Chapter NUI-9. Using Multiple Mice

Multitouch envy is a dangerous emotion for laptop and PC users, and tends to make us forget the excessive effort required to keep lifting your hand (or even both hands, heavens forbid!) up to the screen to move, resize and rotate stuff.

One aspect of a multitouch interface might be helpful – the ability to have more than one cursor active on the screen at once. Imagine plugging two mice into a PC and have them appear on screen as *two* cursors. In fact, most laptops users already employ two input devices, a mouse and a touch pad, but linked to a single, jointly shared, cursor.

Multiple mice (by which I really mean multiple on-screen cursors) are possible in Java by interacting with them via the JInput library (https://jinput.dev.java.net/). I'll go through three examples in this chapter:

- MiceReporter, which lists all the 'mouse controllers' detected by JInput; JInput's definition of a mouse is quite generous, so MiceReporter will list touch pads and Thinkpad-style pointing sticks as well.
- MouseController, which monitors a mouse selected by the user, printing out changes to its buttons, position, and middle wheel.
- Multitouch, a graphical application, shown in Figure 1, which lets the user move, resize and rotate images with two 'finger tips'. Each finger tip is a cursor in the shape of a finger print, and is controlled by a different mouse (or a touch pad and mouse on my laptop).

Figure 1. The Multitouch Application.

1. JInput

JInput (http://java.net/projects/jinput) is a cross-platform API for the discovery and polling of input devices, ranging from the familiar (keyboard, mouse, touchpad) to more fun varieties (e.g. joysticks and game pads). You can find more details on reading input from a game pad in "Building a Game Pad Controller with JInput" at http://fivedots.coe.psu.ac.th/~ad/jg2/ch11/.

The range of supported devices depends on the underlying OS. On Windows, JInput can employ DirectInput, on Linux it relies on /dev/input/event* device nodes, and there's a Mac OS X version as well.

I downloaded the latest nightly build from http://java.net/projects/jinput, and extracted the three files related to Windows 32-bit: jinput-windows.jar (the Java part of the API) and jinput-dx8.dll and jinput-raw.dll (the OS-level parts). Another essential download is the API documentation, which hasn't been updated since December 2007.

A good source of information is the JInput javagaming.org forum at http://www.java-gaming.org/boards/jinput/27/view.html.

2. Listing All the Mouse Controllers

The simplest way of using JInput is to have it list the components of all the device controllers it can see. My MiceReporter application restricts itself to mouse controllers, and only reports on standard mouse components: the left, middle, and right buttons, the (x, y) position, and the wheel.

Typical output looks like:

```
ID 0; name HID-compliant mouse
LEFT: Left - Left - absolute - digital - 0.0
MIDDLE: Middle - Middle - absolute - digital - 0.0
RIGHT: Right - Right - absolute - digital - 0.0
X: x - x - relative - analog - 0.0
Y: y - y - relative - analog - 0.0
Wheel: z - z - relative - analog - 0.0

ID 2; name Microsoft PS/2 Mouse
LEFT: Left - Left - absolute - digital - 0.0
MIDDLE: Middle - Middle - absolute - digital - 0.0
RIGHT: Right - Right - absolute - digital - 0.0
X: x - x - relative - analog - 0.0
Y: y - y - relative - analog - 0.0
Wheel: z - z - relative - analog - 0.0
```

The most useful information for later are the ID numbers of the two controllers (0 and 2 in the example above). I'll use an ID to select a controller in my MouseController application.

The MiceReporter constructor iterates through all the detected controllers, printing information about those of the MOUSE type:

```
ControllerEnvironment ce =
          ControllerEnvironment.getDefaultEnvironment();
```



```
Controller[] ca = ce.getControllers();
Controller.Type type;
for (int i = 0; i < ca.length; i++) {
  type = ca[i].getType();
  if (type == Controller.Type.MOUSE) {
    System.out.println("\nID " + i + "; name " + ca[i].getName());
    reportMouse((Mouse)ca[i]);
  }
}
```

Each ID corresponds to the index position of the controller in JInput's array of Controller objects.

reportMouse() is passed a Controller cast to the Mouse type, which offers mouse-specific access methods for the device's components:

```
private void reportMouse(Mouse mouseCtrl)
{
  System.out.println("LEFT: " +
                       getComponentInfo( mouseCtrl.getLeft()));
  System.out.println("MIDDLE: " +
                       getComponentInfo( mouseCtrl.getMiddle()));
  System.out.println("RIGHT: " +
                       getComponentInfo( mouseCtrl.getRight()));

  System.out.println("X: " + getComponentInfo( mouseCtrl.getX()));
  System.out.println("Y: " + getComponentInfo( mouseCtrl.getY()));
  System.out.println("Wheel: " +
                       getComponentInfo( mouseCtrl.getWheel()));
}  // end of reportMouse()
```

The Mouse get methods return JInput Component objects representing the left, middle, and right buttons, the (x, y) movement, and the wheel rotation. A Mouse object can detect up to five buttons, but I'm restricting myself to the standard three.

JInput Component objects can model a wide range of device elements, including buttons, sliders, sticks, or dials, which supply different kinds of data when polled. The Component class contains several methods for detecting the component's type, which allows us to correctly interpret the input data. For example, data coming from a button will be absolute and digital (the integers 0 or 1) which means the button is released or pressed.

getComponentInfo() displays a component's data types, along with its name and identifier (which are usually the same).

```
private String getComponentInfo(Component c)
{
  if (c == null)
    return "none";
  else
    return (c.getName() + " - " + c.getIdentifier() +
          " - " + (c.isRelative() ? "relative" : "absolute") +
          " - " + (c.isAnalog() ? "analog" : "digital") +
          " - " + c.getDeadZone() );
}  // end of getComponentInfo()
```


The 'dead zone' field is the amount that input data can vary between pollings before being considered a significant change in value. The typical value, 0, means that any change is reported.

3. Collecting Data from a Mouse Controller

It's all very well learning from MiceReporter that there are two mouse controllers (with IDs 0 and 2), but what about actually obtaining some data from them? JInput uses polling to update a controller (the Controller.poll() method). The newly revised values stored in the controller's components can be accessed using calls to Component.pollData().

On Windows, JInput allows a mouse controller to be observed via DirectX or directly through a 'raw input' mode. The difference is that raw input allows multiple mice to be monitored at the same time, a feature which I'll need in my Multitouch application, so will use in this section also. A minor downside is that raw input requires the mouse to be associated with a window (i.e. a JFrame). As a consequence, my MouseController class is called from a TestMouse class which creates an empty JFrame. To keep things simple, data polled from the controller's components are printed to standard output, rather than to the JFrame. Figure 2 shows some typical output:

```
ID: 2; name: "Microsoft PS/2 Mouse"
x: -9
y: 1
x: -20
y: -8
y: -2
x: -23
y: -10
y: 2
x: -1
Left: pressed
Left: released
Right: pressed
Right: released
Left: pressed
Left: released
x: -1
y: 1
x: -37
```

The example shows the monitoring of the mouse controller whose ID is 2; changes to its three buttons, its (x, y) moves, and the wheel rotations are printed.

After I'd experimented with several mice and my laptop's touch pad, it became clear that the controller's x- and y- movements are positive when a mouse moves to the right and down. and the wheel change is positive when it's rotated forwards. When the mouse isn't moved, the x-, y-, and wheel components report 0 values. Buttons return 1 or 0 when pressed or released.

To reduce the flood of data a little, my MouseController class only prints the buttons' states when they change (i.e. when pressed or released), and numerical data when it

differs from 0. If I didn't impose these restrictions, then every call to poll() (which occurs every 40 milliseconds) would be followed by the printing of the all the button states and x-, y-, and wheel data.

The TestMouse class creates a MouseController object, and starts a thread which repeatedly updates the object.

```
// global
private MouseController mc;


public TestMouse(int id)
{
  super("TestMouse");

  mc = new MouseController(id);

  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  setSize(200, 200);
  setVisible(true);

  new Thread(new Runnable() {
    public void run() {
      update(mc);
    }
  }).start();
}  // end of TestMouse()
```

The MouseController object requires a controller ID so it can select a mouse controller to poll. TestMouse subclasses JFrame so the controller is attached to a window, and so can be monitored by JInput's raw input mode.

The update() method calls MouseController.update() and then sleeps for a short time before repeating.

```
// global
private static final int DELAY = 40;  // ms  (update interval)


private void update(MouseController mc)
{
  while (true) {
    if (!mc.update()) {
      System.out.println("Mouse Controller no longer valid");
      System.exit(0);
    }
    try {
      Thread.sleep(DELAY);      // wait a while
    }
    catch (Exception ex) {}
  }
}  // end of update()
```

The MouseController() constructor uses JInput to select the controller with the specified ID, and casts it into a mouse controller.


```
// globals
private Mouse mouseCtrl = null;


public MouseController(int id)
{
  ControllerEnvironment ce =
              ControllerEnvironment.getDefaultEnvironment();
  Controller[] ca = ce.getControllers();
  if (ca.length == 0) {
    System.out.println("No controllers found");
    System.exit(0);
  }

  if ((id < 0) || (id >= ca.length)) {
    System.out.println("Supplied index out of range (0-" +
                                          (ca.length-1) + ")");
    System.exit(0);
  }

  Controller.Type type = ca[id].getType();
  if (type != Controller.Type.MOUSE) {
    System.out.println("Controller[" + id + "] is not a mouse");
    System.exit(0);
  }

  mouseCtrl = (Mouse) ca[id];
   System.out.println("ID: " + id + "; name: \"" +
                                  mouseCtrl.getName() + "\"");
}  // end of MouseController()
```

MouseController.update() polls the controller, and reports the current values of its components:

```
// globals
private boolean isLeftPressed = false;
private boolean isMiddlePressed = false;
private boolean isRightPressed = false;


public boolean update()
{
  if (!mouseCtrl.poll())
    return false;

  isLeftPressed = updatePress(mouseCtrl.getLeft(), isLeftPressed);
  isMiddlePressed =
              updatePress(mouseCtrl.getMiddle(), isMiddlePressed);
  isRightPressed = updatePress(mouseCtrl.getRight(), isRightPressed);

  showMove(mouseCtrl.getX());
  showMove(mouseCtrl.getY());
  showMove(mouseCtrl.getWheel());

  return true;
}  // end of update()
```


The three global booleans for the button pressings allow updatePress() to distinguish between when a button state flips between pressed and released and when it stays unchanged after a poll.

```
private boolean updatePress(Component button, boolean isPressed)
{
  if (button == null)
    return isPressed;    // no change

  String name = button.getName();
  float val = button.getPollData();

  if (isPressed && (val == 0)) {  // button released
    isPressed = false;
    System.out.println(name + ": released");
  }
  else if (!isPressed && (val == 1)) {  // button pressed
    isPressed = true;
    System.out.println(name + ": pressed");
  }

  return isPressed;
}  // end of updatePress()
```

showMove() prints the component value for the x-, y-, and wheel moves, but only when they aren't 0.

```
private void showMove(Component mover)
{
  if (mover != null) {
    int step = (int) mover.getPollData();
    if (step != 0)   // don't report 0 (i.e. no movement)
      System.out.println(mover.getName() + ": " + step);
  }
}  // end of showMove()
```

4. Implementing a Multitouch Panel with Multiple Mice

Multitouch.java creates a full-screen undecorated window with randomly placed images. The pictures can be moved, rotated, and enlarged by using two mice (or a mouse and track pad) acting as 'finger tips', resulting in pictures arranged something like those in Figure 1.

Pressing on an image selects it so it can be moved. It's possible to move two images at once by pressing the finger tips on two different images (see Figure 2).

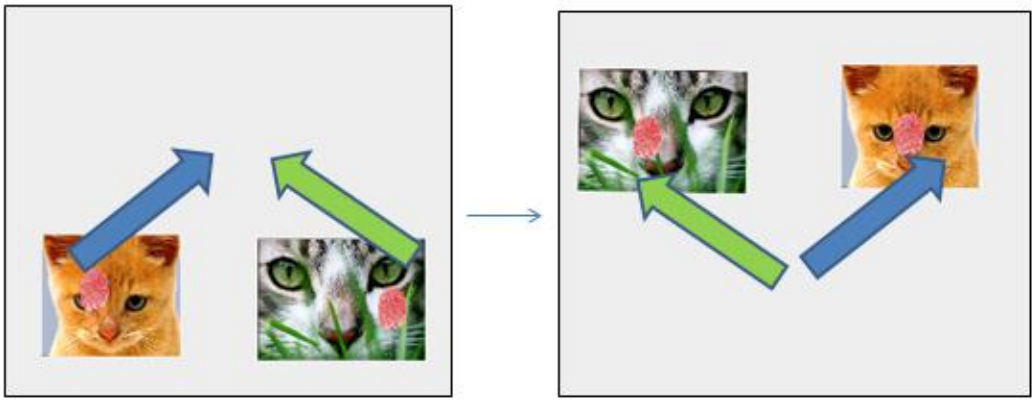

Figure 2. A Double Move in the Multitouch Application.

A pressed finger tip is denoted by a (slightly ghoulish) red finger print. The thick blue and green arrows in Figure 2 aren't part of the application; I added them to the figure to indicate the general direction of movement of the finger tips. The right hand picture shows how the cat pictures are changed by the finger tip movements.

Pressing both finger tips on the same image allows it to be enlarged by dragging apart the finger tips (as in Figure 3). An image can also be shrunk back to its original size by moving the two finger tips together.

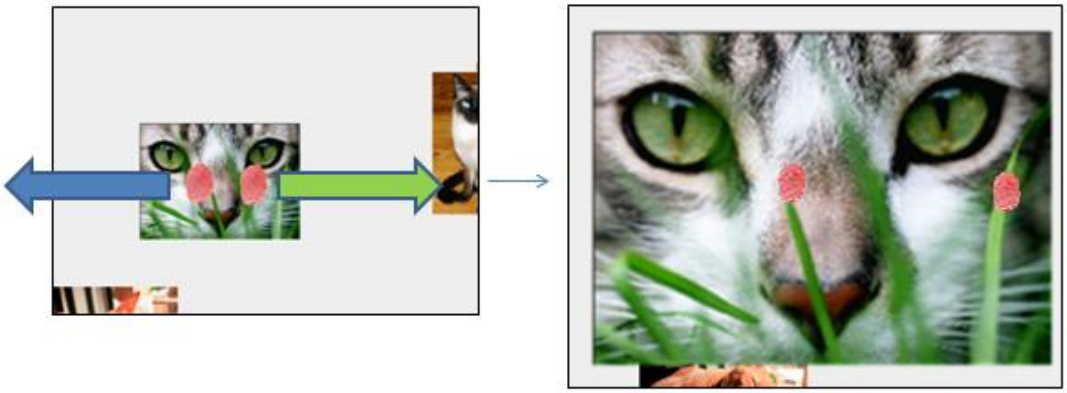

Figure 3. Enlarging an Image in the Multitouch Application.

Pressing one finger tip on the window, and the other on an image allows the image to be rotated around the first 'pivot' tip (see Figure 4). The image also rotates around its center.

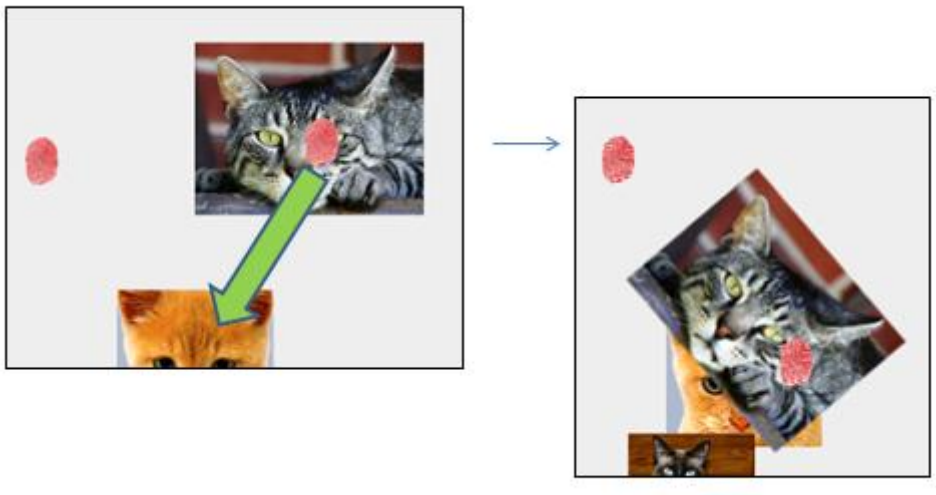

Figure 4. Rotating an Image in the Multitouch Application.

Even these simple operation raise some tricky user interface questions, and require various restrictions to increase their speed and reduce their memory usage.

For example, what should happen when the movement of two images at once causes the finger tips to cross over each other, as might happen in Figure 2? The problem is that when the finger tips are close together, they may both be over the same image, and so could be interpreted as an enlargement gesture (as in Figure 3).

Another problem is the tradeoffs between image quality, speed and memory usage during rotations and enlargements when new BufferedImage objects are created.

I'll discuss some of these issues when I talk about how the gesture operations are implemented later in the chapter.

The class diagrams for the Multitouch application are shown in Figure 5.

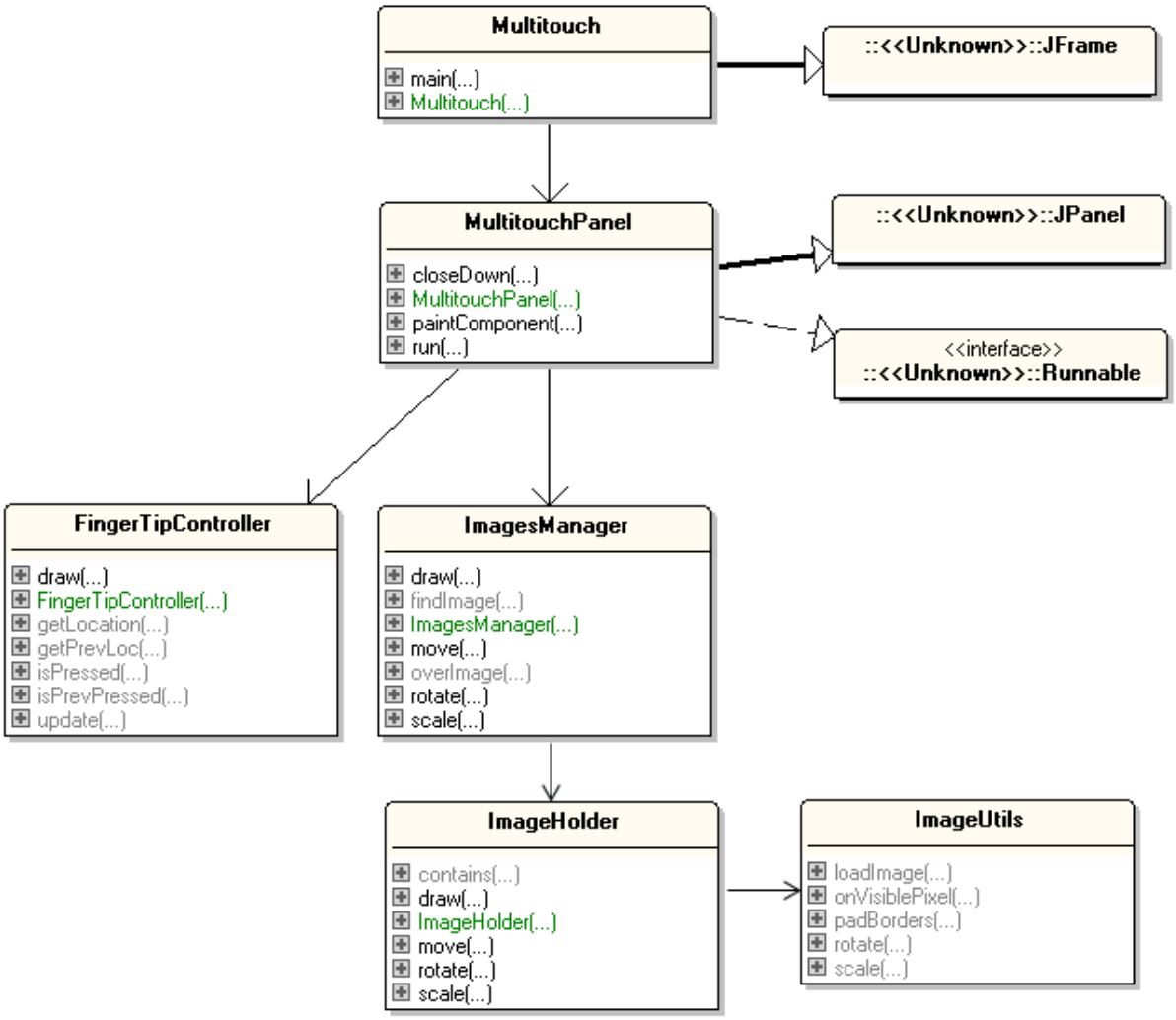

Figure 5. The Multitouch Class Diagrams.

Multitouch and MultitouchPanel implement the undecorated full-screen window as a JPanel inside a JFrame. MultitouchPanel is threaded so it can continuously execute a update/draw/sleep loop without causing the GUI to freeze.

FingerTipController holds the JInput code for 'finger tip' input (actually mouse or touch pad actions), and MultitouchPanel creates two FingerTipController objects to deal with the two tips. Each controller is a simplified version of the mouse controller I described in the previous section, only utilizing the left button and its (x, y) position.

The collection of images on screen are managed by an ImagesManager object, which passes move, enlarge, and rotate commands along to the relevant ImageHolder object. There's one ImageHolder instance for each image.

Image processing functionality, which includes loading, rotating and scaling an image, are handled by static methods in the ImageUtils class.

4.1. Creating the Finger Tip Controllers

MultitouchPanel contains an initFingerTips() method that uses JInput to retrieve a list of all the controllers, and iterates over it to find the mouse-type devices. The first found (which might be plugged in anywhere around a PC) is designated as the 'left' finger tip, and the next as the 'right' tip.

```
// globals
private FingerTipController leftTip, rightTip;


private void initFingerTips(int pWidth, int pHeight)
{
  ControllerEnvironment ce =
              ControllerEnvironment.getDefaultEnvironment();
  Controller[] ca = ce.getControllers();
  if (ca.length == 0) {
    System.out.println("No controllers found");
    System.exit(0);
  }

  // collect the IDs of all the mouse controllers
  int[] mouseIDs = new int[ca.length];
  int mouseCount = 0;
  System.out.println("Mouse Controllers:");
  for (int i = 0; i < ca.length; i++) {
    if (ca[i].getType() == Controller.Type.MOUSE) {
      System.out.println("  ID " + i + "; \"" +
                                 ca[i].getName() +"\"");
      mouseIDs[mouseCount++] = i;
    }
  }

  if (mouseCount < 2) {
    System.out.println("Not enough found (" + mouseCount
                                              + "); 2 needed");
    System.exit(0);
  }

  // left finger tip
  int idx = mouseIDs[0];
  System.out.println("\nInitializing mouse ID " + idx + "...");
  leftTip = new FingerTipController((Mouse)ca[idx], true,
                                              pWidth, pHeight);

  // right finger tip
  idx = mouseIDs[1];
  System.out.println("\nInitializing mouse ID " + idx + "...");
  rightTip = new FingerTipController((Mouse)ca[idx], false,
                                              pWidth, pHeight);
}  // end of initFingerTips()
```

The pWidth and pHeight arguments contains the panel's dimensions, which are also the size of the screen. The boolean argument passed to the FingerTipController constructor denotes whether the control is for the left finger tip or not (i.e. it's meant to be the right finger).

Executing the Application Loop

MultitouchPanel implements the application loop in a separate thread. Each iteration updates the controllers and images, drawing the finger tips and images, then sleeps for a period which aims to keep each cycle's total duration close to DELAY (25) milliseconds. This means that the controllers will be polled roughly at DELAY millisecond intervals.

```
// globals
private static final int DELAY = 25;  // ms  (polling interval)
private boolean isRunning = false;   // used to stop the loop


public void run()
{
  long duration;
  isRunning = true;

  while(isRunning) {
    long startTime = System.currentTimeMillis();
    update();
    duration = System.currentTimeMillis() - startTime;
    repaint();

    if (duration < DELAY) {
      try {
        Thread.sleep(DELAY-duration);
                    // wait until DELAY time has passed
      }
      catch (Exception ex) {}
    }
  }
  System.exit(0);
}  // end of run()
```

Updating the Finger Tips and Images

The update() method is arguably the most complicated part of the application since it updates the finger tip controllers, and then determines how changes to the finger tips should be interpreted as gestures that affect the screen images.

As I mentioned above, the three kinds of gestures are moves, rotations, and scaling. A move only requires a single finger tip be pressed onto an image, so the user can carry out double moves, where the left and right tips move two images at the same time. Enlargement needs both fingers to move apart over the same image (conversely, a move together triggers a reduction in the image size). Rotation requires one finger tip pressed on the blank screen to act as a pivot point, and the other finger tip pressed on the image that will rotate around that point.

Implicit in these gestures is the need to monitor the change to the finger tips over time. I do this by maintaining the current location of a finger tip and its previous position at the last update, and also its current and previous pressed state; this amounts to four pieces of data for each tip, which are obtained during the execution of the first half of the update() method, shown below:


```
// globals
private FingerTipController leftTip, rightTip;
private ImagesManager imsMan;


private void update()
{
  // update the finger tip controllers
  if (!leftTip.update()) {
    System.out.println("Left fingertip Controller no longer valid");
    System.exit(0);
  }

  if (!rightTip.update()) {
    System.out.println("Right fingertip Controller no longer valid");
    System.exit(0);
  }

  // get current and previous finger tip locs and pressed statuses
  Point prevLeftLoc = leftTip.getPrevLoc();
  Point leftLoc = leftTip.getLocation();
  Point prevRightLoc = rightTip.getPrevLoc();
  Point rightLoc = rightTip.getLocation();

  boolean prevLeftPressed = leftTip.isPrevPressed();
  boolean leftPressed = leftTip.isPressed();
  boolean prevRightPressed = rightTip.isPrevPressed();
  boolean rightPressed  = rightTip.isPressed();

  // gesture processing; explained next
  //   :
}  // end of update()
```

Single tip movements (of either the left finger tip or the right) are easily distinguished from other gestures since the others all require both fingers tips to be pressed:

```
  // in update()
  if (leftPressed && !rightPressed){    // left tip move
    int dx = leftLoc.x - prevLeftLoc.x;
    int dy = leftLoc.y - prevLeftLoc.y;
    imsMan.move(prevLeftLoc, dx, dy);
  }
  else if (!leftPressed && rightPressed) { // right tip move
    int dx = rightLoc.x - prevRightLoc.x;
    int dy = rightLoc.y - prevRightLoc.y;
    imsMan.move(prevRightLoc, dx, dy);
  }
    // move code, shown next
```

The movement will consist of the x- and y- offsets from the previous finger position to the current one. Note that the ImagesManager object (imsMan) will determine which ImageHolder object will be sent the move request based on the location passed to its move() method.

The three other possible gestures are for scaling, rotation, and a double move. These can be distinguished from each other by finding out what images (if any) the fingers are pressed down over.

```
  // in update()
  if (leftPressed && rightPressed &&
      prevLeftPressed && prevRightPressed) {  // both fingers pressed
    if (imsMan.overImage(prevLeftLoc) &&
          (imsMan.findImage(prevLeftLoc) ==
           imsMan.findImage(prevRightLoc))) {
      // tips on same image, so a scaling
      double prevDist =
                  Math.abs( prevLeftLoc.distance(prevRightLoc));
      double currDist = Math.abs( leftLoc.distance(rightLoc));
      if ((prevDist != 0) && (currDist != 0))
        imsMan.scale(prevLeftLoc, currDist/prevDist);
    }
    else { //tips not on same image, maybe a rotation or double move
      if (imsMan.overImage(prevLeftLoc) &&
          !imsMan.overImage(prevRightLoc)) {
        // rotation of left tip using right tip as screen pivot
        double angle = angleBetween(rightLoc, leftLoc, prevLeftLoc);
        imsMan.rotate(prevLeftLoc, rightLoc, angle);
      }
      else if (imsMan.overImage(prevRightLoc) &&
               !imsMan.overImage(prevLeftLoc)) {
        // rotation of right tip using left tip as screen pivot
        double angle = angleBetween(leftLoc, rightLoc, prevRightLoc);
        imsMan.rotate(prevRightLoc, leftLoc, angle);
      }
      else {   // tips pressed on two different images, so move both
        int dx = leftLoc.x - prevLeftLoc.x;
        int dy = leftLoc.y - prevLeftLoc.y;
        imsMan.move(prevLeftLoc, dx, dy);   // move left
        dx = rightLoc.x - prevRightLoc.x;
        dy = rightLoc.y - prevRightLoc.y;
        imsMan.move(prevRightLoc, dx, dy);   // move right
      }
    }
  }
}  // end of update()
```

This chunk of code relies on ImagesManager's overImage() and findImage() methods which return boolean and an ImageHolder reference respectively. The chosen operation results in calls to ImagesManager's scale(), rotate() or move() functions, which are supplied with a finger location and the relevant scale factor, rotation, or offset. ImagesManager uses the location to decide which image is affected.

The scale factor is calculated using two absolute distances between the fingers – the distance between the current tip locations is divided by the distance between their previous locations. The use of absolutes means that the scaling cannot be negative.

A rotation of the right finger tip around the left finger tip acting as the pivot is shown in Figure 6.

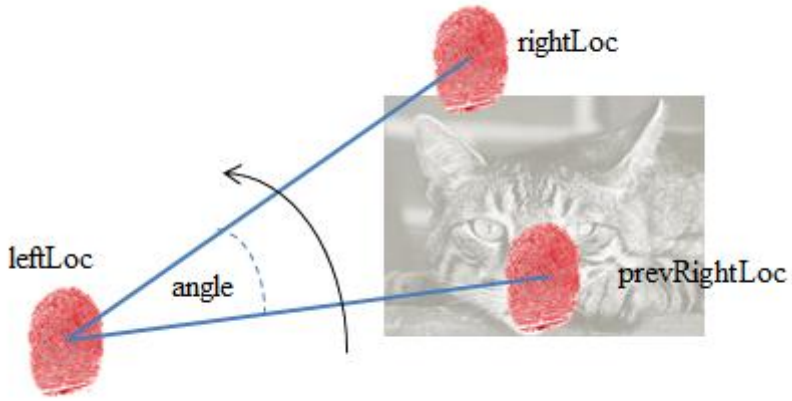

Figure 6. Rotation of Right Finger Tip.

The angle is calculated using some basic trigonometry in angleBetween(), bearing in mind that the y-axis runs down the screen and so the origin is at the top-left.

```
private double angleBetween(Point pivot, Point curr, Point prev)
/* return the angle between prev and curr points relative to the
   pivot point */
{
  return Math.atan2(curr.x - pivot.x, -(curr.y - pivot.y)) -
         Math.atan2(prev.x - pivot.x, -(prev.y - pivot.y));
}
```

The maths (for both 2D and 3D lines) is explained at http://www.euclideanspace.com/maths/algebra/vectors/angleBetween/

Redrawing the Panel

Panel rendering requires the drawing of the finger tips and images, tasks delegated to the relevant objects:

```
// globals
private FingerTipController leftTip, rightTip;
private ImagesManager imsMan;


public void paintComponent(Graphics g)
{
  super.paintComponent(g);
  imsMan.draw(g);         // draw images
  leftTip.draw(g);        // draw finger tips
  rightTip.draw(g);
} // end of paintComponent()
```


4.2. The Finger Tip Controller

The FingerTipController class implements a simple JInput mouse controller which monitors only left button presses and (x, y) moves. The idea is that 'finger tip' movement is implemented as mouse movement, and 'finger tip' presses are equivalent to the pressing of the left button.

An interesting question is how the other mouse components (i.e. the other two buttons and the wheel position) could augment this basic finger tip behavior. For instance, wheel moves could be interpreted as finger tip 'pressure', and the buttons could act as control keys to vary the meaning of the left button press. Alternatively, the buttons could be treated as additional fingers.

To help with gesture interpretation in MultitouchPanel, each FingerTipController stores the current and previous button press states and (x, y) positions. It also loads unpressed and pressed finger tip images, one of which is drawn at the current position when FingerTipController.draw() is called.

```
// globals
private static final String TIP_IMAGE = "blueFingerTip.png";
private static final String PRESSED_IMAGE = "redFingerTip.png";

private static final int TIP_OFFSET = 100;
  // start position offset from screen edge for finger tip image

// JInput controller and components
private Mouse mouseCtrl = null;
private Component leftBut, xMove, yMove;

// finger tip info: current and previous locs and pressed states
private Point prevLoc, location;
private boolean prevPressed = false;
private boolean isPressed = false;

private BufferedImage tipIm, pressedTipIm;
private int pWidth, pHeight, tipWidth, tipHeight;


public FingerTipController(Mouse mc, boolean isLeftTip,
                                      int pWidth, int pHeight)
{
  mouseCtrl = mc;
  System.out.print( (isLeftTip ? "Left " : "Right "));
  System.out.println("finger tip assigned to \"" +
                                  mouseCtrl.getName() + "\"");
  this.pWidth = pWidth;
  this.pHeight = pHeight;

  // initialize finger tip images
  tipIm = ImageUtils.loadImage(TIP_IMAGE);
  pressedTipIm = ImageUtils.loadImage(PRESSED_IMAGE);
  tipWidth = tipIm.getWidth();
  tipHeight = tipIm.getHeight();

  if (isLeftTip)
    location = new Point((TIP_OFFSET + tipWidth/2), (pHeight/2));
  else  // right tip
    location = new Point( (pWidth - TIP_OFFSET - tipWidth/2),
                                                    (pHeight/2));
```



```
  prevLoc = new Point(location.x, location.y);

  leftBut = mouseCtrl.getLeft();
  xMove = mouseCtrl.getX();
  yMove = mouseCtrl.getY();
}  // end of FingerTipController()
```

The initial starting position for a finger tip depends on whether it's a left or right tip, which is determined by the isLeftTip boolean passed to the constructor.

Updating a Finger Tip

The controller updates its JInput components and its finger tip states when MultitouchPanel calls FingerTipController.update():

```
// global
private Mouse mouseCtrl;


public boolean update()
// update the component values in the mouse
{
  boolean isValid = mouseCtrl.poll();
  if (!isValid)
    return false;

  updatePressed();
  updatePosition();
  return true;
}  // end of update()
```

updatePressed() changes the pressed state by modifying a global boolean, after backing up the old value.

```
// globals
private boolean prevPressed, isPressed;
private Component leftBut;


private void updatePressed()
{
  prevPressed = isPressed;  // backup

  float val = leftBut.getPollData();
  if (isPressed && (val == 0))        // button released
    isPressed = false;
  else if (!isPressed && (val == 1))  // button pressed
    isPressed = true;
}  // end of updatePressed()
```

updatePosition() employs a similar coding style to update the location of the finger tip using the x- and y- offsets read from the mouse components.

```
// globals
private Point prevLoc, location;
```



```
private Component  xMove, yMove;


private void updatePosition()
{
  prevLoc = new Point(location.x, location.y);  // backup

  int xNew = location.x + (int) xMove.getPollData();
  if (xNew < 0)   // check xNew is on screen
    xNew = 0;
  else if (xNew >= pWidth)
    xNew = pWidth-1;

  int yNew = location.y + (int) yMove.getPollData();
  if (yNew < 0)    // check yNew is on screen
    yNew = 0;
  else if (yNew >= pHeight)
    yNew = pHeight-1;

  location.setLocation(xNew, yNew);
}  // end of updatePosition()
```

4.3. Managing the Images

The ImagesManager class manages the on-screen images by storing each one in an ImageHolder object in a list.

```
// globals
private static final String[] imFnms = {
                  "cat2.png", "domestic-cat.png", "greeneyes.png",
                  "old-cat-sleeping.png", "SiameseCat.png"
                };

private ArrayList<ImageHolder> images;


public ImagesManager(int pWidth, int pHeight)
{
  images = new ArrayList<ImageHolder>();
  for(String fnm : imFnms)
    images.add( new ImageHolder(fnm, pWidth, pHeight));
}  // end of ImagesManager()
```

ImagesManager passes a move, scale, or rotation command to an image based on a supplied finger tip position. findImage() returns a reference to the matching ImageHolder object, or null. For example, ImagesManager.move() is:

```
public void move(Point tipPt, int dx, int dy)
// move the image at tipPt by dx and dy offsets
{
  ImageHolder imHolder = findImage(tipPt);
  if (imHolder != null)
    imHolder.move(dx, dy);
}  // end of move()
```


ImagesManager's rotate() and scale() methods are coded in a similar manner.

findImage() searches the list of images 'top-down', which means in reverse order from the end of the list to the start. This is due to the way that ImagesManager.draw() renders the list of images from the start to the end, resulting in the last image being drawn top-most on the screen. Not only does findImage() return a reference to an object, but also moves it to the end of the images list so it will be drawn on top. Unfortunately, this behavior introduces a concurrency issue which is fixed by synchronizing findImage() and draw(). findImage() is:

```
// global
private ArrayList<ImageHolder> images;


public synchronized ImageHolder findImage(Point tipPt)
{
  // go through images from 'top' to 'bottom'
  int i = images.size()-1;
  while (i >= 0) {
    if (images.get(i).contains(tipPt))
      break;
    i--;
  }

  if (i >= 0) {    // move found image to the 'top'
    ImageHolder imH = images.remove(i);
    images.add(imH);
    return imH;
  }
  else
    return null;
}  // end of findImage()
```

The concurrency problem is due to the fact that draw() is called by Java's GUI thread, while findImage() is executed by the separate MultitouchPanel thread. findImage() changes the images list, which may occur at the same time that draw() is iterating through the list. Therefore the two methods must be synchronized so that they can't execute concurrently. The draw() method is:

```
public synchronized void draw(Graphics g)
{
  for(ImageHolder imHolder : images)
    imHolder.draw(g);
}  // end of draw()
```

4.4. Representing an Image

An ImageHolder object represents an image which can be moved, scaled, and rotated by finger tips. The image is initially positioned at a random location on the screen, based on its center position. The current rotation angle and scale factor are maintained in globals.

```
// globals
```



```
private int pWidth, pHeight;          // panel size

private BufferedImage origIm, im;     // original and current images
private Point centerLoc;   // center point of current image
private int currAngle = 0;     // in degrees
private double currScale = 1.0;


public ImageHolder(String fnm, int pWidth, int pHeight)
{
  this.pWidth = pWidth;
  this.pHeight = pHeight;

  origIm = ImageUtils.padBorders( ImageUtils.loadImage(fnm));
  int w = origIm.getWidth();
  int h = origIm.getHeight();
  im = origIm;

  Random r = new Random();
  int x = r.nextInt(pWidth-w);
  int y = r.nextInt(pHeight-h);
  centerLoc = new Point(w/2+x, h/2+y);
}  // end of ImageHolder()
```

The borders of the loaded image are padded with transparent pixels in a call to ImageUtils.padBorders(). The new image is big enough so that the image can be rotated around its center without clipping any visible pixels. Figure 7 illustrates the change, which is based on using the length of the image's main diagonal as its new width and height.

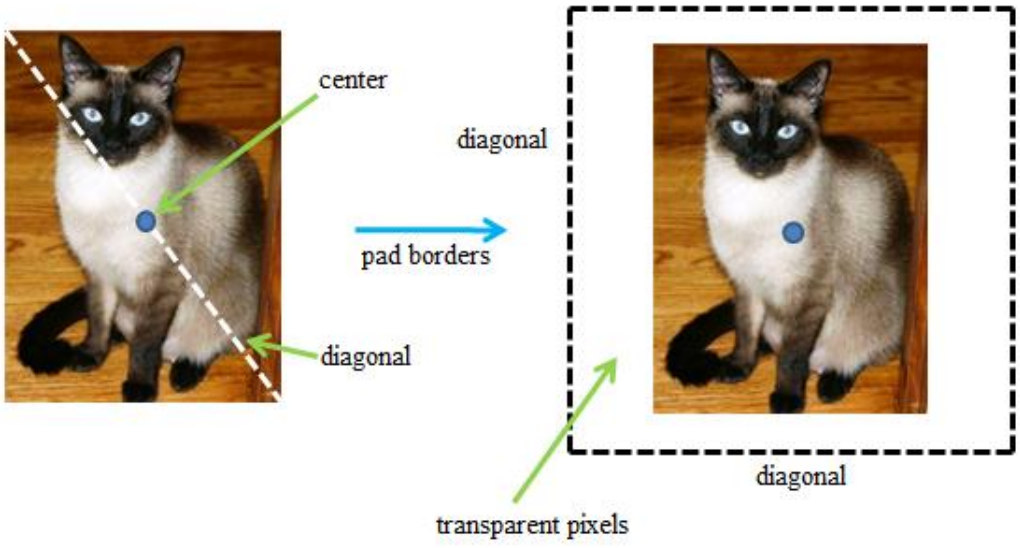

Figure 7. Padding an Image's Borders.

This change simplifies the rotation code, but at the expense of at least doubling the BufferedImage size!

Detecting Containment

An important test is to determine if a finger tip is over (or contained within) an image. This is a little trickier than it may first appear since rotation means that the test must distinguish between transparent border pixels and the visible pixels of the image. Figure 8 illustrates the problem – although the finger tip is inside the borders of the image, it's not over the actual image.

Figure 8. A Finger Tip Which is Not Over an Image.

ImageHolder's contains() method solves this problem by calling ImageUtils.onVisiblePixel() once the finger tip coordinate has been made relative to the image coordinates.

```
public boolean contains(Point pt)
// does this image contain the screen coordinate in pt?
{
  // calculate (x,y) relative to image coords
  int x = pt.x - (centerLoc.x - im.getWidth()/2);
  int y = pt.y - (centerLoc.y - im.getHeight()/2);
  return ImageUtils.onVisiblePixel(im, x, y);
         /* (x,y) is only on an image if it is over a visible pixel
            of the image */
}  // end of contains()
```

onVisiblePixel() accesses the pixel at the tip's (x, y) location and checks if its alpha channel is set to 0, meaning it is completely transparent.

```
// in ImageUtils class
public static boolean onVisiblePixel(BufferedImage im, int x, int y)
{
  if ((x < 0) || (y < 0) || (x >= im.getWidth()) ||
                            (y >= im.getHeight()))
    return false;     // since (x,y) not in range

  if (!im.getColorModel().hasAlpha())    // no alpha pixels
    return true;
```



```
  int alpha = (im.getRGB(x,y) >> 24) & 0xFF;
  return (alpha != 0);    // 0 == completely transparent
  }  // end of onVisiblePixel();
```

Efficient Rotation

Figure 6 correctly shows the rotation angle, but it's misleading about how the rotation is performed. A correct rotation would turn the image around the prevRightLoc point. To avoid clipping the image, it would be necessary to pad the image with transparent pixels, which isn't that complicated *for the first rotation*. The difficulty is that future rotations will require further padding of the image, but it isn't simple to determine by how much because the image contains an uneven border of transparent pixels. If no distinction is made between the two kinds of pixel then the additional padding needed prior to every rotation will keep making the image bigger until Java runs out of heap space for the BufferedImage object. Cropping away excessive transparent pixels is possible, but time-consuming.

I decided to avoid these problems by only letting an image rotate around its center. Now I only have to pad the image with transparent pixels once, at load time, and subsequent rotations can be done quickly. The downside is that some rotations look a bit unnatural since they aren't turning around the finger tip's position on the image.

The ImageHolder rotate() method implements a rotation in two stages – the center point of the image is rotated around the pivot, and then the image is rotated around its center:

```
// globals
private BufferedImage origIm, im;    // original and current images
private Point centerLoc;   // center point of current image
private int currAngle;    // in degrees
private double currScale;


public void rotate(Point pivot, double angle)
{
  // rotate the center point of the image around the pivot
  Point rotatedLoc = rotatePt(centerLoc, pivot, angle);
  Point newLoc = restrictPosition(rotatedLoc.x, rotatedLoc.y);
  centerLoc.setLocation(newLoc);

  int degAngle = (int)Math.round(Math.toDegrees(angle));
  currAngle = (currAngle + degAngle)%360;

  // rotate the image around its center point
  if ((currAngle == 0) && (currScale == 1))
    // reuse original image if unscaled
    im = origIm;
  else
    im = ImageUtils.rotate(im, angle);
}  // end of rotate()


private Point rotatePt(Point pt, Point pivot, double angle)
//  rotate the pt point around a given pivot point
{
  AffineTransform at =
```



```
          AffineTransform.getRotateInstance(angle, pivot.x, pivot.y);
  Point2D pt2 = at.transform(pt, null);
  return new Point((int)pt2.getX(), (int)pt2.getY());
         // truncate to an integer point
}  // end of rotatePt()
```

rotate() contains another optimization – the reuse of the original loaded image if the overall rotation of the image returns to 0 (perhaps because of a full-circle rotation). Multiple rotations and scaling tend to reduce an image's resolution over time. By reverting to the originally loaded image the original image quality is restored.

Efficient Scaling

Scaling an image poses some problems, most notably Java crashes due to a lack of heap space. The problem is caused by large amounts of memory needed to store large BufferedImages. My solution is two fold – extra heap space is allocated to java.exe when the application is started, and an upper limit is placed on the size of an image related to the screen's dimensions.

Another simplification is that the image can't be scaled smaller than its original size. This is easy to detect since ImageHolder stores a copy of the original image, and so has access to its dimensions.

```
// globals
private int pWidth, pHeight;          // panel size

private BufferedImage origIm, im;     // original and current images
private Point centerLoc;   // center point of current image
private int currAngle;    // in degrees
private double currScale;


public void scale(double scale)
// scale image by specified scale factor (within certain limits)
{
  // scaling does not change the center point of the image

  if (scale == 1.0)  // no change in scale
    return;
  else if (scale > 1.0) {
    if (im.getHeight()*scale >= pHeight)  // new height too big
      scale = ((double)pHeight-1)/im.getHeight();
    else if (im.getWidth()*scale >= pWidth)  // new width too big
      scale = ((double)pWidth-1)/im.getWidth();
    im = ImageUtils.scale(im, scale);
    currScale *= scale;
  }
  else {     // scale < 1.0
    if (im.getWidth()*scale <= origIm.getWidth()) {  // too small
      im = ImageUtils.rotate(origIm, Math.toRadians(currAngle));
                                 // use original image, rotated
      currScale = 1;   // back to original size
    }
    else {
      im = ImageUtils.scale(im, scale);
      currScale *= scale;
    }
```



```
  }
}  // end of scale()
```

When the attempted scaling would return the image to its original size or smaller, the current image is discarded, and replaced by a freshly rotated copy of the original image, thereby restoring the image quality.

