Kinect Chapter 10. Gesture GUIs

[**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.]

OpenNI gestures and NITE hand tracking open the door to new kinds of user interfaces utilizing (what I'll call) gesture GUI (GGUI) components. I'll develop three examples in this chapter – a GGUI button, slider, and dial, which are shown deactivated at the top of the screenshot in Figure 1.

Figure 1. Deactivated GGUI Components.

GGUI components share the same general behavior by inheriting a superclass which allows a component to move between three states: inactive, active and pressed.

A component is activated when the user's hand enters its area on screen. The component's image changes (there are examples in Figure 2 below), and may react to hand movements inside its area. For example, the slider tab moves left and right, and the dial knob rotates. If the user doesn't move their hand for two seconds while inside the component, it changes to the pressed state.

The user knows when a component is active or pressed by its appearance changing, as illustrated in Figure 2 for the button, slider, and dial GGUIs.

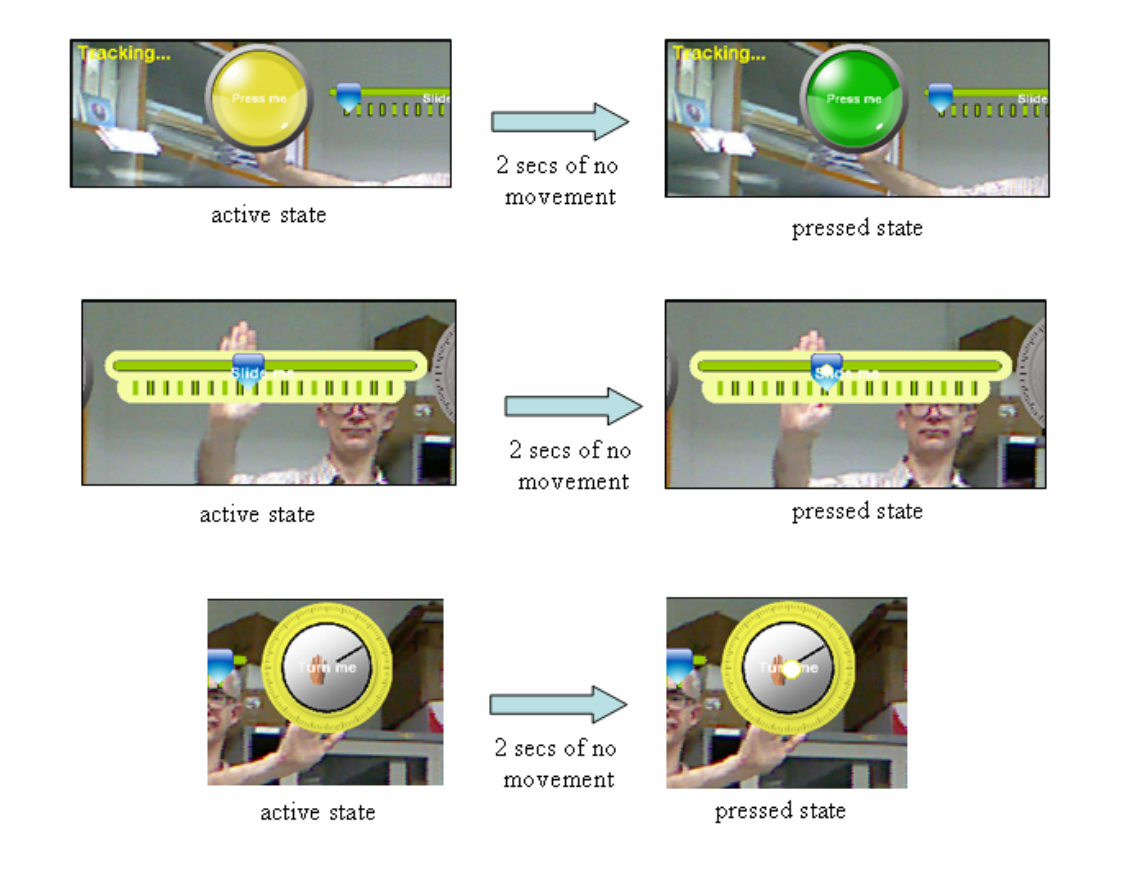

Figure 2. Active and Pressed States for the Components.

The pressed states for the slider and dial have a small yellow circle on the slider tab and in the center of the dial knob.

When a component first enters the pressed state, it sends a state object to the top-level JFrame. In my example application, this information is only printed out, but it could be utilized in more complex ways.

The button state object contains the name of the button that been pressed. The slider returns an integer in the range 0-100, corresponding to the tab's relative position along the slider ruler. The dial sends an angle in degrees representing the angle that the dial's knob line makes with the positive x-axis. For instance, the information printed for the three pressed states shown in Figure 2 are:

```
"Press me" BUTTON
"Slide me" SLIDER; range: 42
"Turn me" DIAL; angle: 49
```

The quoted string is the name of the GGUI component, the capitalized word is the component type, and is followed by the data (in the case of the slider and dial). The slider tab is a little less than halfway along the slider ruler, while the knob line is 49 degrees counter-clockwise to the positive x-axis.

1. The GGUI Application Classes

This chapter's application is called TestGestureGUIs, and most of its classes are shown in Figure 3.

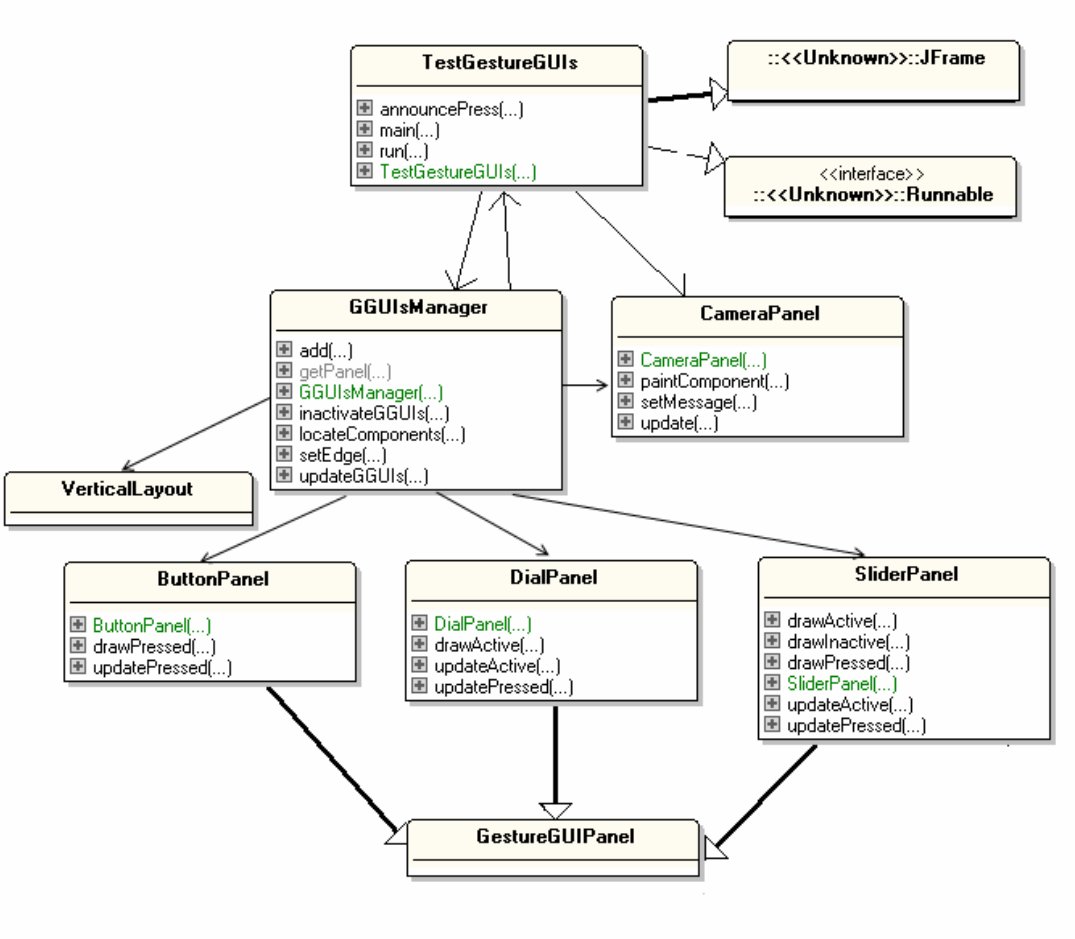

Figure 3. Class Diagrams for the TestGestureGUIs Application.

TestGestureGUI creates a full-screen undecorated JFrame made up of two layered panels – the one at the back shows the current Kinect camera image, scaled to fit the width of the screen. A panel of GGUI components is placed in the foreground layer, with the components lined up along one edge of the screen. The format is illustrated in Figure 4.

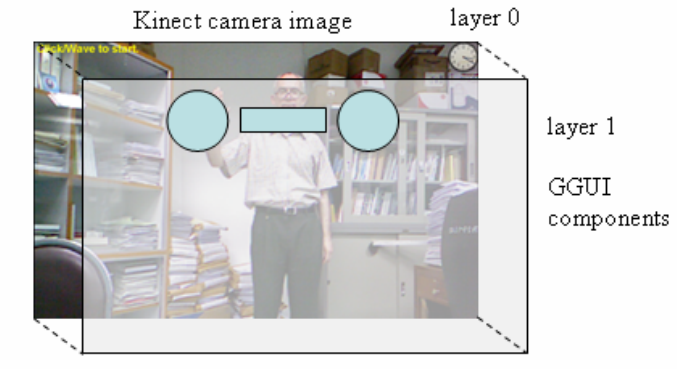

Figure 4. The Layers in TestGestureGUIs.

The camera panel in layer 0 is managed by the CameraPanel class, while the gesture GUI components in layer 1 are positioned and updated by GGUIsManager.

The components in Figure 1 are lined up along the top-edge of the screen, which is denoted by the compass heading NORTH; the other edges are EAST, SOUTH, and WEST. When the components lie along the sides of the screen (i.e. EAST or WEST), the VerticalLayout class is used to position them; Figure 11 shows the WEST layout.

TestGestureGUIs creates three GGUIs – a button, slider and dial, which are instances of the ButtonPanel, DialPanel, and SliderPanel classes respectively. These classes inherit most of their behavior from a GestureGUIPanel superclass (which is not shown in detail in Figure 3).

When a GGUI first enters its pressed state, it passes a state object to the top-level by calling TestGestureGUIs.announcePress(). This method can accept a ComponentInfo object, or one of its subclasses, whose hierarchy is shown in Figure 5.

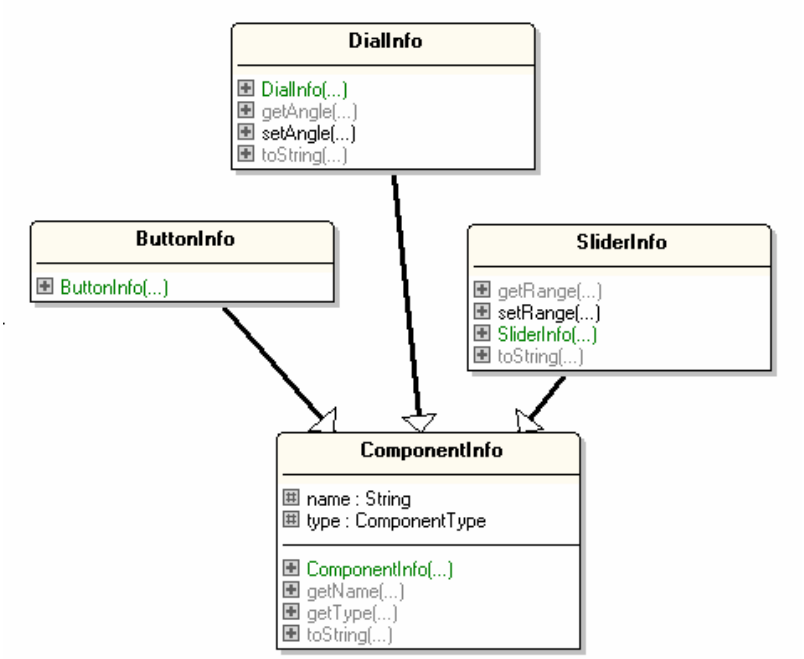

Figure 5 Class Diagrams for ComponentInfo and its Subclasses.

As the names suggest, the ButtonInfo, DialInfo, and SliderInfo subclasses are for information specific to the button, dial, and slider GGUI components.

2. Setting up the Kinect

TestGestureGUIs uses a familiar-looking configKinect() method (familiar if you've read the previous two chapters) to set up OpenNI and NITE generators and listeners. Four generators are utilized: an *image generator* for capturing the Kinect's camera input, a *depth generator* for converting real-world coordinates to screen values, a *hands generator* for hand points, and a *gesture generator* for employing a click or wave gestures to start the session (and a hand raise for refocusing).


```
// globals
private static final int XRES = 640;
private static final int YRES = 480;
                // dimensions of Kinect camera image

private Context context;
private DepthGenerator depthGen;
private ImageGenerator imageGen;
private SessionManager sessionMan;


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

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

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

    MapOutputMode mapMode = new MapOutputMode(XRES, YRES, 30);
    imageGen.setMapOutputMode(mapMode);
    depthGen.setMapOutputMode(mapMode);

    imageGen.setPixelFormat(PixelFormat.RGB24);

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

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

    GestureGenerator gesture = GestureGenerator.create(context);

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

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

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


The SessionManager object specifies the recognized gestures, and two groups of listeners are created, one associated with the SessionManager, the other for hand points.

The SessionManager listener in setSessionEvents() reports the end of the session by setting the global isRunning boolean to false.

```
// globals
private volatile boolean isRunning = true;
private SessionManager sessionMan;

private void setSessionEvents(SessionManager sessionMan)
{
  try {   // session end
    sessionMan.getSessionEndEvent().addObserver(
                        new IObserver<NullEventArgs>() {
      public void update(IObservable<NullEventArgs> observable,
                         NullEventArgs args)
      { isRunning = false; }
    });
  }
  catch (StatusException e) {
    e.printStackTrace();
  }
}  // end of setSessionEvents()
```

initPointControl() creates three hand point listeners: one for hand point updates (i.e. hand movements), one for hand point creation, and a third for when no hand points can be found.

```
// globals
private CameraPanel camPanel;
private GGUIsManager gguisMan;     // manager of GGUI components


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

    // hand has moved -- update the GGUI components
    pointControl.getPointUpdateEvent().addObserver(
                      new IObserver<HandEventArgs>() {
      public void update(IObservable<HandEventArgs> observable,
                         HandEventArgs args)
      { HandPointContext hc = args.getHand();
        gguisMan.updateGGUIs( getScreenCoord(hc.getPosition()) );
      }
    });


    // a hand point has been created
    pointControl.getPointCreateEvent().addObserver(
                             new IObserver<HandEventArgs>() {
      public void update(IObservable<HandEventArgs> observable,
                         HandEventArgs args)
      { camPanel.setMessage("Tracking...");  }
```



```
    });


    // no active hand points -- time to refocus
    pointControl.getNoPointsEvent().addObserver(
                          new IObserver<NullEventArgs>() {
      public void update(IObservable<NullEventArgs> observable,
                         NullEventArgs args)
      { gguisMan.inactivateGGUIs();   // make all comps inactive
        camPanel.setMessage("Refocus please.");
      }
    });

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

The listeners communicate with two objects – the GGUIsManager object which manages the GGUI components on layer 1, and camPanel which updates and draws the Kinect camera image on layer 0.

When a hand moves, the GGUIsManager object is sent the hand point (converted to screen coordinates) so it can decide which component is now active (if any), and which are inactive.

When a hand point is created, it means that the user has successfully started a session. A text message is passed to the camera panel, which draws it in a large yellow font at the top-left of the screen

When no hand point can be detected, it means that the user needs to refocus (or restart) the session with a suitable gesture. This situation is reported to the user by sending a "Refocus please." message to the camera panel. Also, the GGUIsManager object is told to deactivate all the components.

Changing Between Coordinates Spaces

One of the trickier aspects of the application is the number of coordinate spaces being utilized, and the need to convert hand points between them

The hand point delivered to the moving point listener is specified in real-world coordinates, which must be mapped to camera image coordinates. In addition, the image is scaled to fit the width of the screen, and the GUI components are positioned relative to those screen dimensions. Therefore, the camera coordinates must be scaled to match the screen's dimensions. This means that getScreenCoord() must carry out a two-step coordinates conversion, first from real-world to camera space, and then from camera to screen:

```
// globals
private int scrWidth, scrHeight;     // dimensions of screen
private double scaleFactor;   // for scaling image and hand points

private DepthGenerator depthGen;
```



```
private Point getScreenCoord(Point3D posPt)
{
   Point scrPt = new Point(scrWidth/2, scrHeight/2);
                                  // default screen pos
   try {
     // convert from real-world 3D to camera coordinates
     Point3D projPt = depthGen.convertRealWorldToProjective(posPt);

     // scale camera to screen coordinates
     int xPos = (int) Math.round( projPt.getX() * scaleFactor);
     int yPos = (int) Math.round( projPt.getY() * scaleFactor);
     scrPt.x = xPos;
     scrPt.y = yPos;
   }
   catch (StatusException e) {
     e.printStackTrace();
   }
   return scrPt;
}  // end of getScreenCoord()
```

The screen's width, height, and scaling factor are calculated when TestGestureGUIs is first instantiated.

3. The Gesture GUI Manager

GGUIsManager is assigned a collection of GGUI components by TestGestureGUIs calling its add() method:

```
// globals
private ArrayList<GestureGUIPanel> ggComps;  // GGUI components


public void add(GestureGUIPanel ggui)
{ System.out.println("Adding gesture GUI: " + ggui.getName());
  ggComps.add(ggui);
}
```

When all the components have been added, TestGestureGUIs sets their screen positions by calling GGUIsManager.setEdge():

```
public void setEdge(Compass heading)
// position the components along one edge of the panel
{
  switch (heading) {
    case NORTH: positionNorth(); break;
    case EAST: positionEast(); break;
    case SOUTH: positionSouth(); break;
    case WEST: positionWest(); break;
  }
}  // end of setEdge()
```

positionNorth() places the components inside a panel in the north of a BorderLayout (as in Figure 1).

```
private void positionNorth()
{
  JPanel p = new JPanel();
  p.setOpaque(false);
  p.setLayout(new FlowLayout(FlowLayout.CENTER));

  for(GestureGUIPanel ggui : ggComps) {
    ggui.setAlignmentY(Component.TOP_ALIGNMENT);
    p.add(ggui);
  }
  compsPanel.add(p, BorderLayout.NORTH);
} // end of positionNorth()
```

positionSouth() is similar, but uses BorderLayout.SOUTH.

positionEast() and positionWest() place the components in a vertical layout to the east and west (i.e. the right and left sides of screen; see Figure 11). I implemented this by employing Colin Mummery's VerticalLayout class, available at http://www.java2s.com/Code/Java/Swing-JFC/AverticallayoutmanagersimilartojavaawtFlowLayout.htm

3.1. Calculating the Component's Rectangles

Once the application becomes visible, the components' on-screen locations are fixed. TestGestureGUIs can now call GGUIsManager.locateComponents() to calculate and store their rectangular positions.

```
// globals
private ArrayList<Rectangle> ggScrRects;


public void locateComponents()
{
  ggScrRects = new ArrayList<Rectangle>();
  for(GestureGUIPanel ggui : ggComps) {
    Point pos = ggui.getLocationOnScreen();
                        // get panel's on-screen coordinates
    Dimension dim = ggui.getSize();
    ggScrRects.add( new Rectangle(pos.x, pos.y,
                                  dim.width, dim.height) );
  }
}  // end of locateComponents()
```

All the GGUI classes inherit from GestureGUIPanel, which is a subclass of JPanel. This allows me to utilize Component.getLocationOnScreen() and Component.getSize() to build the rectangle data.

3.2. Updating the GGUI Components

GGUIsManager's collection of on-screen rectangles are used during hand point processing to decide if a point is within a component's area, thereby making the

component active (and the others inactive). This is done by the hand point movement listener calling GGUIsManager.updateGGUIs():

```
// globals
private ArrayList<GestureGUIPanel> ggComps;
private ArrayList<Rectangle> ggScrRects;


public void updateGGUIs(Point scrPt)
{
  int xPos = scrPt.x;
  int yPos = scrPt.y;
  for(int i=0; i < ggComps.size(); i++) {
    GestureGUIPanel ggui = ggComps.get(i);
    if (ggScrRects.get(i).contains(xPos, yPos)) {
      Point guiPt = new Point(xPos, yPos);
          // this point will now be changed
      SwingUtilities.convertPointFromScreen(guiPt, ggui);
          // convert values in guiPt so relative to comp's rect
      ggui.updateState(guiPt);    // make this component active
    }
    else
      ggui.updateState(null);  // this comp will become inactive
  }
}  // end of updateGGUIs()
```

The hand point (scrPt) passed to updateGGUIs() is in screen coordinates so can be tested for containment against the GGUI screen rectangles. If the point is inside one of the rectangles, then the corresponding GGUI component's updateState() method is called. The point passed to the GGUI is converted into coordinates relative to the GGUI panel.

The other GGUI components also have their updateState() methods called, but with a null argument, which signals that the components should become inactive.

The hand point listener triggered when no hand points are found also calls GGUIsManager, but to deactivate the components. This is handled by GGUIsManager.inactivateGGUIs():

```
// global
private ArrayList<GestureGUIPanel> ggComps;

public void inactivateGGUIs()
{ for(GestureGUIPanel ggui : ggComps)
    ggui.updateState(null);
}
```

4. A GGUI Component Class

The GestureGUIPanel class implements the standard behavior for updating and drawing a GGUI component, which is summarized by the state diagram in Figure 6.

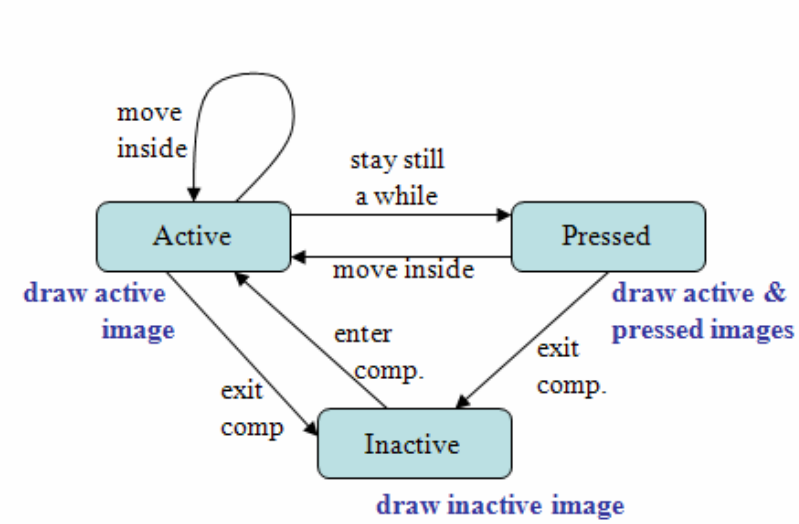

Figure 6. State Diagram for a GGUI Component.

A component can switch between three states: inactive, active, and pressed. It becomes active when the user's hand moves into it's area, and changes to the pressed state when the user's hand stays still for a period of time.

Each state has default update and rendering behaviors, which can be overridden by subclasses of GestureGUIPanel.

An active component records the current hand point position, and the time of the last hand move. When the pressed state is first entered, its default update action is to send a ComponentInfo object to the top-level by calling TestGestureGUIs.announceGGUIPress().

The inactive and active states are represented by different images that fill the component's panel. However, the pressed state drawn as a yellow circle over the active image at the center of the panel.

The state transition diagram in Figure 6 is implemented as the updateState() method in GestureGUIPanel:

```
// globals
private static final long STILL_TIME = 2000;   // ms
       /* min. duration of no movement to trigger
          active --> pressed transition  */

private GestureState gState = GestureState.INACTIVE;
private Point currPoint = null;
private long lastMovedTime = -1;
        // time when the hand point last moved

protected TestGestureGUIs topLevel;
private boolean isPressed = false;
        // has the pressed state been announced at top-level?


public void updateState(Point handPt)
{
```



```
  if (handPt == null) {     // ggui component is inactive
    gState = GestureState.INACTIVE;
    lastMovedTime = -1;
    updateInActive();
  }
  else {    // ggui component is active
    gState = GestureState.ACTIVE;
    if (lastMovedTime == -1) {  // newly active, so record time
      lastMovedTime = System.currentTimeMillis();
      isPressed = false;
      updateActive(handPt);
    }
    else {   // previously active
      long duration = (System.currentTimeMillis() - lastMovedTime);
      if (duration > STILL_TIME) {    // waited long enough?
        if (closeTo(currPoint, handPt)) {    // hand has not moved
          gState = GestureState.PRESSED;
          if (!isPressed) {
               // 'pressed' is announced once at top-level
            updatePressed();
            isPressed = true;
          }
        }
        else {   // moved, so reset time
          lastMovedTime = System.currentTimeMillis();
          isPressed = false;
          updateActive(handPt);
        }
      }
      else { // not waited long enough yet
        if (!closeTo(currPoint, handPt)) {  // hand moved, reset time
          lastMovedTime = System.currentTimeMillis();
          isPressed = false;
          updateActive(handPt);
        }
      }
    }
  }
  repaint();
}  // end of updateState()
```

The point passed to updateState() (from GGUIsManager) is relative to the GGUI panel's coordinate space, or is null. If it's null then the component becomes inactive (leaving the active or pressed state). The new state is recorded in the gState global, which is assigned a value from the GestureState enum type:

```
enum GestureState {
    INACTIVE, ACTIVE, PRESSED
}
```

If updateState() is supplied with a point, then there's several possibilities: enter the active state (either from inactivity or from the pressed state), remain active, or transition to the pressed state. This last case is chosen when the user's hand hasn't moved for STILL_TIME (2) seconds or more. closeTo() compares the current and new hand points:

```
// global
```



```
private static final int STILL_DIST2 = 300;


private boolean closeTo(Point currPt, Point newPt)
// is the new hand pt close enough to the current hand pt?
{
  if ((currPt == null) || (newPt == null))
    return false;
  int xDiff2 = (currPt.x - newPt.x) * (currPt.x - newPt.x);
  int yDiff2 = (currPt.y - newPt.y) * (currPt.y - newPt.y);
  return ((xDiff2 + yDiff2) <= STILL_DIST2);
}  // end of closeTo()
```

closeTo() ignores a small amount of movement (to compensate for hand or image shaking), based on the squared distance between the hand points.

If closeTo() returns false (i.e. there has been movement), then updateState() resets the lastMovedTime global, making the user wait STILL_TIME again before the pressed state can be entered.

updateState() calls either updateInActive(), updateActive(), or updatePressed() depending on the current state. However, updatePressed() is treated specially, only being executed when the pressed state is first entered. It won't be called again until the next active-to-pressed transition. The reason is that updatePressed() sends a state object to the top-level, which should only occur once per each active-to-pressed transition.

GestureGUIPanel's update methods are quite simple, but are likely to be overridden by more complex versions in its subclasses:

```
// globals
private Point currPoint = null;
protected TestGestureGUIs topLevel;
private ComponentInfo compInfo;  // info passed to the top-level


public void updateInActive()
{  currPoint = null;  }


public void updateActive(Point handPt)
// store hand position inside GGUI
{  currPoint = handPt;  }


public void updatePressed()
{  topLevel.announcePress(compInfo);  }
```

Every subclass will override updatePressed() so that the correct subclass object of ComponentInfo is sent to the top-level.

4.1. Component Information

The ComponentInfo class is utilized as a superclass by more specialized information classes for the button, dial, and slider GGUI components. The hierarchy is shown back in Figure 5.

ComponentInfo only maintains information about the component's name and type.

```
public class ComponentInfo
{
  protected String name = null;
  protected ComponentType type;

  public ComponentInfo(String nm, ComponentType ct)
  { name = nm;
    type = ct;
  }

  public String getName()
  {  return name;  }

  public ComponentType getType()
  {  return type;  }

  public String toString()
  {  return "\"" + name + "\" " + type; }

}  // end of ComponentInfo class
```

The ComponentType is the enumeration:

```
enum ComponentType {
    BUTTON, SLIDER, DIAL
}
```

In GestureGUIPanel, a ComponentInfo object is created in the constructor:

```
compInfo = new ComponentInfo(guiLabel, type);
```

The label and type are supplied as arguments to GestureGUIPanel().

4.2. Rendering the GGUI Component

Since GestureGUIPanel is a JPanel, its rendering is managed by its paintComponent() method, which examines its current gesture state to decide what to draw:

```
// globals
private String guiLabel;
private Font labelFont;
private int xLabel, yLabel;   // drawing position for label

private GestureState gState;


public void paintComponent(Graphics g)
// render the current state, and the component label
{
  super.paintComponent(g);

  // draw one of the states
  switch (gState) {
    case INACTIVE:
      drawInactive(g); break;
```



```
    case ACTIVE:
      drawActive(g); break;
    case PRESSED:
      drawActive(g);
      drawPressed(g);    // pressed is drawn on top of active
      break;
  }

  // draw the label
  g.setColor(Color.WHITE);
  g.setFont(labelFont);
  g.drawString(guiLabel, xLabel, yLabel);
} // end of paintComponent()
```

There are separate draw methods for the inactive, active, and pressed states, which are likely to be overridden by subclasses of GestureGUIPanel. Their definitions in GestureGUIPanel are fairly simple:

```
// globals
private static final int PRESSED_SIZE = 30;
       // size of circle drawn in the pressed state

private BufferedImage inactiveIm, activeIm;
private int width, height;
      // of inactive/active images, and the panel also


public void drawInactive(Graphics g)
{  g.drawImage(inactiveIm, 0, 0, null);  }


public void drawActive(Graphics g)
{  g.drawImage(activeIm, 0, 0, null);  }


public void drawPressed(Graphics g)
{
  int x = (width - PRESSED_SIZE)/2;
  int y = (height - PRESSED_SIZE)/2;
  g.setColor(Color.YELLOW);   // draw a yellow circle in center
  g.fillOval(x, y, PRESSED_SIZE, PRESSED_SIZE);
}  // end of drawPressed()
```

The inactive and active images are assumed to be as big as the component's panel, and are stored in inactiveIm and activeIm when the object is instantiated. The default pressed image is a small circle drawn at the center of the panel, drawn on top of the active image by paintComponent().

5. The Button GGUI Component

The ButtonPanel classes inherits GestureGUIPanel, and so utilizes standard behavior for updating and drawing. The class defines button-specific images for the inactive, active, and pressed states, and redefines the update and drawing behaviors for the pressed state. The button version of the state diagram is shown in Figure 7.

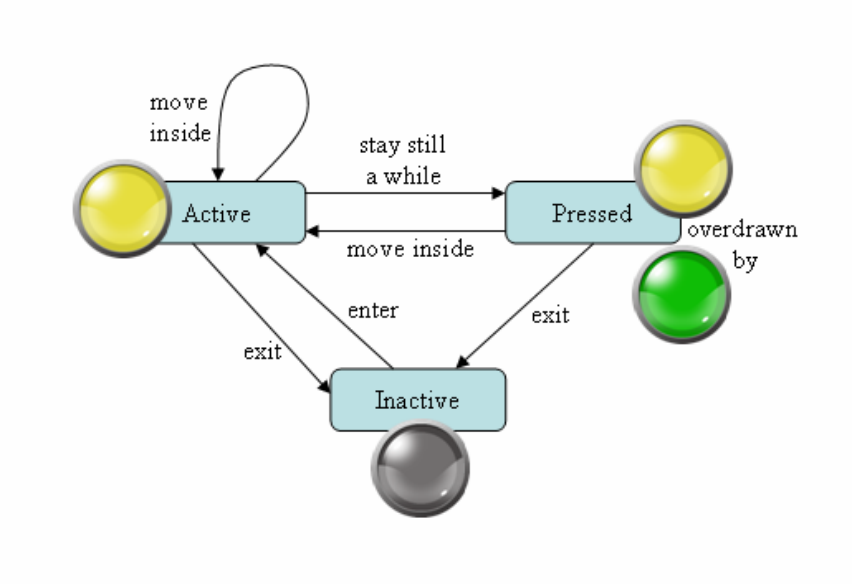

Figure 7. State Diagram for the Button GGUI Component.

The inactive, active, and pressed states are represented by different colored buttons. The inactive and active versions are loaded by code inherited from GestureGUIPanel, but the green, pressed button is managed directly by the ButtonPanel class. This separation is shown in the ButtonPanel constructor:

```
// globals
private static final String INACTIVE_FNM = "buttonInactive.png";
                            // gray version of button
private static final String ACTIVE_FNM = "buttonActive.png";
                            // yellow version
private static final String PRESSED_FNM = "buttonPressed.png";
                            // green version

private BufferedImage pressedIm = null;
private ButtonInfo buttonInfo = null;


public ButtonPanel(String label, TestGestureGUIs top)
{
  super(label, ComponentType.BUTTON, INACTIVE_FNM, ACTIVE_FNM, top);
  pressedIm = loadImage(PRESSED_FNM);
  buttonInfo = new ButtonInfo(label);
}  // end of ButtonPanel()
```

The Constructor also creates a ButtonInfo object, a subclass of ComponentInfo that sets the component type to be ComponentType.BUTTON. updatePressed() is redefined to send this button information to the top-level:

```
public void updatePressed()
{  topLevel.announcePress(buttonInfo);  }
```

drawPressed() is defined to draw the pressed button image rather than the default yellow circle.


```
public void drawPressed(Graphics g)
{  g.drawImage(pressedIm, 0, 0, null); }
```

This approach introduces a small amount of inefficiency since GestureGUIPanel always draws the active image before the pressed graphic. This is a waste of time since the green button completely hides the yellow one.

6. The Dial GGUI Component

The dial GGUI component utilizes inactive and active background images, and additional graphics of the dial's knob and a tiny hand, as shown in Figure 8.

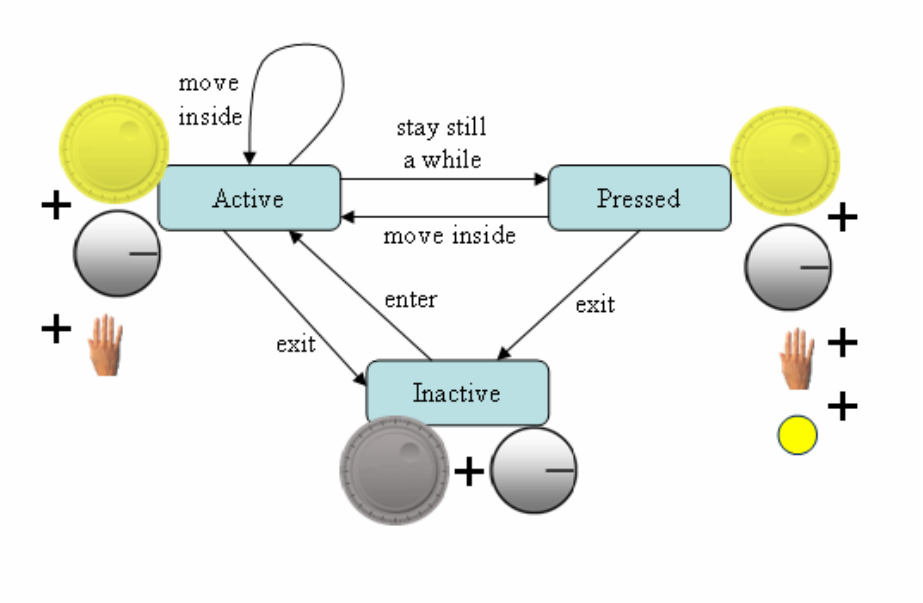

Figure 8. State Diagram for the Dial GGUI Component.

The DialPanel constructor loads the various images, and creates a DialInfo state object:

```
// globals: dial and knob details
private static final String DIAL_INACTIVE = "dialInactive.png";
                              // gray dial background
private static final String DIAL_ACTIVE = "dialActive.png";
                              // yellow dial background

private static final String DIAL_KNOB = "knob.png";
private static final String HAND = "hand.png";

private BufferedImage handImage = null;
private BufferedImage knobImage = null;
private int knobWidth, knobHeight;

private int angle; // in degrees

public DialPanel(String label, TestGestureGUIs top)
```



```
{
  super(label, ComponentType.DIAL, DIAL_INACTIVE, DIAL_ACTIVE, top);

  // load hand and knob images
  handImage = loadImage(HAND);
  knobImage = loadImage(DIAL_KNOB);
  knobWidth = knobImage.getWidth();
  knobHeight = knobImage.getHeight();

  angle = 0;    // initial angle of knob line to +x-axis
  dialInfo = new DialInfo(label, angle);
}  // end of DialPanel()
```

The DialInfo object includes angle information about the rotation of the knob's dial line relative to the +x axis.

6.1. Updating the Dial

Updating the dial is more complicated than the button since the dial's knob rotates due to movements of the user's hand point. The implementation is illustrated by Figure 9.

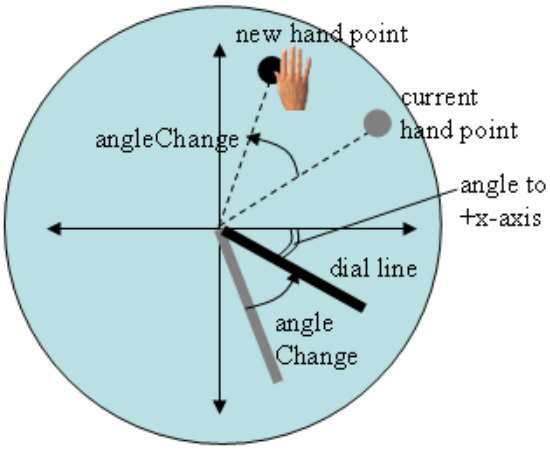

Figure 9. Rotating the Dial Knob.

The tiny hand image acts as feedback about where the user's hand is located on the dial. The hand doesn't need to be over any particular spot in order to turn the knob. Instead, the knob's angle change is calculated by comparing the new hand point to the current position relative to the knob's center. The angle change is used to rotate the knob so the dial line turns.

Even though the dial line rotates in the active state, its angle to the +x axis isn't sent to the top-level until the dial enters the pressed state, which requires the user to not move their hand for 2 or more seconds. The pressed state is signaled by a yellow circle appearing at the center of the dial.

The angle is measured in degrees in the range -180 to 180, with plus values going counterclockwise, and negative values clockwise. Figure 10 shows two examples of a pressed dial.

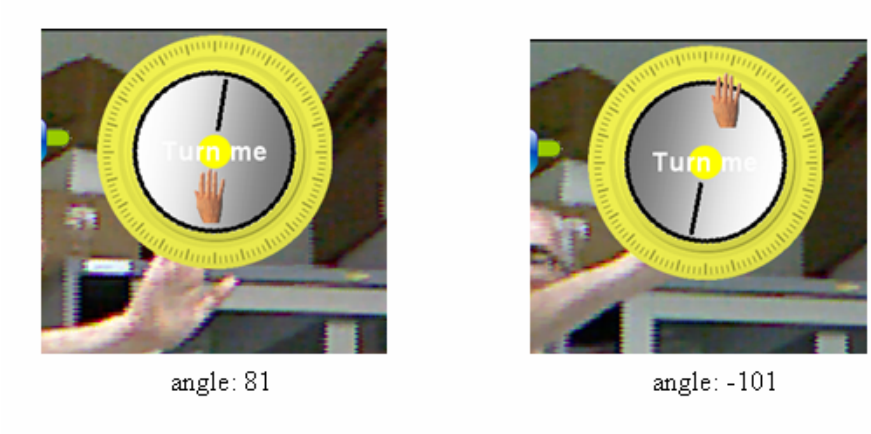

Figure 10. Two Pressed Dials.

The dial on the left of Figure 10 sends the top-level an angle of 81 degrees relative to the +x axis, while the one on the right reports -101 degrees. Note that the user's hand location (denoted by the hand image) does not need to be near the dial line in order to turn the knob, only within the screen area occupied by the dial.

The hard work of knob rotation is carried out by a redefined version of updateActive() in DialPanel:

```
// global
private int angle; // in degrees


public void updateActive(Point newPt)
{ updateKnobAngle(newPt);
  super.updateActive(newPt);    // make new hand pt the current pt
}


private void updateKnobAngle(Point newPt)
/* Calculate new angle of dial line relative to the +x axis,
   by adding angle change between new hand position and
   current hand point.  */
{
  Point currPt = getPoint();   // current hand position
  if (currPt == null)
    return;

  // calculate current and new hand posn relative to dial's center
  // (which is also the knob's center
  int xCurr = currPt.x - getWidth()/2;
  int yCurr = -1*(currPt.y - getHeight()/2);  //so +y is up screen

  int xNew = newPt.x - getWidth()/2;
  int yNew = -1*(newPt.y - getHeight()/2);

  int angleChange = signedAngleChg(xCurr, yCurr, xNew, yNew);
  angle -= angleChange;   // since clockwise is negative change
  if (angle > 180)
    angle -= 360;
  else if (angle < -180)
    angle += 360;
```



```
}  // end of updateKnobAngle()
```

updateKnobAngle() calculates the angle change shown in Figure 9, by comparing the positions of the current hand point (stored in currPt) and the new hand point (in newPt). The signedAngleChg() function uses Math.atan2() to calculate the angles of the two points relative to the +x axis, and returns their difference as an integer degree value:

```
private int signedAngleChg(int xCurr, int yCurr,
                                          int xNew, int yNew)
{
  double angleChg = Math.atan2(yNew, xNew) -
                    Math.atan2(yCurr, xCurr);
  if (angleChg > Math.PI)
    angleChg -= 2*Math.PI;
  else if (angleChg < -Math.PI)
    angleChg += 2*Math.PI;

  return (int) Math.round( Math.toDegrees(angleChg) );
}  // end of signedAngleChg()
```

The angle change is added to the angle back in updateKnobAngle(), and a call to GestureGUIPanel. updateActive() updates the current hand point to be the new hand position.

updatePressed() is also overridden but only to update and send dial information to the top-level:

```
// globals
private int angle;
private DialInfo dialInfo;


public void updatePressed()
{ dialInfo.setAngle(-angle);   // due to mirroring
  topLevel.announcePress(dialInfo);
}
```

The angle stored in the DialInfo object is negated since the Kinect camera image is mirrored (by Context.setGlobalMirror() in TestGestureGUIs.configKinect()).

6.2. Drawing the Dial

The rendering of the active dial involves three images (as shown in Figure 8): the background active image, the rotated knob, and a hand positioned at the current hand point.

The background image drawing is dealt with by the inherited GestureGUIPanel.drawActive(), so DialPanel's drawActive() only concerns itself with the knob and hand.

```
// global
private int angle;
```



```
private BufferedImage handImage;


public void drawActive(Graphics g)
{
  super.drawActive(g);    // draw full-size active image
  drawKnob(g, angle);

  // draw hand image centered at (x, y)
  int x, y = 0;
  Point currPt = getPoint();     // current hand position
  if (currPt == null) {  //if no point, then place hand in center
    x = (getWidth() - handImage.getWidth())/2;
    y = (getHeight() - handImage.getHeight())/2;
  }
  else {
    x = currPt.x - handImage.getWidth()/2;
    y = currPt.y - handImage.getHeight()/2;
  }
  g.drawImage(handImage, x, y, null);
}  // end of drawActive()
```

Positioning the hand is a matter of accessing the current hand point (which was updated at the end of updateActive()). One issue is that the dial is occasionally redrawn before the hand point has a value, so the hand is then placed at the dial's center.

drawKnob() assumes that the center of the knob is located at the center of the image, and so can be rotated around that point.

```
// globals
private BufferedImage knobImage = null;
private int knobWidth, knobHeight;


private void drawKnob(Graphics g, int angle)
{
  Graphics2D g2d = (Graphics2D) g;
  g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                       RenderingHints.VALUE_ANTIALIAS_ON);

  AffineTransform origTF = g2d.getTransform();
  AffineTransform newTF = (AffineTransform)(origTF.clone());

  // center of rotation is the center of the dial image
  newTF.rotate( Math.toRadians(angle), getWidth()/2,
                                       getHeight()/2);
  g2d.setTransform(newTF);

  // draw knob image centered in dial image
  int x = (getWidth() - knobWidth)/2;
  int y = (getHeight() - knobHeight)/2;
  g2d.drawImage(knobImage, x, y, null);

  g2d.setTransform(origTF);
}  // end of drawKnob()
```

Anti-aliasing is turned on so the rotated dial line looks better

7. The Slider GGUI Component

Featuritis crept into my code for SliderPanel, since its possible to supply a boolean in its constructor which specifies a horizontal or vertical orientation. A vertical slider is useful when the GGUI components are lined up along a side of the screen, as in Figure 11.

Figure 11. GGUI Component Aligned to the West.

Both the horizontal and vertical slider versions use the same GGUI behavior model from GestureGUIPanel, but 'dress' the slider with different images. The state diagram for the horizontal slider is shown in Figure 12.

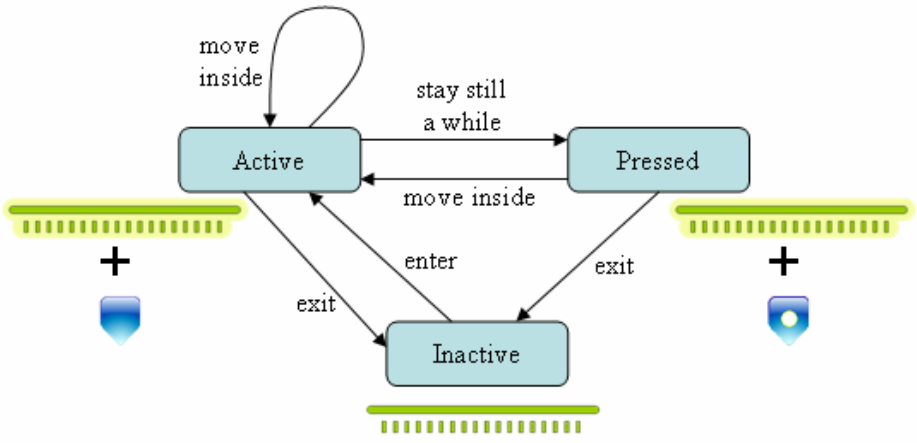

Figure 12. State Diagram for the Horizontal Slider.

The state diagram for the vertical slider is given in Figure 13.

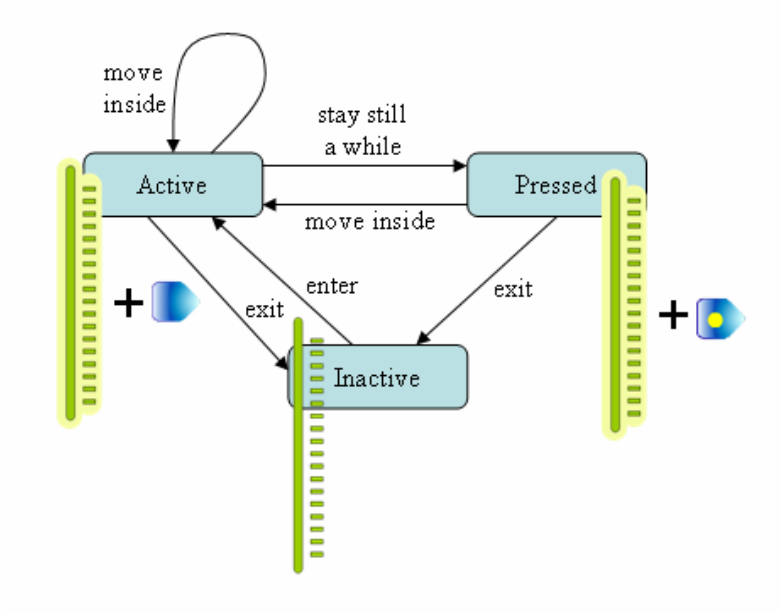

Figure 13.. State Diagram for the Vertical Slider.

The pressed state doesn't draw a yellow circle in the center of the GGUI, instead utilizing a version of the slider tab with a yellow dot in its middle.

7.1. Initializing the Slider

A large part of the SliderPanel constructor is taken up with deciding which images to load (i.e. the horizontal or vertical ones). SliderPanel delegates the loading of the background inactive and active images to GestureGUIPanel, but takes charge of the slider tab itself, since it needs to be moved to follow the user's hand.

7.2. Updating the Slider

updateActive() is redefined so that a new hand position moves the slider tab. It follows the hand position but limited to the x-axis on the slider's ruler. Similarly, the vertical tab is limited to the y-axis.

```
// globals
// limits of the horizontal slider
private static final int MIN_HORIZ_POS = 55;    // left
private static final int MAX_HORIZ_POS = 440;   // right

// limits of the vertical slider
private static final int MIN_VERT_POS = 55;     // top
private static final int MAX_VERT_POS = 447;    // bottom

private boolean isHorizontal;    // orientation of slider (and tab)
private int tabWidth, tabHeight;
private int xTabPos, yTabPos;   // current position of the tab


public void updateActive(Point newPt)
{ updateTabPos(newPt);
```



```
  super.updateActive(newPt);   // make new hand pt the current pt
}


private void updateTabPos(Point newPt)
/* Use the new hand position to update the tab's position, but
   only within the limits for the slider. */
{
  xTabPos = newPt.x;
  yTabPos = newPt.y;

  // apply slider limits to (xTabPos, yTabPos)
  if (isHorizontal) {
    if (xTabPos < (MIN_HORIZ_POS - tabWidth/2))
      xTabPos = MIN_HORIZ_POS - tabWidth/2;
    else if (xTabPos > (MAX_HORIZ_POS - tabWidth/2))
      xTabPos = MAX_HORIZ_POS - tabWidth/2;
    yTabPos = getHeight()/2 - tabHeight/2;
  }
  else {   // slider is vertical
    if (yTabPos < (MIN_VERT_POS - tabHeight/2))
      yTabPos = MIN_VERT_POS - tabHeight/2;
    else if (yTabPos > (MAX_VERT_POS - tabHeight/2))
      yTabPos = MAX_VERT_POS - tabHeight/2;
    xTabPos = getWidth()/2 - tabWidth/2;
  }
}  // end of updateTabPos()
```

updateTabPos() limits the movement of the tab in the x-direction if the slider is horizontal, or in the y-direction if the slider is vertical. The limits (the constants MIN_HORIZ_POS, MAX_HORIZ_POS, MIN_VERT_POS, and MAX_VERT_POS) were arrived at by me measuring the active slider images.

The updatePressed() method is extended to calculate the range data sent in the SliderInfo object.

```
// global
private SliderInfo sliderInfo;


public void updatePressed()
{ sliderInfo.setRange( calcRange() );
  topLevel.announcePress(sliderInfo);
}
```

calcRange() converts the current tab position into an integer between 0 and 100, with 0 being the left of the horizontal slider, and the top of the vertical one. For example, the pressed slider tab in Figure 14 returns a range value of 83.

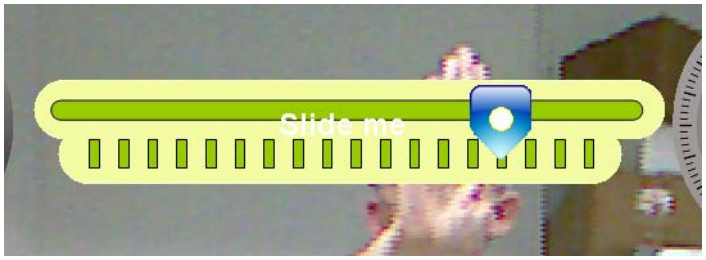

Figure 14. A Pressed Slider.

7.3. Drawing the Slider

All the draw methods need to be overridden in SliderPanel, but only so that the tab can be drawn on top of the background.

```
// globals
// active and pressed version of the tab image
private BufferedImage tabIm = null;        // active
private BufferedImage tabOnIm = null;      // pressed
private int xTabPos, yTabPos;   // current position of the tab


public void drawInactive(Graphics g)
// draw tab on top of inactive image
{ super.drawInactive(g);
  g.drawImage(tabIm, xTabPos, yTabPos, null);
}


public void drawActive(Graphics g)
// draw tab on top of active image
{ super.drawActive(g);
  g.drawImage(tabIm, xTabPos, yTabPos, null);
}


public void drawPressed(Graphics g)
// pressed state is represented by the pressed tab
{  g.drawImage(tabOnIm, xTabPos, yTabPos, null);  }
```

The drawPressed() method uses the pressed version of the tab, which is stored in the tabOnIm global.

