Extracted from:

Rapid Android Development

Build Rich, Sensor-Based Applications with Processing

The Pragmatic Bookshelf

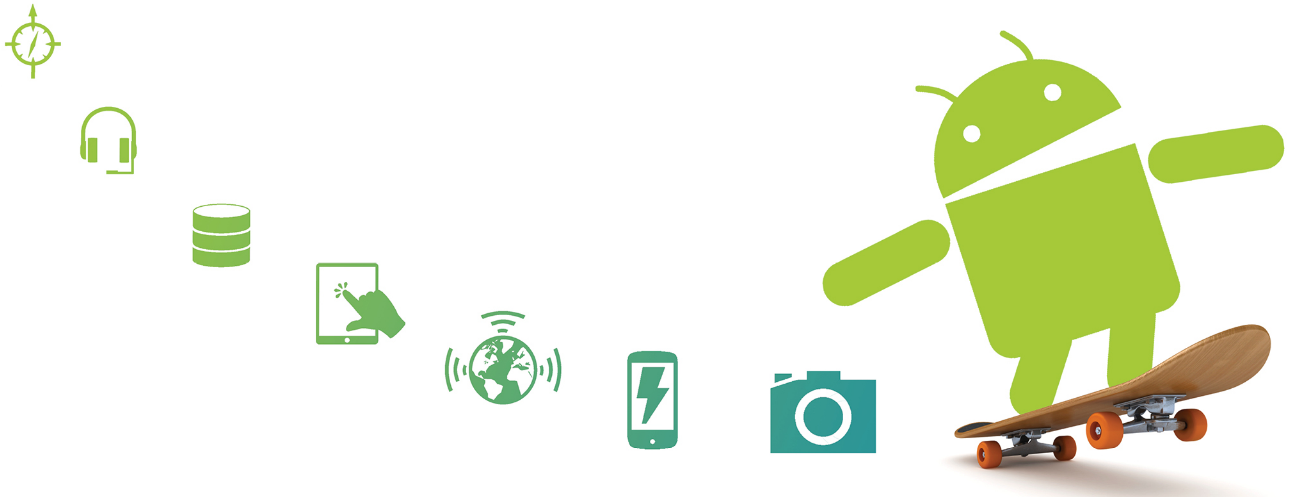
Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Updated for Processing 3 & Android 5 Lollipop

Rapid Android Development

Build Rich, Sensor-Based Applications with Processing



Daniel Sauter

Foreword by Jer Thorp, Co-founder, The Office for Creative Research

Edited by John Osborn

Rapid Android Development

Build Rich, Sensor-Based Applications with Processing

Daniel Sauter

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The team that produced this book includes:

John Osborn (editor)
Potomac Indexing, LLC (indexer)
Molly McBeath (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-93778-506-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P3.0—May 2015

Detect Multitouch Gestures

For this project, we'll implement the most common user interactions using just one simple geometric primitive—a rectangle—drawn on the screen using Processing's rect(x, y, width, height) method.[25] To begin, we'll place a rectangle in a specified size of 100 pixels in the center of the screen. Then we use a series of KetaiGesture callback events to trigger changes to the rectangle, including a change of scale, rotation, color, and position, as illustrated in Figure 5, *Using multitouch gestures*, on page 6.

We have a number of callback events for the touch surface to try out, so we'll assign each of them with a particular purpose. We'll zoom to fit the rectangle onto the screen using onDoubleTap(), randomly change its fill color onLongPress() using Processing's random() method,[26] scale it onPinch(), rotate it onRotate(), drag it using mouseDragged(), and change the background color onFlick(). Besides manipulating color properties and the rectangle, we'll keep track of the multitouch events as they occur by printing a text string to the Processing Console. The code we use to manipulate the properties and the callback methods themselves are not complicated in any way, but we're now dealing with a bit more code than we have before because we're using a series of callback methods in one sketch.

Introducing 2D Transformations

For this project, we'll lock our app into LANDSCAPE orientation() so we can maintain a clear reference point as we discuss 2D transformations in reference to the coordinate system. To center our rectangle on the screen when we start up, to scale from its center point using the pinch gesture, and to rotate it around its center point using the rotate gesture, we need to work with two-dimensional (2D) transformations.[27]

We'll use the Processing's rectMode(CENTER) method to overwrite the default way a rectangle is drawn in Processing,[28] which is from the upper left corner of the rectangle located at position [x, y] with a specified width and height. Instead we draw it from its center point using rectMode(CENTER), which allows us to rotate and scale it around its center point.

25. http://processing.org/reference/rect_.html
26. http://processing.org/reference/random_.html
27. http://processing.org/learning/transform2d/
28. http://processing.org/reference/rectMode_.html

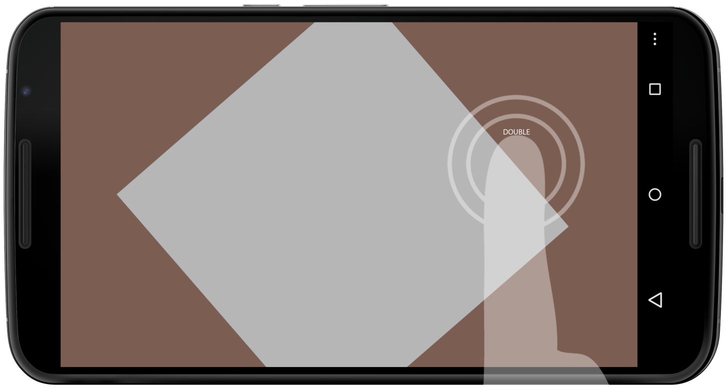


Figure 5—Using multitouch gestures. A rectangle scaled with a two-finger pinch gesture, turned by a two-finger rotation gesture, placed on a brown background color, and triggered by a flick, as well as a gray fill color caused by a long press. The text "DOUBLE" appears due to a double-tap gesture at the position indicated by the hand silhouette.

It is common to explain 2D transformations in terms of a grid, where each grid cell stands for one pixel of our app's display window. The default origin in Processing's coordinate system is always the upper left corner of the window. Each graphic element is drawn relative to this origin. We'll use Processing's transformation methods, translate() and rotate(), to move and rotate our rectangle.[29] We also have a scale() method,[30] which we won't use in this sketch.

When we draw graphic objects in Processing on our grid paper, we are used to specifying the rectangle's horizontal and vertical coordinates using *x* and *y* values. We can use an alternative method, which is necessary here, where we move our grid (paper) to specified horizontal and vertical coordinates, rotate, and then draw the rotated rectangle at position *x* and *y* [0, 0]. This way the rectangle doesn't move to our intended position, but our grid paper (coordinate system) did. The advantage is that we can now rotate() our rect() right on the spot around its center point, something we can't do otherwise.

What's more, we can introduce a whole stack of grid paper if we'd like to by using the Processing methods pushMatrix() and popMatrix(). When we move, rotate, and scale multiple elements and would like to transform them separately, we need to draw them on separate pieces of grid paper. The pushMatrix() method saves the current position of our coordinate system, and popMatrix() restores the coordinate system to the way it was before pushing it.

29. http://processing.org/reference/translate_.html and http://processing.org/reference/rotate_.html.
30. http://processing.org/reference/scale_.html

Like our first project in this chapter, in which we used Processing's mousePressed(), mouseReleased(), and mouseDragged() callback methods to identify touches to the screen, some of the multitouch gestures introduced here fulfill the same purpose. If we'd like to use Processing's mouse methods alongside multitouch methods provided by KetaiGesture, we'll need to notify the superclass method surfaceTouchEvent() to notify the Processing app that a surface touch event has occurred.[31]

Now let's take a look at our multitouch code.

Display/Gestures/Gestures.pde

```
 1  import ketai.ui.*;
 2  import android.view.MotionEvent;

 3  KetaiGesture gesture;
 4  float rectSize = 100;
    float rectAngle = 0;
    int x, y;
 5  color c = color(255);
 6  color bg = color(78, 93, 75);

    void setup()
    {
      orientation(LANDSCAPE);
 7    gesture = new KetaiGesture(this);

      textSize(32);
      textAlign(CENTER, BOTTOM);
      rectMode(CENTER);
      noStroke();

 8    x = width/2;
 9    y = height/2;
    }

    void draw()
    {
      background(bg);
10    pushMatrix();
11    translate(x, y);
      rotate(rectAngle);
      fill(c);
      rect(0, 0, rectSize, rectSize);
12    popMatrix();
    }

13  public boolean surfaceTouchEvent(MotionEvent event) {
```

31. http://processing.org/reference/super.html

```
     //call to keep mouseX and mouseY constants updated
     super.surfaceTouchEvent(event);
     //forward events
     return gesture.surfaceTouchEvent(event);
   }

14 void onTap(float x, float y)
   {
     text("SINGLE", x, y-10);
     println("SINGLE:" + x + "," + y);
   }

15 void onDoubleTap(float x, float y)
   {
     text("DOUBLE", x, y-10);
     println("DOUBLE:" + x + "," + y);

     if (rectSize > 100)
       rectSize = 100;
     else
       rectSize = height - 100;
   }

16 void onLongPress(float x, float y)
   {
     text("LONG", x, y-10);
     println("LONG:" + x + "," + y);

     c = color(random(255), random(255), random(255));
   }

17 void onFlick( float x, float y, float px, float py, float v)
   {
     text("FLICK", x, y-10);
     println("FLICK:" + x + "," + y + "," + v);

     bg = color(random(255), random(255), random(255));
   }
18 void onPinch(float x, float y, float d)
   {
     rectSize = constrain(rectSize+d, 10, height);
     println("PINCH:" + x + "," + y + "," + d);
   }

19 void onRotate(float x, float y, float angle)
   {
     rectAngle += angle;
     println("ROTATE:" + angle);
   }
```

```
⑳ void mouseDragged()
{
  if (abs(mouseX - x) < rectSize/2 && abs(mouseY - y) < rectSize/2)
  {
    if (abs(mouseX - pmouseX) < rectSize/2)
      x += mouseX - pmouseX;
    if (abs(mouseY - pmouseY) < rectSize/2)
      y += mouseY - pmouseY;
  }
}
```

Let's take a look at the steps we need to take to capture and use multitouch gestures on the Android touch screen.

1. Import Ketai's ui package to give us access to the KetaiGesture class.
2. Import Android's MotionEvent package.
3. Define a variable called gesture of type KetaiGesture.
4. Set a variable we call rectSize to 100 pixels to start off.
5. Define the initial color c (white), which we'll use as a fill color for the rectangle and text.
6. Define the initial color bg (dark green), which we'll use as a background color.
7. Instantiate our KetaiGesture object gesture.
8. Set the initial value for our variable x as the horizontal position of the rectangle.
9. Set the initial value for y as the vertical position of the rectangle.
10. Push the current matrix on the matrix stack so that we can draw and rotate the rectangle independent of other UI elements, such as the text.
11. Move to the position [x, y] using translate().
12. Pop the current matrix to restore the previous matrix on the stack.
13. Use the Processing method surfaceTouchEvent() to notify Processing about mouse/finger-related updates.
14. Use the callback method onTap() to display the text string SINGLE at the location (x, y) returned by KetaiGesture.
15. Use the callback method onDoubleTap() to display the text string DOUBLE at the location returned by KetaiGesture, indicating that the user triggered a double-tap event. Use this event to decrease the rectangle size to the

original 100 pixels if it's currently enlarged, and increase the rectangle scale to the display height minus 100 pixels if it's currently minimized to its original scale.

⑯ Use the callback method onLongPress() to display the text string "LONG" at the location (x, y) returned by KetaiGesture. Use this event to randomly select a new color c using random(), which we'll use as a fill color for the rectangle.

⑰ Use the callback method onFlick() to display the text string FLICK at the location x and y returned by KetaiGesture. Also, receive the previous location where the flick has been initiated as px and py, as well as the velocity v.

⑱ Use the callback method onPinch() to calculate the scaled rectSize using the pinch distance d at the location x and y returned by KetaiGesture.

⑲ Use the callback method onRotate() to calculate the rotation angle rectAngle using the angle returned by KetaiGesture.

⑳ Use Processing's mouseDragged() callback to update the rectangle position (x and y) by the amount of pixels moved. Determine this amount by subtracting the previous pmouseX from the current mouseX, and pmouseY from mouseY. Move the rectangle only if absolute distance between the rectangle and the mouse position is less than half the rectangle's size, or when we touch the rectangle.

Let's test the app.

Run the App

Run the app on your device. You'll see a square show up in the center of the screen. Drag it to a new location, flick to change the background color, and give it a long tap to change the foreground fill color.

To test the multitouch gestures, put two fingers down on the screen and pinch, and you'll see how the rectangle starts scaling. Now rotate the same two fingers to see the rectangle rotate. If you use more than two fingers, the first two fingers you put down on the screen are in charge.

Finally, double-tap the screen to zoom the square to full screen, and double-tap again to scale it to its initial size of 100 pixels.

This completes our investigation into the multitouch features of the touch screen panel.