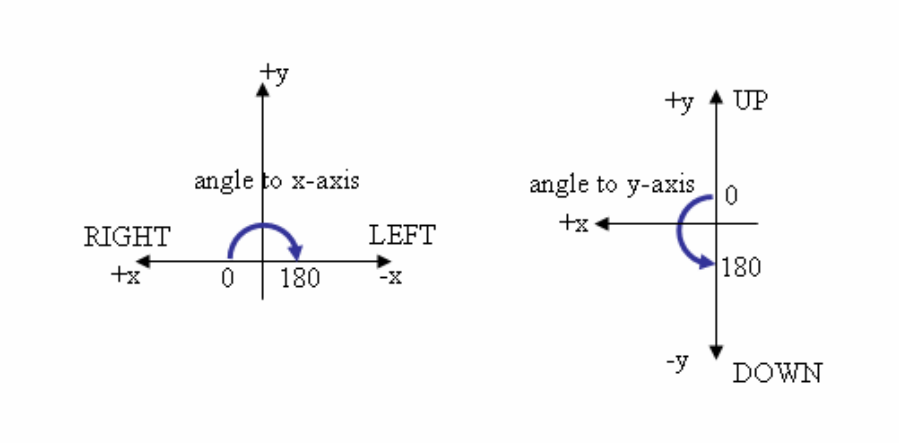

Figure 4. Swipe Positions and Angles.

For example, a left swipe moves along the negative x-axis, and is rotated somewhere close to 180 degrees relative to the +x axis.

10. The Circle Detector

The circle detector tries to connect successive hand points to form a circle. It needs at least enough data to form a full circle in either the clockwise or anti-clockwise

direction. It's parameters include the minimum number of data points it needs, the minimum and maximum radii of a valid circle, and a weight factor which controls how much a new point modifies the circle defined by existing points.

CircleDetector has two listener variables, one for detecting a circle, and one for a non-circle. A non-circle event is triggered when the incoming stream of hand points no longer matches a circle.

```
Observable<CircleEventArgs> circleEvent;
Observable<NoCircleEventArgs> noCircleEvent;
```

initCircleDetector() creates a CircleDetector, prints out the maximum and minimum radii accepted for a circle, and sets up a circle listener.

```
private CircleDetector initCircleDetector()
{
  CircleDetector circleDetector = null;
  try {
    circleDetector = new CircleDetector();

    // print 2 circle settings
    System.out.println("Circle setting -- min-max radius: " +
                circleDetector.getMinRadius() + " - " +
                circleDetector.getMaxRadius() +" mm");

    // callback for detecting a circle
    circleDetector.getCircleEvent().addObserver(
                      new IObserver<CircleEventArgs>() {
      public void update(
                IObservable<CircleEventArgs> observable,
                CircleEventArgs args)
      { Circle circle = args.getCircle();
        Point3D center = circle.getCenter();
        System.out.printf("Circle: center (%.0f, %.0f, %.0f),
                    radius %.0f, times %d\n",
                  center.getX(), center.getY(), center.getZ(),
                  circle.getRadius(), args.getTimes());
      }
    });
  }
  catch (GeneralException e) {
    e.printStackTrace();
  }
  return circleDetector;
}  // end of initCircleDetector()
```

The reported radii are:

```
Circle setting -- min-max radius: 40.0 - 1200.0 mm
```

I couldn't compile the initCircleDetector() listener since CircleDetector.getCircleEvent() is incorrectly declared as private (the same is true for CircleDetector.getNoCircleEvent()). Hopefully these oversights will be fixed in future versions of the API.

11. The Steady Detector

SteadyDetector looks for the absence of hand movement, which might seem a tad useless. However, no movement is a good way to signal the end of a gesture before a new one starts. SteadyDetector's parameters include the minimum time needed for a steady state to be recognized, and the maximum amount of movement permitted before the steady state becomes invalid.

SteadyDetector has two listener variables, one for detecting a steady state, and the other for recognizing the end of the state (i.e. when the hand starts moving again):

```
private Observable<IdValueEventArgs> steadyEvent;
private Observable<IdValueEventArgs> notSteadyEvent;
```

initSteadyDetector() creates a SteadyDetector object, prints two settings, and implements a steady listener. This listener tends to print a lot of messages since most gestures end with a short stationary period.

```
private SteadyDetector initSteadyDetector()
{
  SteadyDetector steadyDetector = null;
  try {
    steadyDetector = new SteadyDetector();

    // print 2 settings
    System.out.println("Steady settings -- min duration: " +
                   steadyDetector.getDetectionDuration() + " ms");
    System.out.printf("          max movement: %.3f mm\n",
                steadyDetector.getMaxDeviationForSteady());

    // callback for steady detection
    steadyDetector.getSteadyEvent().addObserver(
                    new IObserver<IdValueEventArgs>() {
      public void update(
                    IObservable<IdValueEventArgs> observable,
                    IdValueEventArgs args)
      { System.out.printf("Hand %d is steady: movement %.3f\n",
                    args.getId(), args.getValue());
        System.out.println("  " + pi);
      }
    });
  }
  catch (GeneralException e) {
    e.printStackTrace();
  }
  return steadyDetector;
}  // end of initSteadyDetector()
```

The parameter output reports the minimum duration for a steady state, and the maximum amount of movement allowed:

```
Steady settings -- min duration: 250 ms
                   max movement: 0.010 mm
```


These numbers mean that a user only has to pause for a quarter of a second for the steady detector to be fired. The small deviation means that the user's hand must be very still for the detector to be triggered.

Example listener output shows the amount of hand movement and the hand position.:

```
Hand 1 is steady: movement 0.000
  Hand Point 1 at (-96, 256, 1704) at 9 secs
```

12. Filter Objects

I haven't employed NITE hand point filters in GestureDetect. A filter object can be inserted between the SessionManager (or a flow object such as FlowRouter) and a detector to pre-process the stream of hand points before they reach the detector. NITE includes two filters, PointDenoiser and PointArea, which are shown in their class hierarchy in Figure 5.

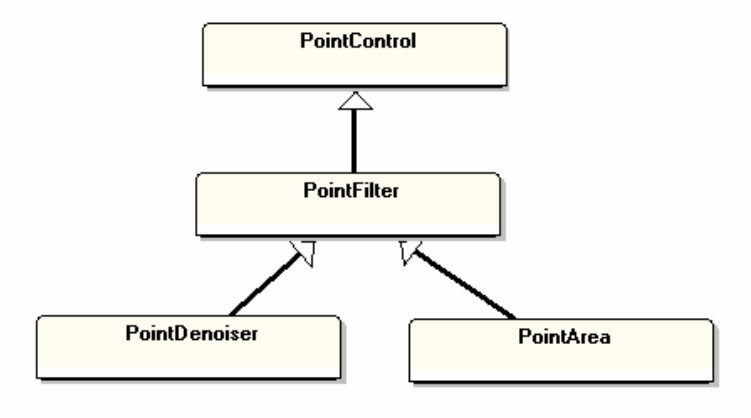

Figure 5. Filter Object Hierarchy in NITE.

PointDenoiser smoothes the stream of hand points, eliminating small movements due to image noise or user inaccuracy.

PointArea applies 3D space cropping to the stream so that only points inside the volume (or outside it) are passed onto the detectors.

