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

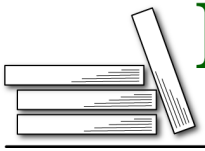
Rapid Android Development

Build Rich, Sensor-Based Applications with Processing

Daniel Sauter

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *https://pragprog.com*.

The team that produced this book includes:

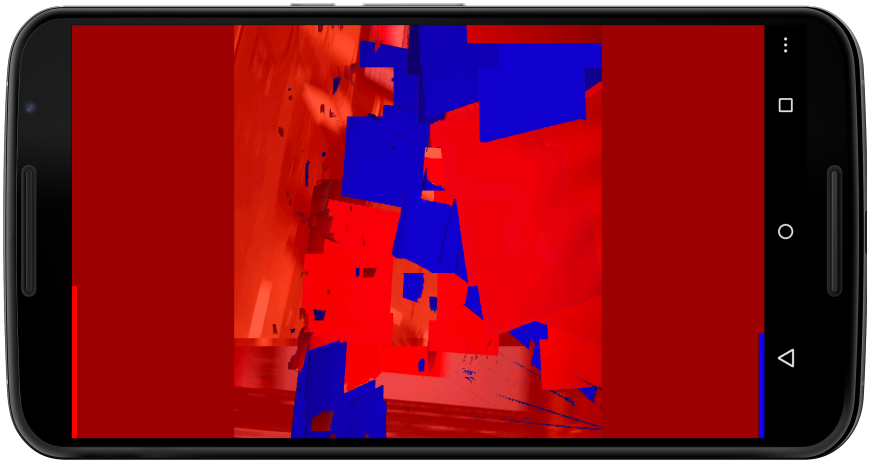
John Osborn (editor)
Potomac Indexing, LLC (indexer)
Molly McBeath (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

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 and Trace the Motion of Colored Objects

In the drawing game that we'll build in this section, two players will compete to see who can fill the screen of an Android device with the color of a red or blue object first. Without touching the device screen, each player scribbles in the air above it with a blue or red object in an attempt to fill as much space as possible with the object's color. When more than 50 percent of the screen is filled, the player that filled in the most pixels wins. We'll use the front-facing camera as the interactive interface for this game. It's job is to detect the presence of the colors blue or red within its field of vision and capture them each time it records a frame. The game code will increase the score of each player who succeeds in leaving a mark on the screen. It will look like this.



The camera remains static during the game, and only the primary colors red and blue leave traces and count toward the score. If the red player succeeds in covering more pixel real estate than the blue, red wins. If blue dominates the screen, blue wins. If you are using an Android tablet you can step a little bit further away from the device than is the case for a phone, where the players are more likely to get in each other's way, making the game more competitive and intimate.

The magic marker drawing game uses color tracking as its main feature. As we implement this game, we put Processing's image class, called *PImage*, to use. The main purpose of this datatype is to store images, but it also contains a number of very useful methods that help us manipulate digital images. In the context of this game, we'll use `PImage` methods again to retrieve pixel color

values and to set pixel values based on some conditions we implement in our sketch.

Manipulating Pixel Color Values

To create this magic marker drawing game, we need to extract individual pixel colors and decide whether a pixel matches the particular colors (blue and red) we are looking for. A color value is only considered blue if it is within a range of "blueish" colors we consider blue enough to pass the test, and the same is true for red. Once we detect a dominant color between the two, we need to call a winner.

For an RGB color to be considered blue, the blue() value of the pixel color needs to be relatively high,[24] while at the same time the red() and green() values must be relatively low.[25] Only then does the color appear blue. We are using the Processing color methods red(), green(), and blue() to extract *R*, *G*, and *B* values from each camera pixel. Then we determine whether we have a blue pixel, for instance, using a condition that checks if blue() is high (let's say 200) and at the same time red() and green() are low (let's say 30) on a scale of 0..255. To make these relative thresholds adjustable, let's introduce variables called high and low for this purpose.

Let's take a look. The sketch again contains CameraControls, which we don't discuss here because we already know the method to start() and stop() the camera.

Camera/CameraMagicMarker/CameraMagicMarker.pde

```
import ketai.camera.*;

KetaiCamera cam;
PImage container;
int low = 30;
int high = 100;
int camWidth = 1280;
int camHeight = 768;
int redScore, blueScore = 0;
int win = 0;

void setup() {
  orientation(LANDSCAPE);
  imageMode(CENTER);
  cam = new KetaiCamera(this, camWidth, camHeight, 30);
  // 0: back camera; 1: front camera
  cam.setCameraID(1);
```

24. http://processing.org/reference/blue_.html
25. http://processing.org/reference/red_.html and http://processing.org/reference/green_.html.

```
  container = createImage(camWidth, camHeight, RGB);
}

void draw() {
  if (win == 0) background(0);
  if (cam.isStarted()) {
    cam.loadPixels();
    float propWidth = height/camHeight*camWidth;
    if (win == 0) image(cam, width/2, height/2, propWidth, height);
    for (int y = 0; y < cam.height; y++) {
      for (int x = 0; x < cam.width; x++) {
        color pixelColor = cam.get(x, y);
        if (red(pixelColor) > high &&
          green(pixelColor) < low && blue(pixelColor) < low) {
          if (brightness(container.get(x, y)) == 0) {
            container.set(x, y, pixelColor);
            redScore++;
          }
        }
        if (blue(pixelColor) > high &&
          red(pixelColor) < low && green(pixelColor) < low) {
          if (brightness(container.get(x, y)) == 0) {
            container.set(x, y, pixelColor);
            blueScore++;
          }
        }
      }
    }
    image(container, width/2, height/2, propWidth, height);
    fill(255, 0, 0);
    rect(0, height, 20, map(redScore, 0, camWidth*camHeight, 0, -height));
    fill(0, 0, 255);
    rect(width-20, height, 20, map(blueScore, 0, camWidth*camHeight, 0, -height));
    if (redScore+blueScore >= camWidth*camHeight * 0.50) {
      win++;
      if (redScore > blueScore) {
        fill(255, 0, 0, win);
      }
      else {
        fill(0, 0, 255, win);
      }
      rect(0, 0, width, height);
    }
    if (win >= 50) {
      container.loadPixels();
      for (int i = 0; i < container.pixels.length; i++) {
        container.pixels[i] = color(0, 0, 0, 0);
        redScore = blueScore = win = 0;
      }
    }
```

```
  }
}

void mousePressed()
{
   if(cam.isStarted())
      cam.stop();
   else
     cam.start();
}
```

There are a couple of new methods for us to look at.

❶ Create an empty PImage called container using the createImage() method to hold red and blue color pixels that have been detected in the camera preview image. The empty RGB image container matches the size of the camera preview image.

❷ Calculate the fullscreen camera preview image width propWidth proportional to the camera preview aspect ratio. We get the ratio by dividing the screen height by the camera preview height camHeight and multiplying that with the camWidth.

❸ Draw the camera preview image in fullscreen size using image() if no player has won the game yet (win equals 0). Match the image height with the screen height and scale the image width proportionately.

❹ Get the color value at the image pixel location x and y using the PImage method get(). Store the value in the color variable pixelColor.

❺ Check for reddish pixel values within the camera preview using the red(), green(), and blue() PImage methods to extract individual color values from the color datatype. Consider only pixel values with a red content greater than the high threshold and low green and blue values. Use the globals high and low for the upper and lower limits of this condition.

❻ Check if the pixel is already taken by a color using brightness(). If the container is empty and not set yet, it has a brightness value of 0.

❼ Check for blueish pixel value in the camera image. It requires a color with a high blue content, while the red and green values are low.

❽ Draw the container using the image() method. This PImage contains all the red and blue pixels we grabbed from the camera's preview image.

❾ Check for the winner when at least 50 percent of the image is covered, comparing the combined redScore and blueScore values against 0.50 of all camera preview pixels.

⑩ Fade to the winning color by changing the fill() opacity of a colored rectangle covering the screen. To achieve a continuous fade, use the win variable for the alpha parameter so that the following rectangle is drawn with decreasing opacity (0: fully opaque, 255 fully transparent).

⑪ Load the pixel data from the container PImage into the pixels[] array. The function must be called before writing to (or reading from) pixels[].

⑫ Empty all pixels[] in the container image pixel array. Set all pixels to the color(0, 0, 0, 0), which is a fully transparent black color. The Processing rule is that you must call loadPixels() before you read from or write to pixels[], even if some renderers seem not to require this call.

Now let's see how well the camera picks up the colorstest of some blueish and reddish objects. Any kind of object will do as long as its color is a vibrant red or blue—the more intense its hue and brightness the better.

Run the App

Grab a friend and a few blueish and reddish objects, and get ready to scribble madly mid-air and fight for pixel real estate on the Android device. When you run the sketch, the camera preview will appear centered on the screen, stretched to fullscreen size. Reddish and blueish colors are instantly picked up and drawn on top of the preview image. This immediate feedback lets us play with different objects and quickly get an idea about which objects have the greatest color impact as we try to cover the screen.

Try it. The status bar on either side of the screen grows as colors are picked up, showing us how much pixel real estate each player owns. Individual scores are compared with the total number of available pixels. If 50 percent of all pixels are grabbed by the red player, for instance, the red progress bar covers half of the screen height. Once more than 50 percent of all available pixels are taken, the sketch calls a winner and fades to the winning color. It resets the game to start over.

This game has taken us deep into the world of pixels using all the prior color knowledge we've acquired in *Build a Motion-Based Color Mixer and Palette*, on page ?. The PImage datatype is a convenient way to work with images, which are in principle "just" lists of colors containing red, green, blue, and alpha (transparency) values that we can use for our own purposes, such as our magic marker drawing game.

If your device is up to the challenge, feel free to double the camera resolution via camWidth and camHeight for better image quality, but consequently you'll

have to lower the frame rate. We've discussed that pixel-level calculations are computationally expensive and hence require a speedy Android device to run smoothly. In Chapter 11, *Introducing 3D Graphics with OpenGL*, on page ?, we will learn a few tricks that help us put the graphics processing unit (GPU) to use, keeping the central processing unit (CPU) free for other tasks.

Since you've successfully interpreted images on a pixel level, let's take it a step further now and explore how pixel-level image algorithms are used for advanced image processing and computer vision purposes, specifically for Android's face detection API.