Chapter 14. Using the Leap Motion Controller

A few weeks ago, I finally received my much-anticipated Leap Motion controller (https://www.leapmotion.com/), a device that grants us mere mortals the ability to interact with a computer by waving our hands and fingers. Figure 1 shows my left hand wafting around over the controller (the small metallic and black glass device on the table).

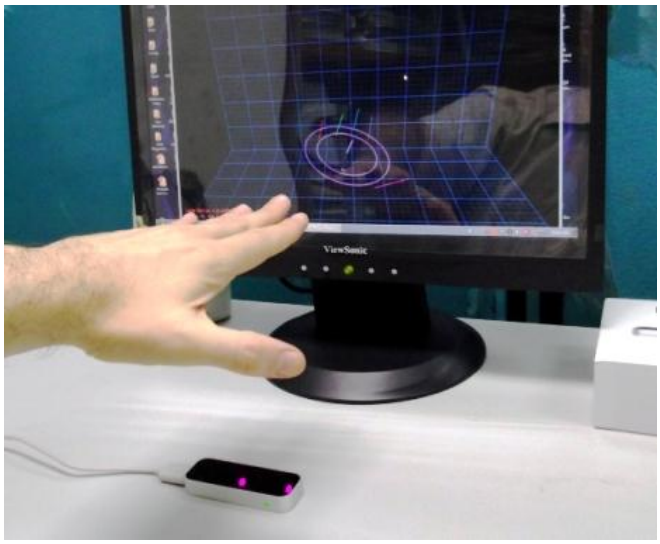

Figure 1. Commanding the Leap.

A slew of user reviews have appeared, including from *The New York Times* (http://www.nytimes.com/2013/07/25/technology/personaltech/no-keyboard-and-now-no-touch-screen-either.html), *Engadget* (http://www.engadget.com/2013/07/22/leap-motion-controller-review/), and *TechCrunch* (http://techcrunch.com/2013/07/22/leap-motion-launches-with-limited-appeal-but-it-could-be-a-ticking-time-bomb-of-innovation/). I tend to agree with their conclusions that the device is tricky and tiring to use. But it's early days for such an innovative idea, and the technology may be a more natural match for hand-held devices such as smartphones and tablets. It also nicely complements the Kinect sensor, with the Kinect handling whole-body tracking across meters while the Leap is fine-tuned for hand and finger tracking and gestures, accurate to millimeters.

The main thrust of this chapter is how to program with the Leap using its Java API (there are also libraries for Python, C++, C#, Objective C, and JavaScript). Free registration at the Leap Motion developer website (https://developer.leapmotion.com/) allows you to download various goodies such as the Leap SDK, API documentation, technical overviews, and access forums (which are also archived at https://forums.leapmotion.com/archive/).

I'll describe three applications in this chapter:

1. A GUI for a slightly modified version of the SDK's Java example. It displays a large amount of rapidly changing data about detected hands, fingers, and gestures. The application, called LeapViewer, is shown in Figure 2.

2. A 2D canvas which illustrates how to use hands, fingers, and the finger twirling gesture to move, draw and delete colored dots. A screenshot of LeapDoodle is included in Figure 2.
3. A hand-controlled 3D model (the robot in Figure 2), which can glide across a checkerboard floor and rotate around its vertical axis. This example is a simplification of a Java 3D application described at length in the online chapter "3D Sprites" at http://fivedots.coe.psu.ac.th/~ad/jg/ch10/. I'll focus on the Leap-related code here, and won't be explaining the complexities of Java 3D. The application is called LeapModel3D.

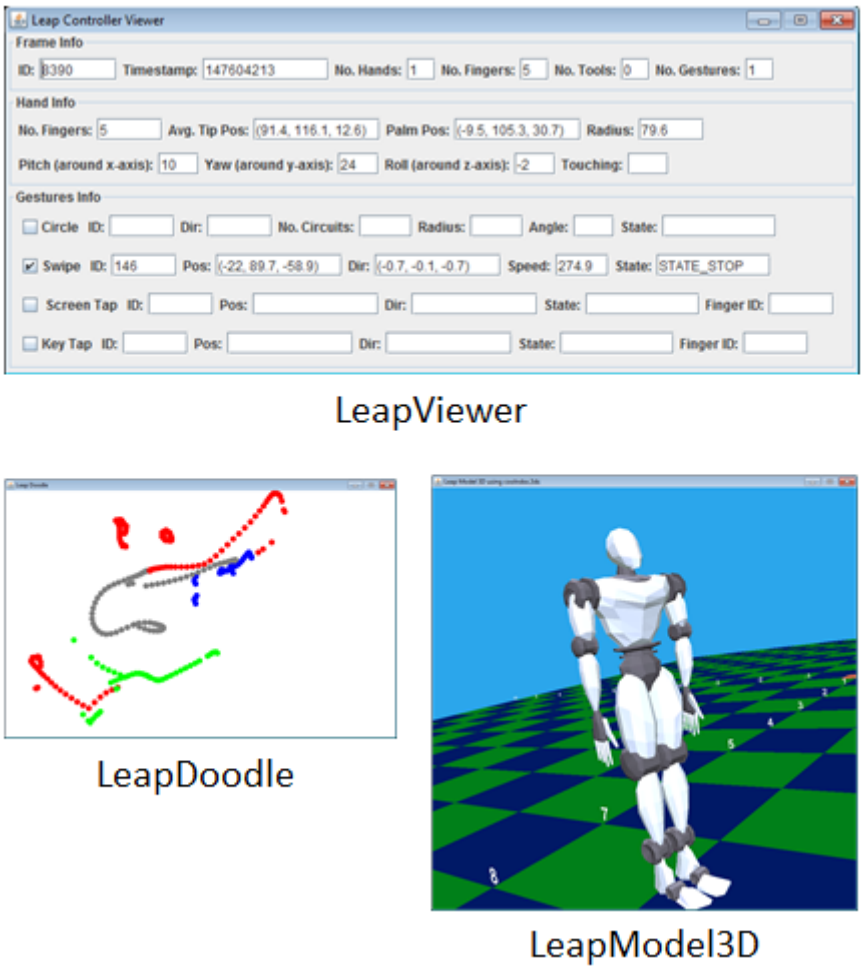

Figure 2. The Three Leap Examples.

1. An Overview of the Leap Motion API

The API returns very detailed information about the user's hands, fingers, tools held in the hand (such as a pen), and gestures, when they're located within a roughly hemispherical volume between 25 to 600 millimeters above the controller's center. Positional information is calculated in millimeter units relative to that center point using a left-handed coordinate system (as in Figure 3), to a spatial precision of about 0.01 mm.

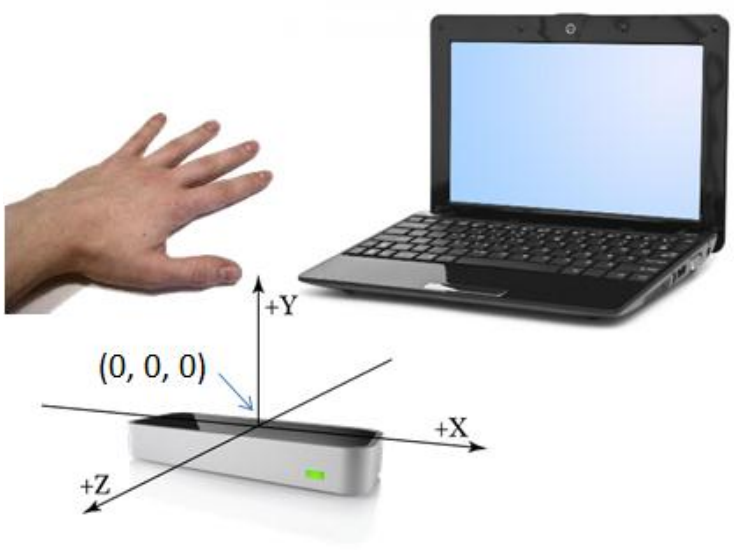

Figure 3. The Controller's Center and Axes

An assortment of directional, velocity, and rotational information is available. For instance, it's possible to read the velocity of a hand palm, its direction vector relative to the fingers, and its rotation angle clockwise around that vector (using the right-hand rule). The rotation information can also be retrieved in matrix form, or as pitch, yaw, and roll angles, as in Figure 4.

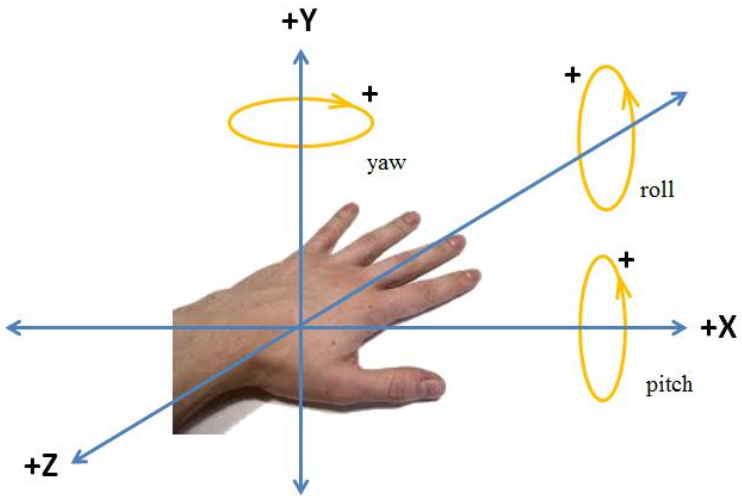

Figure 4. Pitch, Yaw, and Roll of a Hand.

Figures 3 and 4 only show one hand in action, but the Leap is capable of processing multiple hands (i.e. 2 or more), as long as they fall within its viewing volume.

Hand curvature is expressed in terms of the center and radius of a sphere that fits the curvature (see Figure 5 for two examples).

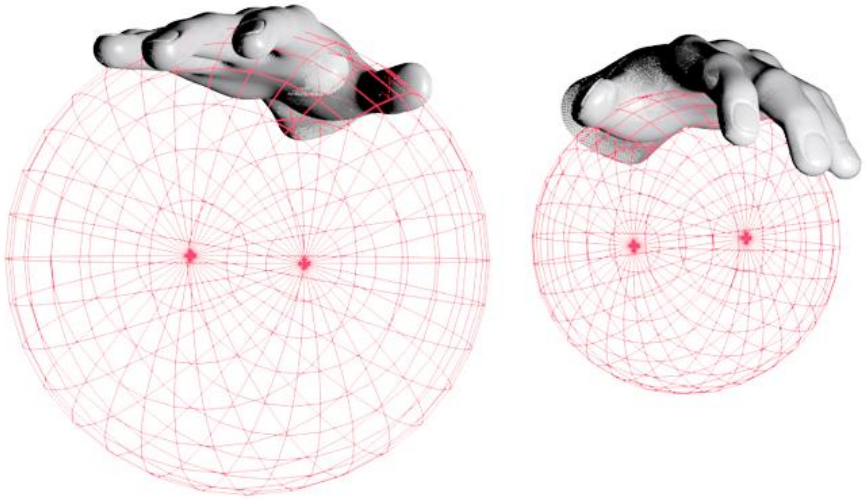

Figure 5. Hand Curvatures.

As Figure 5 suggests, the sphere reduces in size as the user bends their fingers.

The Leap gathers information into *frames* which allows velocity and other time-based data (such as gestures) to be calculated in terms of how consecutive frames change relative to the frame rate. This rate can be adjusted via the Leap's control panel, but the default 'balanced' tracking setting delivers about 10 frames/second on my Windows 7 test machine, with an average frame processing time of 4 milliseconds. A frame is constructed from the images captured from two cameras and three infrared LEDs inside the controller.

The Leap is able to distinguish between fingers and *tools* (which are longer, thinner, and straighter than a finger); the user guide's example shows a hand holding a wand. Fingers and tools share many properties, which is reflected in the API by making the Finger and Tool classes subclasses of Pointable. Pointable information include length, width, tip position, direction, and velocity.

Each detected hand, finger, tool, or gesture is assigned a unique ID. This can be employed at the programming level to track a particular entity between frames, although the approach suffers from the fact that an object's ID only remains constant while the object is visible to the Leap. For instance, if a finger is momentarily hidden from the Leap's sensors by the rest of the hand, then it will be assigned a new ID when it is next observed. The Leap developers are aware of this drawback, and the documentation mentions the future introduction of a skeletal hand model that can be tracked across frames.

1.1. Gestures

Four types of gestures are currently supported by the API:

- Circle – a single finger tracing a circle.
- Swipe – a linear movement of the hand.
- Key tap – a downwards tapping finger movement, similar to how a keyboard is pressed.

- Screen tap – a forwards tapping movement, similar to how a vertical surface, such as a screen, would be touched.

Examples are shown in Figure 6.

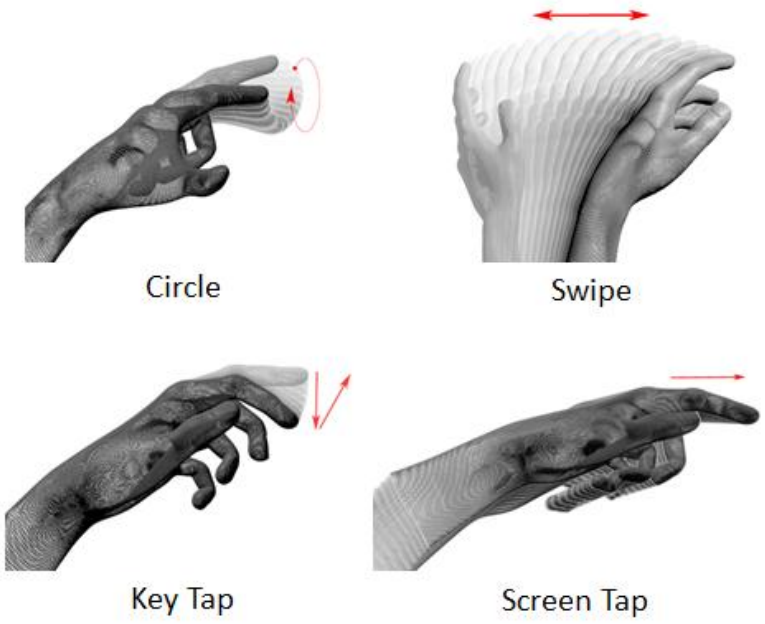

Figure 6. Leap Gestures.

'Key' and 'screen' taps don't require the user to touch a key or the screen. The gesture is interpreted in terms of the sequence of finger movements.

1.2. Other Spatial Views

Most positional data is in given in terms of millimeter x-, y-, and z- distances relative to the controller's center, as shown in Figure 3. However, it is possible to utilize three other coordinate systems: the *screen*, *interaction box*, and *touch zone*.

The API's ***Screen*** class describes the position and orientation of the user's monitor relative to the Leap's coordinate system. The data include the bottom-left corner of the screen, direction vectors for the horizontal and vertical axes of the screen, and the screen's normal vector. The class offers several intersection methods which calculate how the hand's direction vector intersects the screen. The intersection point is defined using normalized screen coordinates, like those in Figure 7.

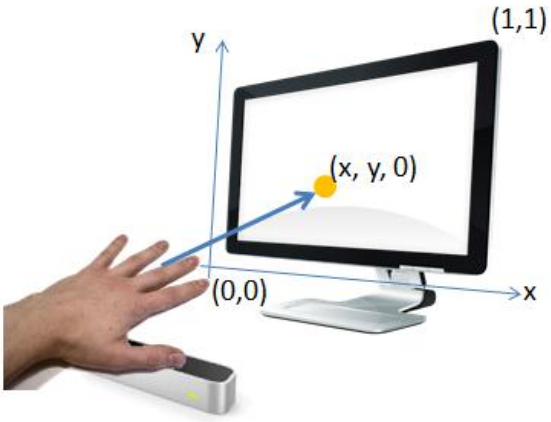

Figure 7. Intersection with the Screen.

The screen's origin is considered to be located at the bottom-left corner, with the top-right corner is normalized to be (1, 1). This means that the intersection point's x- and y- values must fall between 0 and 1 (its z- value is always 0). I use screen coordinates in my second example, LeapDoodle.

The ***interaction box*** is a rectangular space above the Leap that falls completely within its view volume (Figure 8).

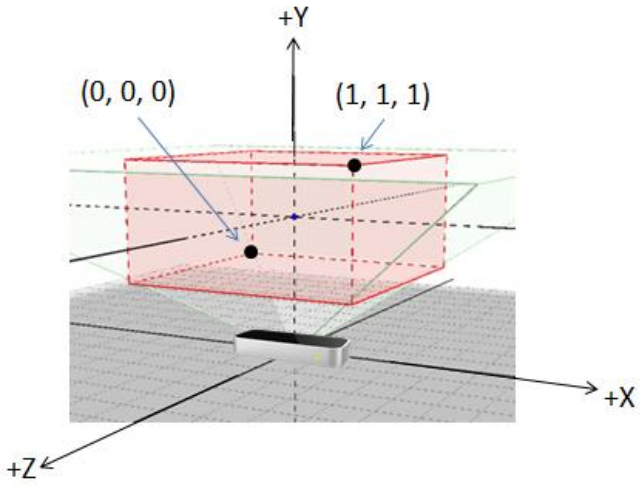

Figure 8. The Interaction Box.

A standard Leap coordinate (i.e. one defined relative to the Leap's center) can be mapped to a point relative to the box. The value is scaled so that the box's volume is treated as a unit cube with its origin at the minimum corner (the left-most, lowest, and most distant corner relative to the user).

The ***touch zone*** can be utilized with a Pointable object (i.e. a finger or tool) to implement a touch-like behavior without having to actually touch any physical surface (such as a screen). The touch zone is divided into hovering and touching zones which are identified by a touching distance that ranges between -1 and 1, as in Figure 9.

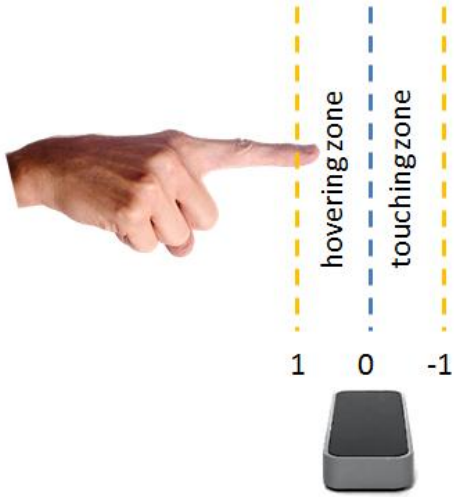

Figure 9. The Hovering and Touching Zones.

For example, the touching distance for the finger in Figure 9 would be about 0.5 indicating that it was halfway into the hovering zone.

Very clearly a great deal of thought has been put into the API, which offers many novel ways to interpret hand and finger movement.

2. Viewing the Flood

The default tracking settings for the Leap means that the SDK delivers roughly 10 frames of information per second, each containing an enormous amount of hand, finger, tool, and gesture data. The SDK comes with a useful Java example that reports this data to standard output, but the user is soon overloaded by the deluge of information that quickly scrolls off the top of the output window. My LeapViewer application replaces the standard output by a GUI, making it easier to separate the data into types, and monitor changes to the numbers over time.

A screenshot of LeapViewer is shown in Figure 10.

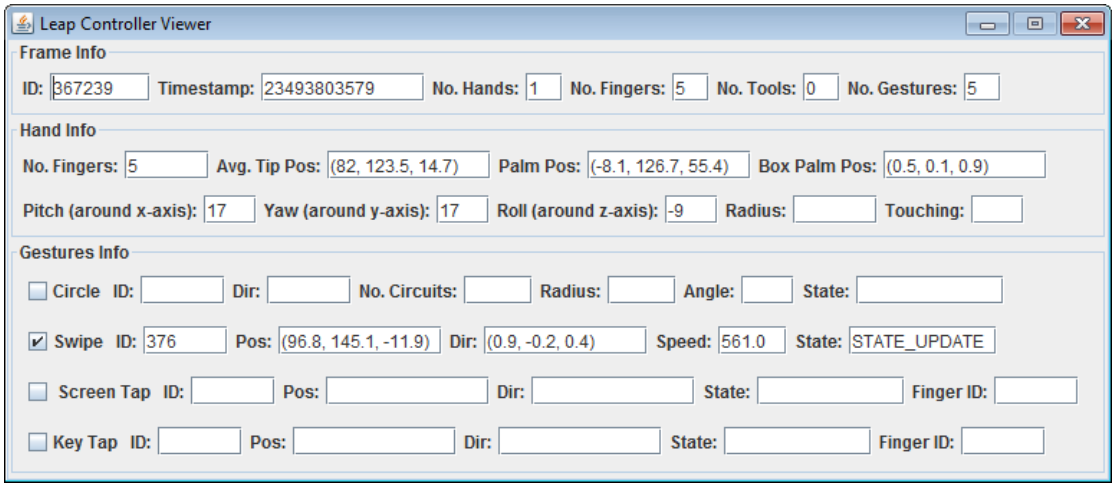

Figure 10. The LeapViewer GUI.

The information is separated into three broad groups: frame, hand, and gestures. Figure 10 indicates that the user is currently swiping a single hand left-to-right over the Leap. All five fingers are visible, and the hand is flat and level according to the rotation details (pitch, yaw, and roll).

The screenshot fails to show how rapidly the data is changing – a new frame is displayed roughly every 100 ms. A small amount of information still goes to standard output, including some configuration settings and a few details about screen and key taps which are otherwise rendered too quickly to be read from the GUI text fields. An example of the output:

```
Interaction Box Info
  center: (0, 200, 0)
  (x,y,z) dimensions: (221.4, 221.4, 154.7)
Controller has been connected
Key Tap MinDownVelocity: 30.0
Key Tap HistorySeconds: 0.1
Key Tap MinDistance: 5.0

Screen Tap MinDownVelocity: 30.0
Screen Tap HistorySeconds: 0.1
Screen Tap MinDistance: 1.0

Key tap(42): (0, -1, 0) / STATE_STOP
Key tap(42): (0, -1, 0) / STATE_STOP
Key tap(57): (0.3, -1, 0) / STATE_STOP
Key tap(8): (-0.5, -0.8, -0.3) / STATE_STOP
Key tap(51): (0, -1, -0.3) / STATE_STOP
Key tap(8): (-0.2, -0.9, -0.4) / STATE_STOP

Screen tap(35): (0, -0.6, -0.8) / STATE_STOP
Screen tap(35): (0, -0.5, -0.9) / STATE_STOP
Screen tap(35): (0, -0.6, -0.8) / STATE_STOP
```

The class diagrams for the application are shown in Figure 11.

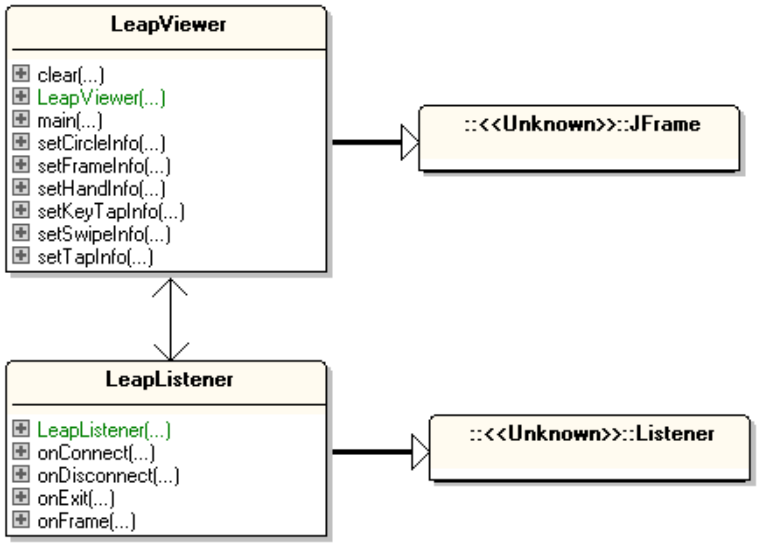

Figure 11. Class Diagrams for LeapViewer.

The LeapViewer class is mostly standard GUI code – a series of panels containing text fields and a few checkboxes, which are updated by the LeapListener object calling various public 'set' methods, such as setFrameInfo() to write data into the top-most panel of Figure 10.

I'll concentrate on explaining LeapListener, which subclasses the Leap Motion Listener class. Listener is a listener in the Java sense since it defines a set of callback functions that respond to events dispatched by the Leap controller. However, it's not implemented as an interface or abstract class unlike JDK listeners such as WindowListener. Instead, it's a concrete class whose methods must be overridden, in the same way as a JDK adapter class such as WindowAdapter.

Listener defines seven "on" callback methods which are fired when different states are reached in the Leap controller's configuration and execution. The most useful is the onFrame() method which is called whenever a new frame of data is received from the Leap.

A simple example of subclassing Listener:

```
public class FrameListener extends Listener
{
  public void onFrame(Controller controller)
  {
    Frame frame = controller.frame();   // the latest frame
    System.out.println("Received frame with ID: " + frame.id());
  }
};
```

This FrameListener class will print out the ID of a newly received frame.

The listener subclass is attached to the Leap by creating an instance of the Leap Motion Controller class, and adding the listener to it. For instance, the following code fragment might appear in an application's main() method or constructor:


```
FrameListener fl = new FrameListener();
Controller controller = new Controller();
controller.addListener(fl);
```

The corresponding code in the LeapViewer application appears in the LeapViewer() constructor:

```
// globals
private LeapListener listener;
private Controller controller;


public LeapViewer()
{
  super("Leap Controller Viewer");

  buildGUI();

  // Create a listener and controller
  listener = new LeapListener(this);
  controller = new Controller();
  controller.addListener(listener);

  addWindowListener( new WindowAdapter() {
    public void windowClosing(WindowEvent e)
    { controller.removeListener(listener);
      System.exit(0);
    }
  });

  pack();
  setResizable(false);
  setLocationRelativeTo(null);  // center the window
  setVisible(true);

  try {   // wait a bit
    Thread.sleep(5000);
  }
  catch(InterruptedException e) {}

  if (!controller.isConnected()) {
    System.out.println("Cannot connect to Leap");
    System.exit(1);
  }
} // end of LeapViewer()
```

The 'this' reference passed to LeapListener has nothing to do with the Leap's set-up; it's used by my code to call the relevant GUI 'set' methods in LeapViewer when text fields need to be updated.

The constructor also includes two other useful coding features. One is the utilization of a window listener which is triggered when the user presses the GUI's close box. I use this event to detach the listener from the controller. The constructor also tests the controller's connectivity after a 5 second wait. One reason for Controller.isConnected() to return false is if the Leap is not plugged into the PC.

2.1. Listening to the Leap

The seven 'on' methods defined in the Listener class define a state diagram for the controller and its delivery of data to the listener, as shown in Figure 12.

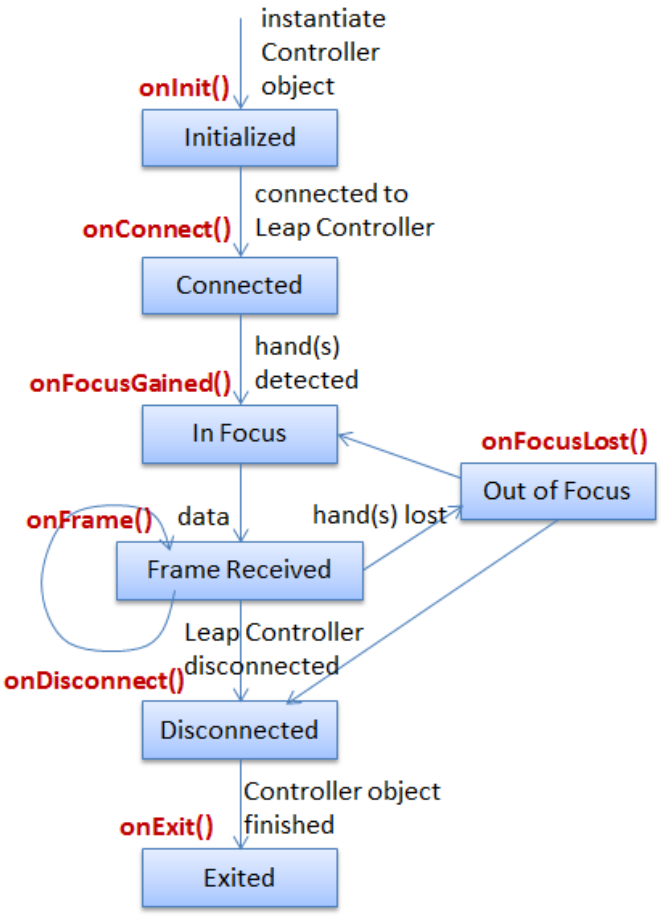

Figure 12. The States of a Controller and its Data.

The methods shown next to the states have the following signatures:

```
public void onInit(Controller paramController){}
public void onConnect(Controller paramController){}
public void onFocusGained(Controller paramController){}
public void onFocusLost(Controller paramController){}
public void onFrame(Controller paramController){}
public void onDisconnect(Controller paramController){}
public void onExit(Controller paramController){}
```

My LeapListener has one-line implementations for onInit(), onDisconnect(), and onExit():

```
public void onInit(Controller controller)
{  System.out.println("Initialized"); }

public void onDisconnect(Controller controller)
{  System.out.println("Disconnected");  }
```



```
public void onExit(Controller controller)
{  System.out.println("Exited");  }
```

The enabling of gesture processing must occur after the Leap is connected to the Controller object, so is performed inside onConnect():

```
public void onConnect(Controller controller)
// listen for all gestures, and adjust taps
{
  System.out.println("Controller has been connected");
  controller.enableGesture(Gesture.Type.TYPE_SWIPE);
  controller.enableGesture(Gesture.Type.TYPE_CIRCLE);
  controller.enableGesture(Gesture.Type.TYPE_SCREEN_TAP);
  controller.enableGesture(Gesture.Type.TYPE_KEY_TAP);

  Config config = controller.config();
  // key tap parameters
  config.setFloat("Gesture.KeyTap.MinDownVelocity", 30.0f);

  System.out.println("Key Tap MinDownVelocity: " +
              config.getFloat("Gesture.KeyTap.MinDownVelocity"));
  System.out.println("Key Tap HistorySeconds: " +
              config.getFloat("Gesture.KeyTap.HistorySeconds"));
  System.out.println("Key Tap MinDistance: " +
              config.getFloat("Gesture.KeyTap.MinDistance"));
  System.out.println();

  // screen tap parameters
  config.setFloat("Gesture.ScreenTap.MinForwardVelocity", 30.0f);
  config.setFloat("Gesture.ScreenTap.MinDistance", 1.0f);

  System.out.println("Screen Tap MinDownVelocity: " +
         config.getFloat("Gesture.ScreenTap.MinForwardVelocity"));
  System.out.println("Screen Tap HistorySeconds: " +
         config.getFloat("Gesture.ScreenTap.HistorySeconds"));
  System.out.println("Screen Tap MinDistance: " +
         config.getFloat("Gesture.ScreenTap.MinDistance"));
  System.out.println();
}  // end of onConnect()
```

The four calls to Controller.enableGesture() mean that the frame data will include information about all the different possible Leap gestures.

The Config object allows the key tap and screen tap parameters to be adjusted and printed to standard output. The changes reduce the minimum triggering velocities for the two kinds of tap, and lowers the screen tapping distance. I included this code to try to make it easier for the Leap to detect these gestures, but it had no appreciable effect, at least on my test machine.

2.2. Reporting Frame Data

The onFrame() method does all the serious work of extracting hand, finger, and gesture data from an incoming Frame object, and passing it to the GUI.


```
// globals
private LeapViewer viewer;     // GUI for Leap Controller data
private boolean reportedBox = false;
             // for reporting details about the interaction box


public void onFrame(Controller controller)
// fired when a frame is received from the Leap controller
{
  viewer.clear();    // reset the GUI window

  // get the most recent frame
  Frame frame = controller.frame();

  // report frame info to the GUI
  viewer.setFrameInfo(frame.id(), frame.timestamp(),
                frame.hands().count(),
                frame.fingers().count(), frame.tools().count(),
                frame.gestures().count());

  InteractionBox ib = frame.interactionBox();
  if (!reportedBox) {
    System.out.println("Interaction Box Info");
    System.out.println("  center: " + round1dp(ib.center()));
    System.out.println("  (x,y,z) dimensions: (" +
                     round1dp(ib.width()) + ", " +
                     round1dp(ib.height()) + ", " +
                     round1dp(ib.depth()) +")");
    reportedBox = true;
  }

  if (!frame.hands().empty())
    examineHand( frame.hands().get(0), ib);   // examine first hand

  examineGestures( frame.gestures(), controller);
}  // end of onFrame()
```

The viewer reference links the listener to the GUI. Details about the frame are set by calling LeapViewer.setFrameInfo(), which fills the "Frame Info." panel in the window (as shown in Figure 10).

The Leap Motion API has a compositional structure which is quite intuitive. A Frame object contains a hands list and a tools list, which can be viewed collectively as a single pointables list. Each hand contains a fingers list. If gestures have been enabled, then the frame will also include a list of gestures, of the type enabled in onConnect(). This frame structure is outlined in Figure 13.

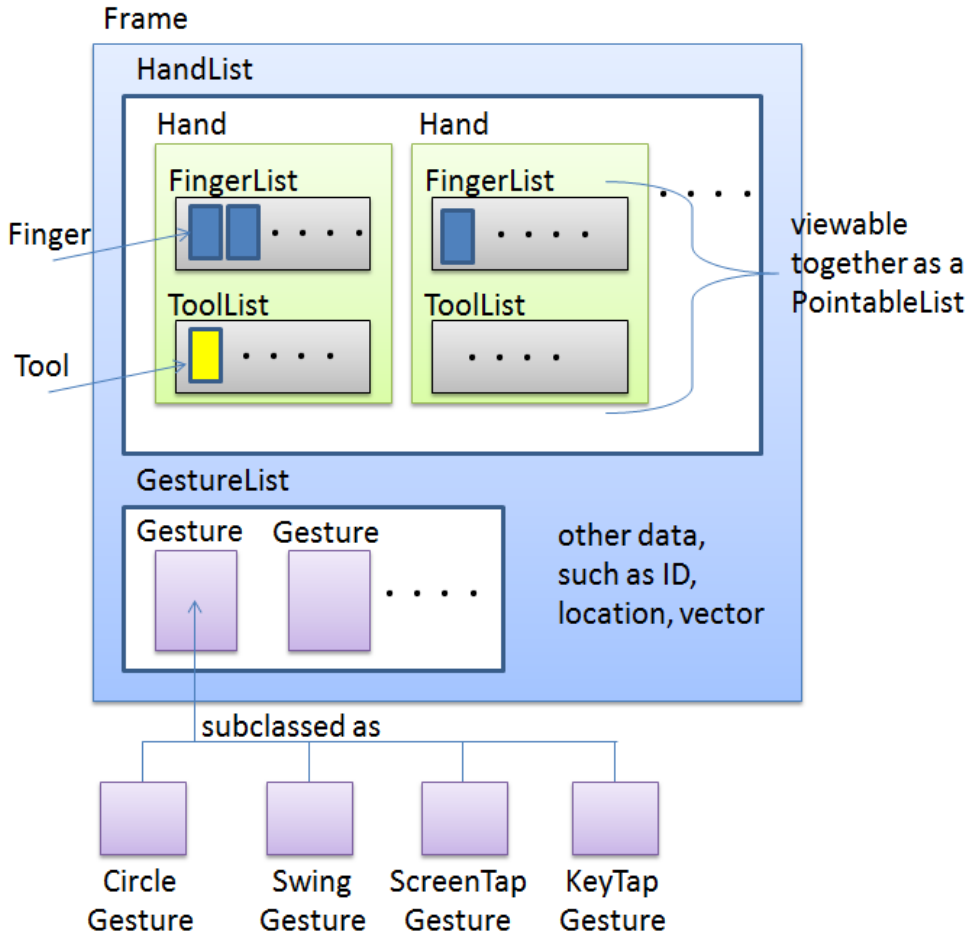

Figure 13. A Simplified View of a Frame object.

The various lists (e.g. HandList, FingerList) can be accessed using a get() method and indices, or using a for-each loop. The size of a list is obtained using a count() method, so the method call frame.hands().count() returns the length of the HandList object in the frame.

An onFrame() method usually starts with a call to Controller.frame() which retrieves the current frame. However, the controller also retains up to 60 previous frames which can be accessed using an index. It starts at 0 for the current frame, 1 for the previous frame, and so on. For instance:

```
Frame previousFrame = controller.frame(1);
```

One unusual aspect of my onFrame() method is the use of the interaction box. The InteractionBox class represents the box-shaped region within the field of view of the Leap Motion controller (see Figure 8). A boolean, reportedBox, is employed to restrict the output to the first time that onFrame() is called.

There are two round1dp() methods defined in my LeapListener, one for vectors, the other for floats, which round the data to one decimal place.

2.3. A Closer Look at a Hand

onFrame() only examines the first hand in the HandList, by looking up the first element in the HandList and passing its reference to examineHand():

```
examineHand( frame.hands().get(0), ib)
```

examineHand() accesses a range of details about the hand, and also examines its fingers. The reported hand details include the palm position, the direction of the hand, and its normal vector which typically points downwards from the palm towards the Leap. The direction and normal vectors make it easy to extract the hand's pitch, yaw, and roll angles (shown in Figure 4). The fingertip positions are used to calculate an average finger position, and to determine if the front-most finger is in the touch zone.

```
private void examineHand(Hand hand, InteractionBox ib)
{
  int fCount = 0;
  Vector avgPos = Vector.zero();

  // check if the hand has any fingers
  FingerList fingers = hand.fingers();
  if (!fingers.empty()) {
    // Calculate the hand's average fingertip position
    fCount = fingers.count();
    for (Finger finger : fingers)
      avgPos = avgPos.plus(finger.tipPosition());
    avgPos = avgPos.divide(fingers.count());
  }

  /* check if finger is deep within the touch zone, which
     ranges from 1 to -1 (-1 being nearer the screen) */
  boolean isTouched = false;
  Finger frontFinger = hand.fingers().frontmost();
  float touchDist = frontFinger.touchDistance();
  if (touchDist< -0.8f) {
    // System.out.println("Pressed: touch distance: " + touchDist);
    isTouched = true;
  }

  // get the hand's normal vector and direction
  Vector normal = hand.palmNormal();
      // a unit vector pointing orthogonally downwards
      // relative to the palm
  Vector direction = hand.direction();
        // a unit vector pointing from the palm position
        // towards the fingers

  // calculate the hand's pitch, roll, and yaw angles
  int pitch = (int) Math.round( Math.toDegrees(direction.pitch()));
  int roll = (int) Math.round( Math.toDegrees(normal.roll()));
  int yaw = (int) Math.round( Math.toDegrees(direction.yaw()));

  // convert the palm to interaction box coordinates
  Vector palmIB = ib.normalizePoint(hand.palmPosition());

  // report hand info to the GUI
  viewer.setHandInfo(fCount, round1dp(avgPos),
                       round1dp( hand.sphereRadius()),
                       round1dp(hand.palmPosition()),
                       round1dp(palmIB),
```



```
                        pitch, roll, yaw, isTouched);
}  // end of examineHand()
```

The call to Finger.touchDistance() returns a float between -1 and 1, which corresponds to a position in the touch zone (see Figure 9). My code looks for a value less than -0.8, which would place the fingertip well into the touching zone.

The pitch, roll, and yaw are converted from radians to degrees, and rounded to make them easier to read when displayed in the GUI.

The palm position is reported twice, once as Leap coordinates (using hand.palmPosition()), and once relative to the interaction box (by calling InterationBox.normalizePoint()).

2.4. Considering the Gestures

As Figure 13 indicates, gestures are accessed via the Frame, not through a Hand object. The typical coding style is to iterate over the GestureList, and use a multiway branch to distinguish between the different gesture types. The Gesture.type() method returns a constant which allows each Gesture object to be cast to its correct subclass: CircleGesture, SwingGesture, ScreenTapGesture, or KeyTapGesture (as in Figure 13). This coding approach is evident in my examineGestures() method:

```
private void examineGestures(GestureList gestures,
                                    Controller controller)
{
  int fID;   // the ID of the finger making the gesture

  for (Gesture gesture : gestures) {
    switch (gesture.type()) {
      case TYPE_CIRCLE:
        CircleGesture circle = new CircleGesture(gesture);
        // calculate clock direction using the angle between
        // circle normal and pointable
        boolean isClockwise =
              (circle.pointable().direction().
               angleTo(circle.normal()) <= Math.PI/4);

        // calculate angle swept since last frame
        double sweptAngle = 0;
        if (circle.state() != State.STATE_START) {
          CircleGesture previousUpdate = new CircleGesture(
                      controller.frame(1).gesture(circle.id()));
          sweptAngle = (circle.progress() -
                previousUpdate.progress()) * 2*Math.PI;
        }

        int angle = (int) Math.round(Math.toDegrees(sweptAngle));
        viewer.setCircleInfo(circle.id(), circle.state(),
                 isClockwise, round1dp(circle.progress()),
                 round1dp(circle.radius()), angle);
        break;

      case TYPE_SWIPE:
        SwipeGesture swipe = new SwipeGesture(gesture);
        viewer.setSwipeInfo(swipe.id(), swipe.state(),
                 round1dp(swipe.position()),
```



```
                round1dp(swipe.direction()),
                round1dp(swipe.speed()));
        break;

      case TYPE_SCREEN_TAP:
        ScreenTapGesture screenTap = new ScreenTapGesture(gesture);
        fID = screenTap.pointable().id();
               // finger ID will stay the same across frames
        viewer.setTapInfo(screenTap.id(), screenTap.state(),
                        round1dp(screenTap.position()),
                        round1dp(screenTap.direction()), fID);
        break;

      case TYPE_KEY_TAP:
        KeyTapGesture keyTap = new KeyTapGesture(gesture);
        fID = keyTap.pointable().id();
             // finger ID will stay the same across frames
        viewer.setKeyTapInfo(keyTap.id(), keyTap.state(),
                        round1dp(keyTap.position()),
                        round1dp(keyTap.direction()), fID);
        break;

      default:
        System.out.println("Unknown gesture type.");
        break;
    }
  }
}  // end of examineGestures()
```

Each gesture has an ID and current state, although the ID is not that useful since it only exists during the lifetime of the gesture. Particularly for key and screen taps, it's more useful to have the ID of the finger that's making the gesture, which is available via the gesture's pointable() method. Most of the gesture classes also support methods for returning the position, direction, and speed of an action. There are four gesture state values – STATE_INVALID, STATE_START, STATE_STOP, and STATE_UPDATE (which means that the gesture is in progress).

Probably the trickiest code in examineGestures() deals with finger twirling (TYPE_CIRCLE). The rotation direction (clockwise or counter-clockwise) is obtained by calculating the angle between the finger direction and the vector normal to the rotation; the two cases are illustrated in Figure 14.

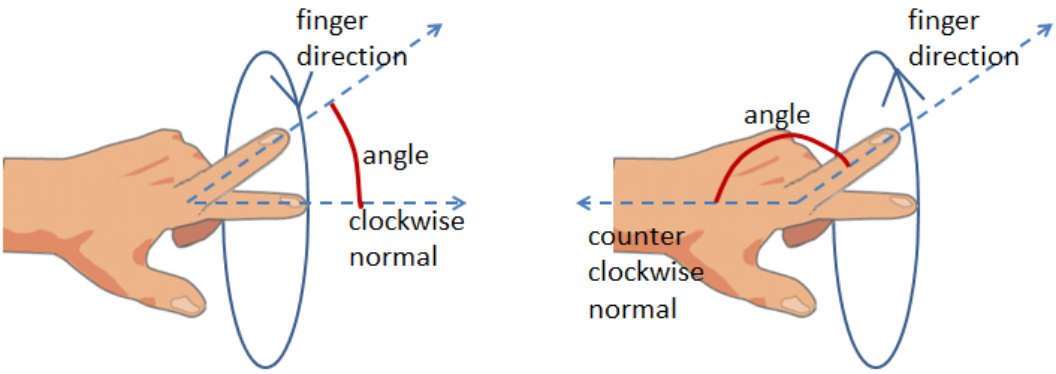

Figure 14. Rotation Angle for a Circling Finger.

The code sets a boolean depending on if the angle is acute (actually 45 degrees or less):

```
boolean isClockwise =
  (circle.pointable().direction().angleTo(circle.normal())
                                                  <= Math.PI/4);
```

Another interesting piece of code is the calculation of the arc angle travelled between the previous frame and the current one, which requires data from the previous frame (i.e. from controller.frame(1)):

```
if (circle.state() != State.STATE_START) {
  CircleGesture previousUpdate = new CircleGesture(
                      controller.frame(1).gesture(circle.id()));
  sweptAngle = (circle.progress() -
                previousUpdate.progress()) * 2*Math.PI;
}
```

The previous frame's circle gesture is accessed using the current circle gesture's ID, which could fail if the gesture had started in this frame. That's the reason for wrapping the code in a test of the circle's state to ensure that it's not just begun.

CircleGesture.progress() doesn't return an angle but a float indicating the fraction of the circle completed. For instance, a 0.5f result would indicate that the finger had progressed halfway around the circle. The conversion to radians is obtained by multiplying by 2π.

The four 'set' methods in examineGestures() pass data to the GUI which writes it into the relevant text fields (see Figure 10).

2.5. Gesturing Ease

One advantage of a GUI interface is that it separates the voluminous hand and finger data from the gesture output. This makes it easier to evaluate how easy it is to perform a gesture.

On my test machine, circling gestures were reliably detected most of the time, as long as the rotating tip of the finger was above the Leap, or very close.

Swipe gestures were slightly less reliable, and seem to work best when the swing extends beyond the left or right edges of the Leap before stopping; small swipes were often not recognized.

Key taps work well, but require a rather exaggerated swing downwards of a finger, and work best when the fingertip is over the Leap.

Screen taps were by far the least reliable gesture, perhaps because they seem to require not only a forward movement of a finger, but a distinct period of no motion, followed by a short movement of the finger backwards or down. It took me a while to learn to do these three steps.

Taps only seem to be detected at the end of the gesture (i.e. when Gesture.state() returns STATE_STOP), and are only reported for a single frame's duration. This means that my GUI displays the information for 100 ms before the text fields are cleared, which makes it difficult to read the output. As a consequence, I added calls to System.out.println() in the 'set' methods for screen and key taps, so there was a longer-lived record of the tap details.

An alternative way of implementing a form of screen tapping is by monitoring the touching zone (see Figure 9). This is carried out in the examineHand() method described earlier, using:

```
float touchDist = frontFinger.touchDistance();
if (touchDist< -0.8f)
  isTouched = true;
```

3. Doodling with the Leap

The second Leap application consists of a canvas where the user can move a dot, draw dots, delete them, and change the drawing color (which starts as blue). An example of my artistic endeavors are shown in Figure 15.

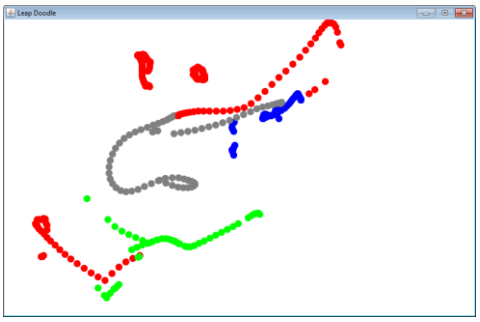

Figure 15. Doodling with Dots.

These actions are controlled via the Leap controller using:

- one finger to move the current dot;
- two fingers to draw a trail of dots onto the canvas as the current dot is moved;

- three fingers to un-draw dots, starting with the most recent and working backwards for as long as three fingers are detected.

If two hands are observed, then the dot stops moving, being drawn, or being deleted. If a circling finger gesture is performed by the leftmost hand then the color of the current dot cycles through a range of values; counter-clockwise circling reverses the cycling of the colors.

The first dot always starts at the center of the screen, and movement is relative to the region of the screen that the hand is pointing towards. In other words, the hand's direction vector is mapped to screen coordinates using Leap's Screen class as in Figure 7.

The class structure is shown in Figure 16.

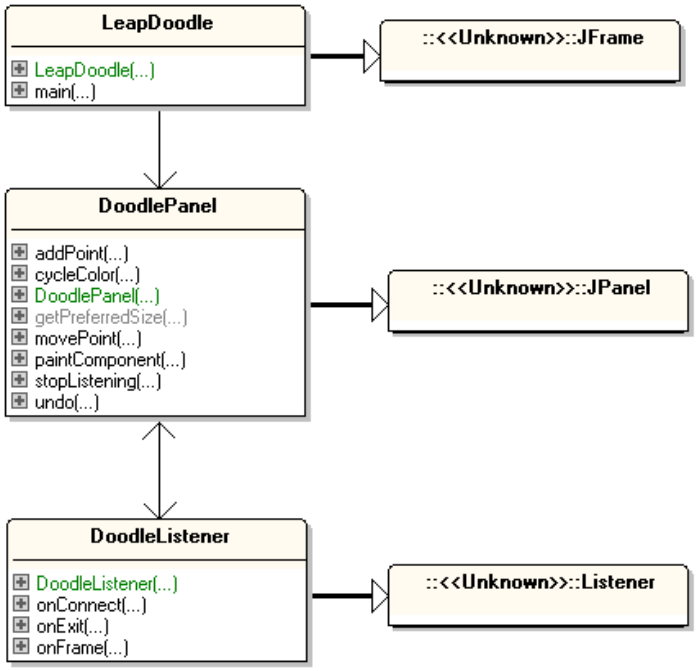

Figure 16. Class Diagrams for the LeapDoodle Application.

The top-level LeapDoodle class is responsible for window creation, the canvas is managed by DoodlePanel, as a subclass of JPanel, and the Leap-specific code is implemented as a subclass of Listener called DoodleListener.

Structurally, DoodleListener is quite similar to LeapListener from the previous example, but somewhat simpler since it deals with specific hand and finger behaviors and only a circling gesture.

The novel aspects of DoodleListener are its use of screen coordinates and coding tricks to slow down the drawing rate and color change.

3.1. Creating the Listener

The DoodleListener object is created inside DoodlePanel, and passed a reference to the panel:


```
// globals
private DoodlePanel doodlePanel;
                          // reference back to drawing area
private long startTime;
       // used to control the frequency of listener changes
private int cycleCounter = 0;
       // used to control the frequency of color changes


public DoodleListener(DoodlePanel dp)
{ super();
  doodlePanel = dp;
  startTime = System.currentTimeMillis();
}
```

The reference back to the panel is necessary so that methods can be called inside DoodlePanel to draw, move or delete a dot, or change its color. The startTime variable is used to constrain the drawing rate, and the counter (cycleCounter) implements a slowdown of color changing, both of which I'll explain in the next section.

The listener's onConnect() method enables the circling gesture, but the other gestures aren't required:

```
public void onConnect(Controller controller)
{  System.out.println("Controller has been connected");
   controller.enableGesture(Gesture.Type.TYPE_CIRCLE);
}
```

3.2. Processing Each Frame

onFrame() is called whenever a new frame of Leap data is received, which occurs roughly 10 times per second. This is a reasonable rate for updating the canvas, but I noticed that the update frequency often has 'spurts' of higher speed. This can cause extra dots to be drawn, producing a jittering effect on-screen. As a result, I added timer code to reduce the update fluctuations.

onFrame() begins with a series of tests of the hand and screen data, and a check of the time that has passed since the previous call to onFrame():

```
// globals
private static final int CHANGE_INTERVAL = 100;
          // (ms) minimum interval between changes to the canvas

private DoodlePanel doodlePanel;
private long startTime;


public void onFrame(Controller controller)
{
  Frame frame = controller.frame();

  // if no hand detected, give up
  if (frame.hands().empty())
    return;
```



```
  // if the screen isn't available give up
  Screen screen = controller.calibratedScreens().get(0);
  if (screen == null) {
    System.out.println("No screen found");
    return;
  }
  if (!screen.isValid()) {
    System.out.println("Screen not valid");
    return;
  }

  // slow down processing rate to be every CHANGE_INTERVAL ms
  long currTime = System.currentTimeMillis();
  if (currTime - startTime < CHANGE_INTERVAL)
    return;   // don't do anything until CHANGE_INTERVAL has passed

  int handsCount = frame.hands().count();
  if (handsCount == 1) {      // one hand
    Hand hand =  frame.hands().get(0);
    int fingersCount =  hand.fingers().count();
    Point2D.Float normPt = calcScreenNorm(hand, screen);

    if (fingersCount == 1)        // one finger == move point
      doodlePanel.movePoint(normPt);
    else if (fingersCount == 2)   // two fingers == add point
      doodlePanel.addPoint(normPt);
    else if (fingersCount == 3)   // 3 fingers == undo
      doodlePanel.undo();

  }
  else if (handsCount == 2)
                  // two hands == circle gestures with left
    processCircle(frame, leftHand.id());

  startTime = System.currentTimeMillis();    // reset start time
}  // end of onFrame()
```

The start time is subtracted from the current time, and if the interval is less than CHANGE_INTERVAL (100 ms) then no actions are taken upon this frame.

The processing follows a two-way branch depending on how many hands are detected. If only one hand is present, then the number of fingers is used to decide between moving, drawing or deleting a dot. The conversion of the hand position to screen coordinates is dealt with by my calcScreenNorm() method.

Figure 17 shows the purpose of calcScreenNorm(): to map the screen coordinates for a hand so that the origin is at the screen's center, and scaled to be between -1 and 1, with the positive y-axis running down the screen.

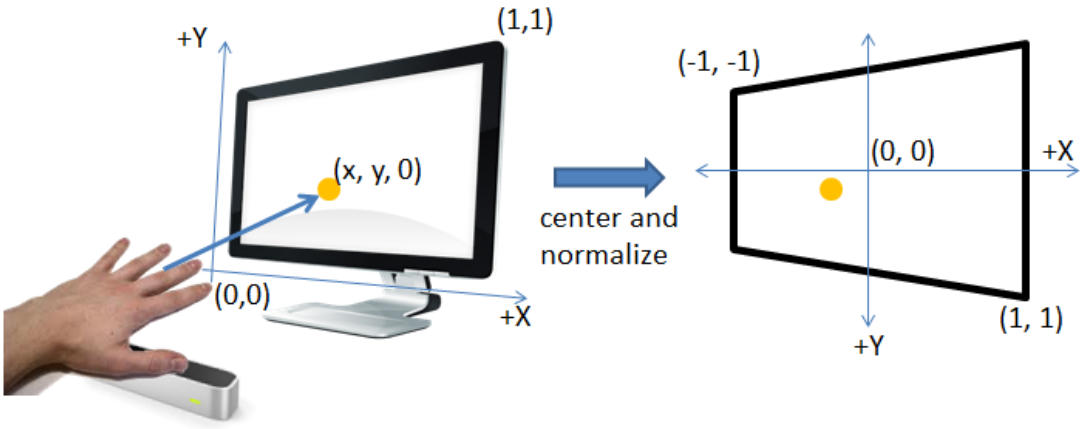

Figure 17. Mapping a Hand's Screen Coordinates.

The code for calcScreenNorm():

```
private Point2D.Float calcScreenNorm(Hand hand, Screen screen)
{
  Vector palm = hand.palmPosition();
  Vector direction = hand.direction();
  Vector intersect = screen.intersect(palm, direction, true);
        // intersection is in screen coordinates

  // test for NaN (not-a-number) result of intersection
  if (Float.isNaN(intersect.getX()) ||
      Float.isNaN(intersect.getY()))
    return null;

  // convert to point to -1 to 1 range
  float xNorm = (Math.min(1, Math.max(0,
                      intersect.getX())) - 0.5f)*2;
  float yNorm = (Math.min(1, Math.max(0,
                      (1-intersect.getY())))) - 0.5f)*2;
  return new Point2D.Float(xNorm, yNorm);
}  // end of calcScreenNorm()
```

The Screen.intersect() method does the difficult task of projecting the hand's direction onto the plane of the screen, and the rest of the code shifts the origin to the screen's center. There are two less obvious aspects of this function. The first is the testing for NaN (not a number) which occasionally arises when the Leap loses contact with the hand's position. The other is the use of Math.min() and Math.max() which utilize the fact that the intersection point is guaranteed to be between 0 and 1 (even when the hand is not pointing at the screen). This is due to the use of the 'true' setting in the call to Screen.intersect().

Back in onFrame(), if two hands are detected, then the left hand must be examined to see if the user is twirling one of his fingers. processCircle() convert this into a boolean denoting clockwise or counter-clockwise rotation, and calls DoodlePanel with the information.


```
// globals
private static final int TICK_COLOR = 4;
    // number of circle gestures required before a color change
private DoodlePanel doodlePanel;
private int cycleCounter = 0;
    // used to control the frequency of color changes


private void processCircle(Frame frame)
{
  Hand leftHand = frame.hands().leftmost();
  int leftHandID = leftHand.id();

  GestureList gestures = frame.gestures();
  for (Gesture gesture : gestures) {
    if (isGestureHand(gesture, leftHandID) &&
            (gesture.type() == Type.TYPE_CIRCLE)) {
                        // only process circles from left hand
      CircleGesture circle = new CircleGesture(gesture);
      // clock dir is angle between normal and pointable
      boolean isClockwise =
               (circle.pointable().direction().
                   angleTo(circle.normal()) <= Math.PI/4);
      if (cycleCounter == TICK_COLOR) {
        doodlePanel.cycleColor(isClockwise);
        cycleCounter = 0;
      }
      else
        cycleCounter++;
      break;
    }
  }
}  // end of processCircle()
```

In the first version of this method, because the circle gesture was detected every 100 ms, the DoodlePane.cycleColor() method was called at that rate. This changed the on-screen color of the current dot at such a fast rate that it was very difficult to control. The solution is more delaying code, this time implemented via a counter (called cycleCounter), which counts up to TICK_COLOR (4) before calling cycleColor().

Aside from the counter, the rest of the programming is similar to the earlier gesture processing method, examineGestures(), in LeapListener. For example, the finger direction and circle normal are employed to calculate the rotation direction (as in Figure 14).

I only want the user's left hand to be able to twirl a finger, which necessitates some checking of IDs. The ID for the left hand is obtained at the start of processCircle() using the useful HandList.leftmost() method (there are also rightmost() and foremost() functions):

```
Hand leftHand = frame.hands().leftmost();
int leftHandID = leftHand.id();
```

The ID is passed to my isGestureHand() method, which dereferences the gesture to its corresponding HandList. This requires a loop to check if the gesture's hand ID matches the left hand's ID:


```
private boolean isGestureHand(Gesture gesture, int handID)
// does gesture originate from the hand with the specified ID?
{
  HandList gestHands = gesture.hands();
  for(Hand h : gestHands)
    if (h.id() == handID)
      return true;
  return false;
}  // end of isGestureHand()
```

3.3. A Good Choice of Behaviors?

The Leap API is flexible enough to allow me to implement a wide range of hand, finger and gesture behaviors, but are they simple to use, and reliable? The simplest operation is probably dot moving (which requires a single finger), while drawing needs two fingers, and deletion three. Color changes requires the user to bring their left hand into the Leap's field of view, and then rotate a finger

All of these interactions seem fairly straightforward, except that my approach presupposes that the user is using their right hand as the doodling tool, but what if they're left-handed?

A more serious problem is that the drawing gesture is not reliably detected in many situations since it requires the Leap to 'see' two fingers. This is a problem because the user's hand often obscures the Leap's view of the fingers. This shouldn't come as a surprise bearing in mind how the Leap is implemented, using up-facing cameras. More generally, the Leap will probably have difficultly recognizing gestures that rely on the close proximity of fingers, such as grasping or pinching.

4. Moving a 3D Model

Many Leap Motion demos seem to involve 3D, so it's natural for me to want to include such an example here. The problem is that 3D programming, whether in OpenGL, DirectX, JavaFX, or Java 3D, is not simple. So I've decided to avoid talking about most of the 3D aspects of my LeapModel3D example, shown in Figure 18.

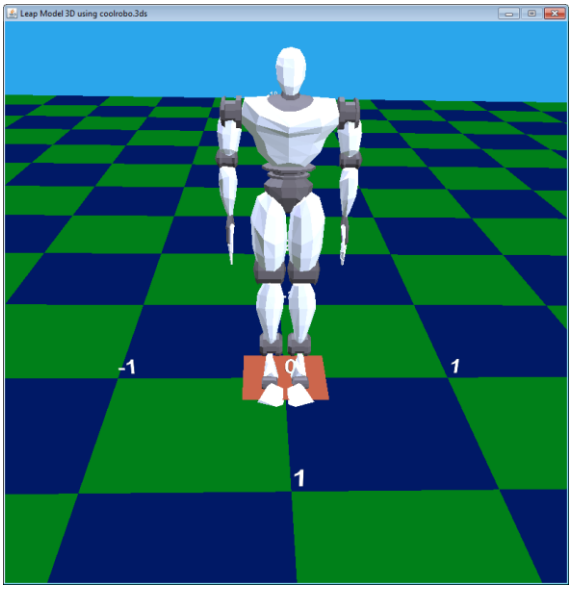

Figure 18. The LeapModel3D Application.

The code is described in great detail in my book *Killer Game Programming in Java*, where the background Java 3D concepts are explained, including the creation of lights, backgrounds, the loading of models, their translation, rotation, and scaling, and adjustments to the camera (the user's view of the 3D world). This example (minus the use of the Leap) appears as Chapter 18. An early draft of that chapter, and all the others in the book, are online, with this example explained at http://fivedots.coe.psu.ac.th/~ad/jg/ch10/

I've modified and simplified the example in several ways. The most important change is to replace keyboard controls of the robot by the Leap. I've also removed most of the 3D scenery, including a castle, a giant, attacking purple hand, and obstacles blocking the robot's movement.

I'll limit myself to an overview of the application in this section, focusing on the connection between Java 3D and the Leap. This involves using hand positions to move the robot over the checkerboard floor (the robot glides, without bending its legs), and rotating it around the vertical axis.

The translations are controlled using a hand's x- and z- position relative to the Leap controller, while rotation is implemented by a circling finger gesture. One pleasant aspect of these behaviors is the ability to combine them, to make the robot move forward through an arc for example.

The class diagrams for LeapModel3D in Figure 19 will be familiar to any reader of the 3D sections of *Killer Game Programming in Java*.

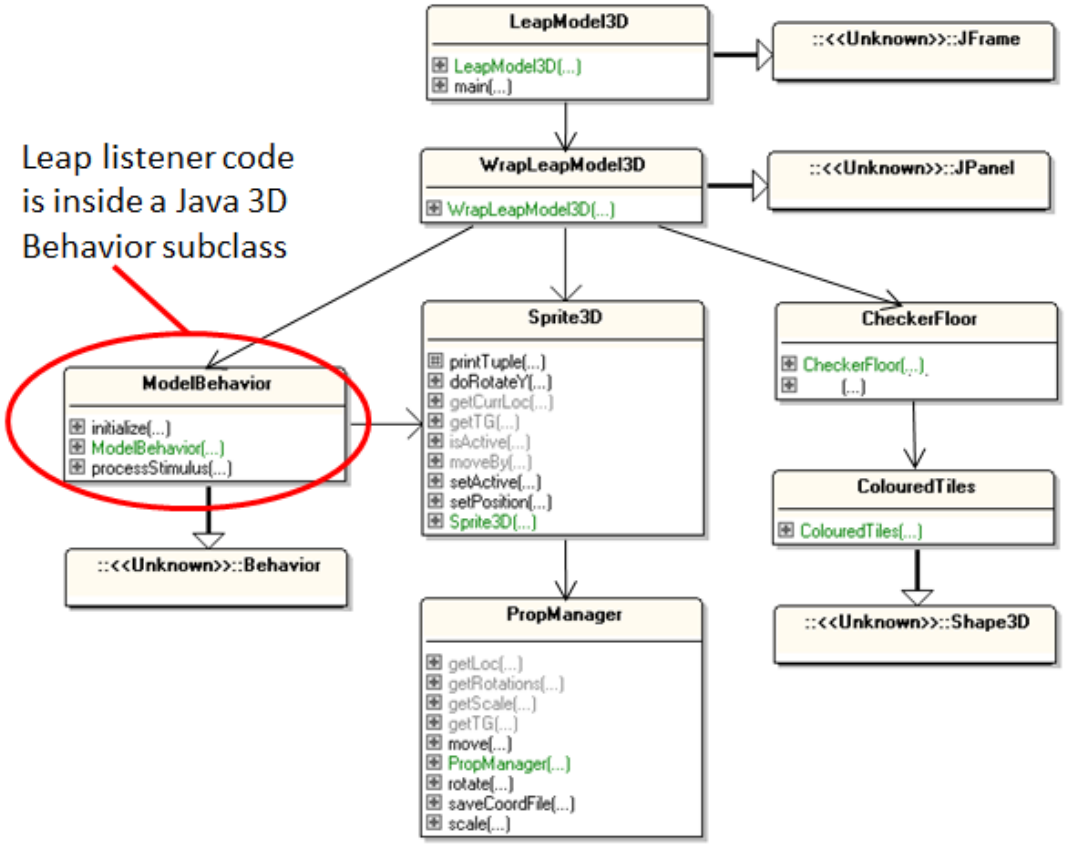

Figure 19. Class Diagrams for LeapModel3D.

Many of the classes create different 3D elements in the scene. For instance, CheckerFloor and ColouredTiles are responsible for building the floor, while PropManager handles the loading and initial positioning of the 3D robot model. The translation and rotation of the model at run time is performed by the Sprite3D class, by calling its moveBy() and doRotateY() methods.

The Java 3D way of dealing with user input (and most other forms of change inside the scene) is by subclassing the built-in Behavior class. In the keyboard-driven version of this example, the user's key presses were caught by a Behavior subclass and converted into calls to the sprite's moveBy() and doRotateY() functions. The ModelBehavior class does something similar in the current example, except that it's triggered by the Leap's detection of hand movements and circling gestures. I'll concentrate on explaining the details of that class.

4.1. Building a Scene Graph

The main task of most Java 3D programs is to build a data structure called a scene graph, which connects the 3D elements in the scene together. In LeapModel3D, this job is performed by the WrapLeapModel3D class, which creates a scene graph like the one in Figure 20.

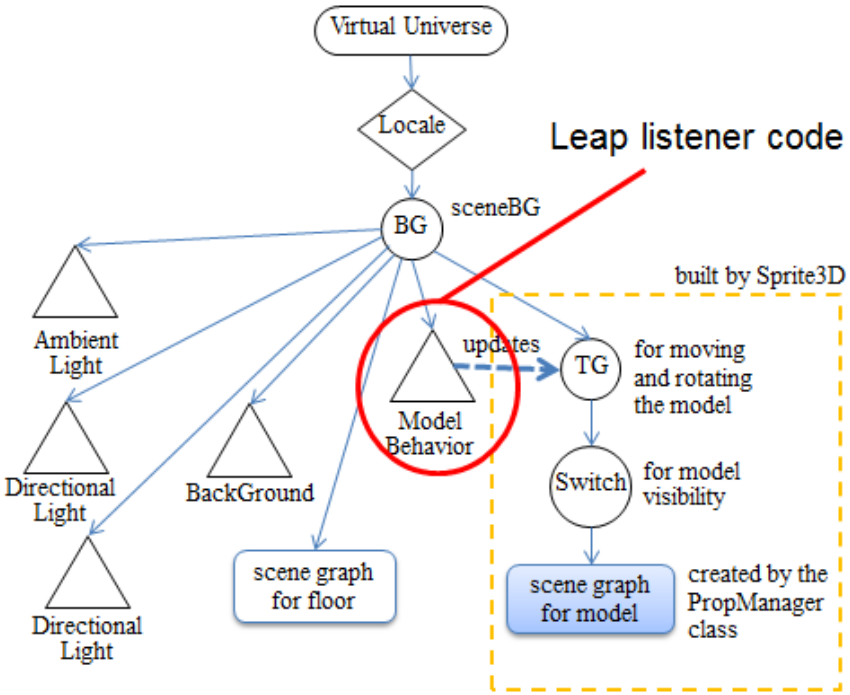

Figure 20. The LeapModel3D Scene Graph.

The graph includes a node for a ModelBehavior object which will call Sprite3D.moveBy() and Sprite3D.doRotateY() to translate and rotate the 3D model. Inside the Sprite3D class this is achieved by applying matrix operations to a TransformGroup node (labeled as TG in Figure 20), which affects the model connected beneath it. I won't discuss the rest of the graph since it's not really relevant for listening to the Leap.

4.2. Implementing a Java 3D Behavior

A Behavior object is used to monitor events occurring in a Java 3D application, such as key presses, 3D rendering, the passage of time, the movement of the camera, node changes in the scene graph, and collisions. These events, called wakeup criteria, activate the Behavior object so it can carry out specified tasks.

A typical Behavior subclass has the following format:

```
public class FooBehavior extends Behavior
{
  private WakeupCondition wc;   // what will wake the object
  // other global variables

  public FooBehavior(…)
  {  // initialize globals
     wc = new ...  //  create the wakeup criteria
  }

  public void initialize()
  // register interest in the wakeup criteria
```



```
  {  wakeupOn(wc);  }


  public void processStimulus(Enumeration criteria)
  {
     WakeupCriterion wakeup;
     while (criteria.hasMoreElements() ) {
       wakeup = (WakeupCriterion) criteria.nextElement();
       // determine the type of criterion assigned to wakeup;
       // carry out the relevant task;
     }

     wakeupOn(wc);  // re-register interest
   } // end of processStimulus()

}  // end of FooBehavior class
```

A subclass of Behavior must implement initialize() and processStimulus(). initialize() should register the behavior's wakeup criteria, but other initialization code can be placed in the constructor for the class. processStimulus() is called by Java 3D when an event (or events) of interest to the behavior is received. Often, the simple matter of processStimulus() being called is enough to decide what task should be carried out (e.g. as in the ModelBehavior class described below). In more complex classes, the events passed to the object must be analyzed. For example, a key press may be the wakeup condition, but the code will also need to determine which key was pressed.

A WakeupCondition object can be a combination of one or more WakeupCriterion. There are many subclasses of WakeupCriterion, including:

- **WakeupOnAWTEvent**. For AWT events such as key presses and mouse movements.
- **WakeupOnElapsedFrames**. An event is generated after a specified number of renderings.
- **WakeupOnElapsedTime**. An event is generated after a specified time interval. WakeupOnElapsedTime is used in ModelBehavior

4.3. A Leap Behavior

My ModelBehavior class uses a WakeupOnElapsedTime criteria to trigger it's processStimulus() method every 100 ms. This means that there's no need to subclass Leap's Listener class which implements a similar behavior. Instead, the first action of processStimulus() is to access the current frame.

The resulting ModelBehavior class has the following structure:

```
public class ModelBehavior extends Behavior
{
  private final static int DELAY = 100;   // ms, time between updates

  private Sprite3D sprite;
                     // the 3D sprite being controlled from here
  private Controller controller;
  private WakeupCondition timeOut;
                     // time-based condition to wakeup this behavior
```



```
  public ModelBehavior(Sprite3D s)
  {
    sprite = s;

    controller = new Controller();    // no Leap Listener assigned
    controller.enableGesture(Gesture.Type.TYPE_CIRCLE);

    timeOut = new WakeupOnElapsedTime(DELAY);
  } // end of ModelBehavior()



  public void initialize( )
  {  wakeupOn(timeOut);  }


  public void processStimulus(Enumeration criteria)
  // update the sprite every DELAY milliseconds
  {
    // ignore Java 3D criteria;
    // look at Controller's current frame instead
    if(controller.isConnected()) {
      Frame frame = controller.frame();

      // translate model using user's palm x- and z- position;
      // turn model using a circle gesture;
    }

    wakeupOn(timeOut);    // set wakeup condition again
  } // end of processStimulus()

} // end of ModelBehavior class
```

I've left out the processing of the hand and gesture inputs in processStimulus() to make the class easier to read.

The time-based wakeupOn() condition is created inside ModelBehavior's constructor, set in initialize(), and reset at the end of processStimulus().

A Leap Controller object is created in ModelBehavior's constructor, and the circling gesture enabled. Note that no Listener class is instantiated and assigned to the controller. Instead, the controller's current frame is directly accessed at the start of each call to processStimulus via Controller.frame(). This call is protected by a test of the controller's connectivity, which may not be established until a few hundred milliseconds after the behavior has started.

Any code that links Java 3D and the Leap will most likely use a Behavior subclass like this one. The main change being what object references are passed to the behavior as arguments of its constructor. In this case, a reference to the Sprite3D object is required, so that its moveBy() and doRotateY() methods can be called.

4.4. Processing the Leap's Input

As the comments in the processStimulus() method above suggest, there are two main stages to processing the Leap's input:

- translating the model over the floor by calling Sprite3D.moveBy() with the user's hand palm x- and z- positions;
- rotating the model around the scene's vertical axis by calling Sprite3D.doRotateY() using the rotational direction extracted from a circle gesture.

The complete code for processStimulus() is:

```
public void processStimulus(Enumeration criteria)
// update the sprite every DELAY milliseconds
{
  // ignore Java 3D criteria;
  // look at Controller's current frame instead
  if(controller.isConnected()) {
    Frame frame = controller.frame();

    // translate model using user's palm x- and z- position
    if (!frame.hands().empty()) {
      Hand hand = frame.hands().get(0);      // access first hand
      Vector handPos = hand.palmPosition();
      translateModel(handPos.getX(), handPos.getZ());
    }

    // turn model using a circle gesture
    GestureList gestures = frame.gestures();
    for (Gesture gesture : gestures) {
      if (gesture.type() == Type.TYPE_CIRCLE) {
        CircleGesture circle = new CircleGesture(gesture);

        // calculate clock dir using angle between
        // circle normal and pointable
        double turnAngle = circle.pointable().direction().angleTo(
               circle.normal()) - Math.PI/4;
            // clockwise if angle is less than 45 degrees;
            // i.e. turnAngle is negative
        if (turnAngle != 0) {  // ignore no change to angle
          boolean isClockWise = (turnAngle < 0);
          turnModel(isClockWise);
        }
        break;    // don't bother with other gestures
                  // once a circle has been processed
      }
    }
  }
  wakeupOn(timeOut);    // set wakeup condition again
} // end of processStimulus()
```

The palm's x- and z- positions are obtained from Hand.palmPosition() which returns a coordinate relative to the Leap controller (i.e. as in Figure 3).

The mapping of the positions to a suitable call to Sprite3D.moveBy() is handled by translateModel():

```
// globals
// ignore x- z- distances smaller than these constants
private final static float MIN_X_DIST = 40;   // mm
private final static float MIN_Z_DIST = 30;   // mm
```



```
private final static double MOVE_AMT = 0.3;    // translation step

private Sprite3D sprite;   // the 3D sprite


private void translateModel(float x, float z)
// move sprite forward, back, left, or right
{
  if (Math.abs(x) > MIN_X_DIST) {
    if (x > 0)
      sprite.moveBy(MOVE_AMT, 0);     // right step
    else if (x < 0)
      sprite.moveBy(-MOVE_AMT, 0);    // left step
  }

  if (Math.abs(z) > MIN_Z_DIST) {
    if (z > 0)
      sprite.moveBy(0, MOVE_AMT);    // forwards
    else if (z < 0)
      sprite.moveBy(0, -MOVE_AMT);   // backwards
  }
}  // end of translateModel()
```

translateModel() moves the sprite by a fixed distance in one of four directions. The chosen move is preceded by tests of the Leap x- and z- values, so that smallish offsets from the Leap's origin are ignored. This increases the size of a "stationary zone" above the Leap where a hand will have no effect on the model.

In processStimulus(), the circle gesture is detected in the usual way, by cycling through all the detected gestures. The conversion to a clockwise boolean is achieved in much the same as before except that an angle of 0 is ignored. The turnModel() method converts the boolean into a suitable call to Sprite3D.doRotateY():

```
// globals
private final static double ROTATE_AMT = Math.PI / 16.0;
                                // rotation step

private void turnModel(boolean isClockWise)
// rotate the model left or right about the y-axis
{
  if (isClockWise)
    sprite.doRotateY(-ROTATE_AMT);    // clockwise
  else
    sprite.doRotateY(ROTATE_AMT);    // counter-clockwise
}  // end of turnModel()
```

4.5. Analysis of the Leap Behaviors

The use of hand position and circling is reliable, except that it's hard to keep the model stationary because it's unclear how far the 'stationary zone' extends from the center of the Leap.

Another problem, more related to my Java 3D coding, is that the translation direction is relative to the robot's viewpoint. For example, the robot is initially facing towards the camera (as in Figure 18), and so forward means towards the camera. This is

implemented by the user positioning their hand along the Leap's positive z-axis (i.e. away from the screen). However, after the robot has been rotated to face the opposite direction (as in Figure 21), forward is still implemented by a backward hand movement, even though the user may perceive 'forward" as being nearer the screen (i.e. between the Leap and the screen).

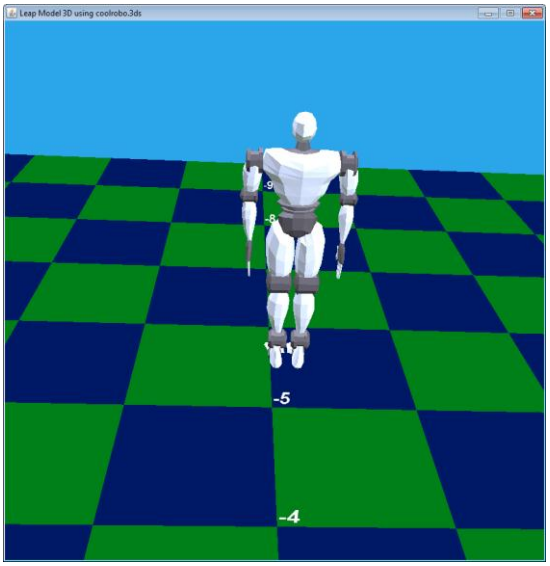

Figure 21. The Robot Rotated by 180 Degrees.

This same problem of deciding which viewpoint (robot or user) to use to control movement occurs when using keyboard input.

