NUI Chapter 5. Blobs Drumming

[**Note**: all the code for this chapter is available online at http://fivedots.coe.psu.ac.th/~ad/jg/??; only important fragments are described here.]

A particularly simple technique for detecting shapes (or blobs) in an image is to look for regions consisting of the same color. My BlobsDrumming application, shown in Figure 1, looks for light blue and red blobs, and highlights them as rectangles.
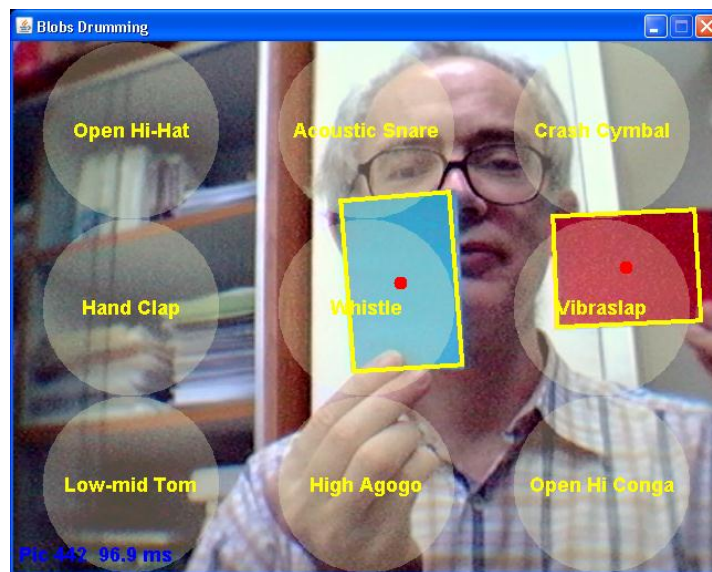

Figure 1. Blob Drumming.

The screen area is divided into nine 'drumming' areas (shown as translucent labeled circles in Figure 1). If the center of a blob rectangle falls within one of these areas, then a sequence of percussion beats is generated. In Figure 1, my PC's speakers are emitting a mix of whistles and vibraslap beats.

A rectangle's orientation affects tempo, with a faster beat assigned to a rectangle whose long edge is rotated away from the vertical. Since the red card shown in Figure 1 is almost horizontal, the vibraslaps occur with great rapidity.

The noise produced can hardly be called music, although moving the cards around the screen produces some amusing percussion effects (amusing for me, but perhaps not so much for my cubicle neighbors).

I'll start this chapter with a brief introduction to MIDI sound synthesis, which is how the percussion sounds are generated. Then I'll describe blob detection using contour finding in JavaCV.

Detection success greatly depends on the quality of the camera and the lighting conditions. For instance, the difference between indoor and natural lighting are enough to confuse a detector. For that reason, blob detection must be preceded by a

"color calculation" phase where color attributes for the cards are obtained under the same camera and environmental conditions that are used later by the BlobsDrumming application. I obtain these color calculations using a separate application, called HSVSelector. It represents colors in the HSV format (hue/color, saturation, and value/brightness), which makes it easier for the detector to ignore variations in brightness.

1. MIDI

The Java Sound API supports the capture, playback, and synthesis of sampled audio and MIDI (Musical Instrument Digital Interface) sequences.

A key benefit of MIDI is that it represents musical data in an extremely efficient way, leading to drastic reductions in file sizes compared to sampled audio. For instance, files containing high quality stereo sampled audio require about 10 Mb per minute of sound, while a typical MIDI sequence may need less than 10 Kb.

The secret to this phenomenal size reduction is that a MIDI sequence stores 'instructions' for playing the music rather than the music itself. A simple analogy is that a sequence is the written score for a piece of music rather than a recording of it.

The drawback is that the sequence must be converted to audio output at run-time. This is achieved using a sequencer and synthesizer. Their configuration is shown in greatly simplified form in Figure 2.
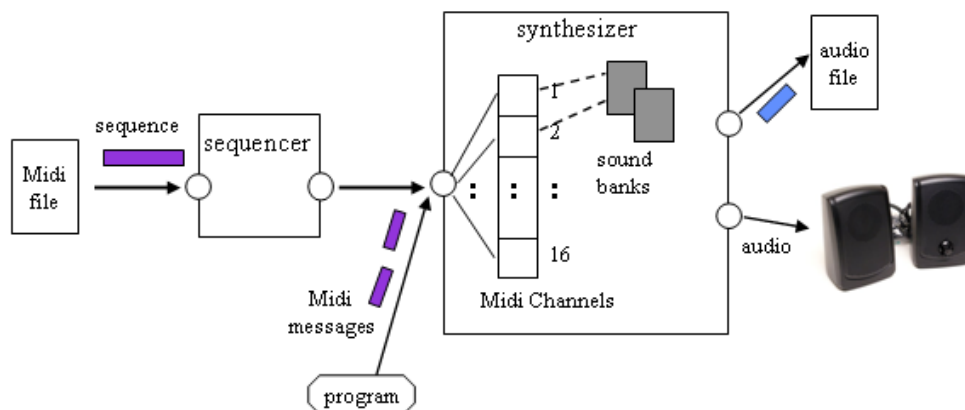

Figure 2. A MIDI sequencer and synthesizer

A MIDI sequencer allows MIDI data sequences to be captured, stored, edited, combined, and performed, while the MIDI data's transformation into audio is carried out by the synthesizer.

Continuing my analogy, the sequencer is the orchestral conductor, who receives the score to play, perhaps making changes to it in the process. The synthesizer is the orchestra, made up of musicians playing different parts of the score, corresponding to the MidiChannel objects in the synthesizer. They're allocated instruments from the sound banks, and play concurrently. In this chapter, the only musician I'll be using is the MidiChannel in charge of percussion.

Usually a complete sequence (a complete score) is passed through the sequencer to the synthesizer, but it's also possible to send a stream of MIDI messages directly to the synthesizer. I'll be using that approach, so I won't need a sequencer in my code.

MIDI Synthesis

The MidiChannel class in the javax.sound.midi package offers noteOn() and noteOff() methods which produce NOTE_ON and NOTE_OFF MIDI messages:

```
void noteOn(int noteNumber, int velocity);

void noteOff(int noteNumber);
```

The note number is the MIDI number assigned to a musical note, while velocity is roughly equivalent to the note's loudness.

A note will keep playing after a noteOn() call, until it's terminated with noteOff(). This means that it's possible to have several notes playing at the same time, by calling noteOn() several times before calling noteOff().

A table linking MIDI note numbers to musical note names can be found at http://www.phys.unsw.edu.au/~jw/notes.html. Table 1 shows the mapping for part of the 4th octave.

MIDI Number	Note Name
60	C4
61	C#4
62	D4
63	D#4
64	E4
65	F4

Table 1. MIDI Numbers and Note Names.

A MIDI channel is obtained in the following way:

```
Synthesizer synthesizer = MidiSystem.getSynthesizer();
synthesizer.open();
MidiChannel drumChannel = synthesizer.getChannels()[9]; // channel 9
```

Channel 9 is non-standard because it maps note numbers to percussion sounds, as illustrated by Table 2.

MIDI Number	Percussion Name
60	Hi Bongo
61	Low Bongo
62	Mute Hi Conga
63	Open Hi Conga
64	Low Conga
65	High Timbale

Table 2. Some MIDI Numbers and Percussion Names.

A full list of the mappings of MIDI numbers to percussion sounds can be found at http://www.midi.org/about-midi/gm/gm1sound.shtml (in the "General MIDI Level 1 Percussion Key Map" table).

Playing a note (or percussion sound) corresponds to sending a NOTE_ON message, letting it play for a while, and then killing it off with a NOTE_OFF message. This can be wrapped up in a playNote() method:

```java
public void playNote(int note, int duration)
{
  drumChannel.noteOn(note, 100);  // 100 is the volume; max is 127
  try {
    Thread.sleep(duration);       // ms sleep time
  }
  catch (InterruptedException e) {}
  drumChannel.noteOff(note);
}
```

The following will trigger applause:

```java
for(int i=0; i < 10; i++)
  playNote(39, 1000);  // 1 sec duration
```

Note 39 corresponds to the percussion version of a "Hand Clap".

Percussion notes behave a little differently from notes generated by other channels: most percussion notes have a fixed duration, after which the sound goes silent (or very quiet).

2. The Percussion Player

My PercussionPlayer class links to the synthesizer's MIDI channel for playing percussion beats. The instruments are identified by name which means that a user doesn't need to know about MIDI messages, and needn't store a MIDI note-to-name mapping table in their heads (similar to Table 2).

The class is designed to act as a percussion player for multiple instruments at once (think of what a real-world drummer does). As a consequence, I don't wrap the

playing of a beat and its switching off in a playNote() method like the one above. Instead, it's necessary to use separate methods so that several instruments can be switched on/off at the same time.

A short code fragment showing PercussionPlayer playing two instruments at once:

```
// whistle and crash cymbals together
PercussionPlayer player = new PercussionPlayer();
player.drumOn("Whistle");
player.drumOn("Crash Cymbal");

try {
  Thread.sleep(200);    // wait a short time
}
catch (InterruptedException e) {}

player.drumOff("Whistle");
player.drumOff("Crash Cymbal");
player.close();
```

The BlobsDrumming application treats its two cards as drumsticks, so it's natural for its audio output to contain two instruments beating at the same time. This is implemented by two drum threads communicating with the PercussionPlayer object with drumOn() and drumOff() calls.

I'll explain the details when I describe BlobsDrumming at the end of this chapter.

2.1. Starting and Closing the Player

The PercussionPlayer constructor connects to the hardware synthesizer, then contacts its percussion MIDI channel. The close() method closes down these links.

```
// globals
private static final int PERCUSSION_CHANNEL = 9;

private Synthesizer synthesizer = null;
private MidiChannel channel = null;     // percussion channel


public PercussionPlayer()
{
  try {
    synthesizer = MidiSystem.getSynthesizer();
    synthesizer.open();
    channel = synthesizer.getChannels()[PERCUSSION_CHANNEL];

    // wake up the synthesizer channel
    drumOn("Crash Cymbal");
    wait(100);
    drumOff("Crash Cymbal");
  }
  catch(MidiUnavailableException e) {
    System.out.println("Cannot initialize MIDI synthesizer");
    System.exit(1);
  }
}  // end of PercussionPlayer()
```

© Andrew Davison 2013

```
synchronized public void close()
{
  if (channel != null) {
     channel.allNotesOff();
     channel = null;
  }
  if (synthesizer != null) {
    synthesizer.close();
    synthesizer = null;
  }
}  // end of close()
```

The PercussionPlayer will be used by multiple threads in BlobsDrumming, so methods, such as PercussionPlayer.close(), which might be called concurrently, are synchronized to prevent contention.

I added the calls to drumOn() and drumOff() in the PercussionPlayer() when I updated my JDK to version 7, which uses the Gervill software synthesizer. They cause the synthesizer to 'wake up' so that subsequent calls to play notes are processed immediately. Without this wake-up, the synthesizer sometimes adds a 1-2 second delay to the first note playing requests made by the user.

2.2. Turning the Percussion On and Off

PercussionPlayer's drumOn() and drumOff() methods are sugared calls to MidiChannel.noteOn() and MidiChannel.noteOff(), adding error checking and the conversion of instrument names to MIDI note numbers.

```
// globals
private static final int VELOCITY = 127;       // max volume

private static String[] instrumentNames = {
  "Open Hi-Hat", "Acoustic Snare", "Crash Cymbal",
  "Hand Clap", "Whistle", "Vibraslap",
  "Low-mid Tom", "High Agogo", "Open Hi Conga"
};

private int[] instrumentKeys = {
  46, 38, 49, 39, 72, 58, 47, 67, 63
};   // must correspond to names in instrumentNames[]

private MidiChannel channel = null;


synchronized public void drumOn(String name)
{
  int key = name2Key(name);
  if ((channel != null) && (key != -1))
    channel.noteOn(key, VELOCITY);
}  // end of drumOn()


synchronized public void drumOff(String name)
{
  int key = name2Key(name);
  if ((channel != null) && (key != -1))
```

```
      channel.noteOff(key);
}  // end of drumOff()



private int name2Key(String name)
// convert an instrument name to its MIDI percussion key
{
  for (int i=0; i < instrumentNames.length; i++)
    if (instrumentNames[i].equals(name))
      return instrumentKeys[i];
  return -1;
}  // end of name2Key()
```

The drumOn() and drumOff() methods are synchronized so that two threads can't call the same method at once.

2.3. A Long NoteOn Duration ≠ Multiple Beats

If a note is turned on via most MIDI channels, then it will keep playing until it's switched off. This is not the case for percussion channel instruments. Switching an instrument 'on' means that it will beat once then go silent. For example:

```
player.drumOn("Whistle");
player.drumOn("Crash Cymbal");
wait(5000);     // wait for 5 seconds
      // beats do NOT play for 5 seconds

player.drumOff("Whistle");
player.drumOff("Crash Cymbal");
```

Only a short whistle and a single cymbal crash are heard even though both notes are left on for 5 seconds. wait() calls Thread.sleep():

```
private static void wait(int delay)
{
  try {
    Thread.sleep(delay);
  }
  catch (InterruptedException e) {}
}  // end of wait()
```

Actually the situation is somewhat more complicated, since some instruments *do* prolong their playing while the note is on. For instance, although the whistle stops after a fraction of a second, a faint echo can be heard in the background until the note is turned off.

After testing, I decided to specify a single beat in later code as a drumOn() call followed by a drumOff() call after a fixed wait of 200 ms. This definition is used to define a single drum beat in the Drum class in the BlobsDrumming application described later.

2.4. JMF and Java Sound

I discarded the Java Media Framework (JMF) back in Chapter 2 due to its platform problems. Another reason for avoiding it arises here – it causes the MIDI synthesizer in Java Sound to crash (at least in JDK 7). The raised exception is an IllegalAccessError to the AbstractMidiDevice class when MidiSystem.getSequencer() is called.

The solution is to uninstall all traces of JMF. I used the freeware version of Revo Uninstaller (http://www.revouninstaller.com/revo_uninstaller_free_download.html) which also cleans out the registry on my Windows machine. If the error persists, then the only certain solution is to uninstall Java completely, and reinstall it from scratch.

3. Selecting a Color for a Blob

Before the BlobsDrumming application can detect blobs, I need to collect information about their colors. This isn't as easy as it first appears, since their attributes depend greatly on the webcam, the local lighting factors, and even the color of my shirt!

I'm using two colored cards as my blob rectangles, shown in Figure 3.
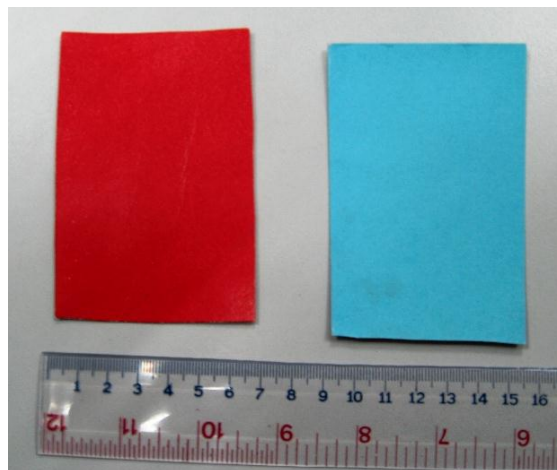

Figure 3. My Blue and Red Cards.

The picture in Figure 3 was taken with a good digital camera, but both cards look somewhat different when seen through my test machine's el-cheapo webcam. Figure 4 shows me holding the red card, which looks lighter, and has a lot of image noise mixed in with the main color. The camera also adds a reddish tint to the entire scene.
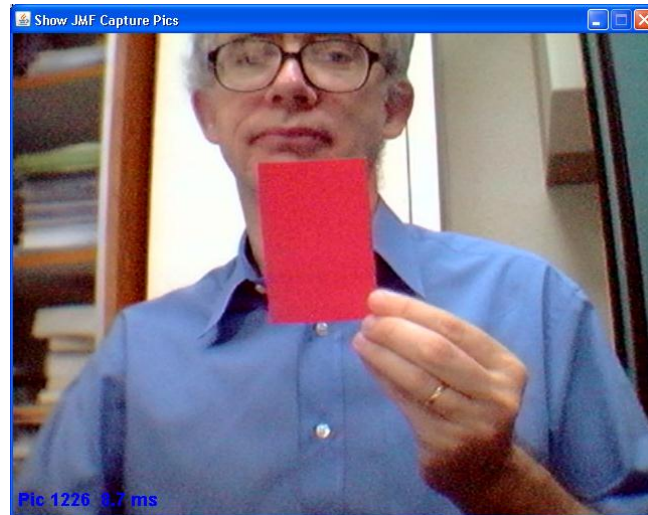
Figure 4. The Red Card Viewed Via Webcam.

Figure 4 illustrates some problems with my test environment. One issue is the harsh lighting, which causes the red card to shine excessively (i.e. appear white) when held at certain angles.

Another problem is that I'm wearing a blue shirt, whose color is too similar to the blue card. The detector easily mistakes parts of my shirt for the card. In other tests, I wore a white shirt.

Most blob detectors don't use the familiar RGB (red, green, blue) format to define colors, but HSV (hue/color, saturation, and value/brightness), which makes it easier for the detector to ignore variations in brightness. I followed the same strategy, and found that my detector didn't need brightness information at all in order to find a blob.

Figure 5 shows my HSVSelector application, which I used to determine suitable HSV ranges for each card. Figure 5 shows me fiddling with the ranges for the red card.
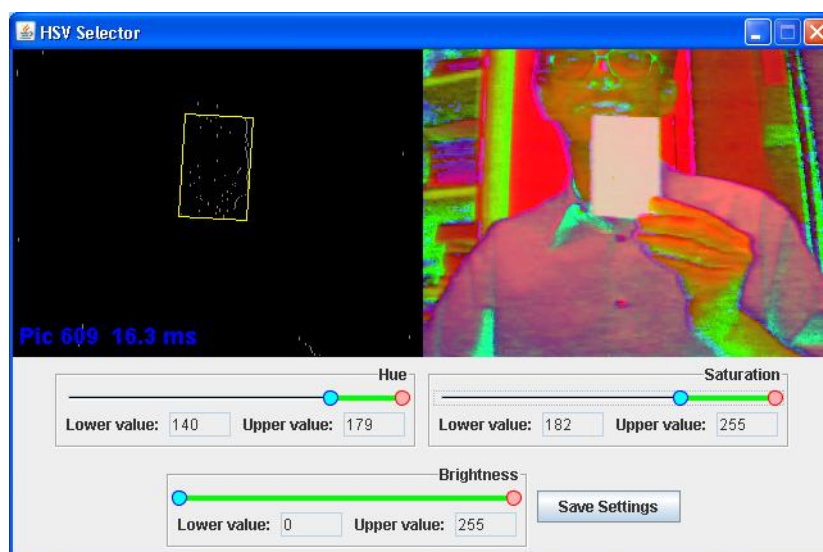

Figure 5. HSVSelector and the Red Card.

The ranges are manipulated by three sliders labeled "Hue", "Saturation" and "Brightness". In Figure 5, the current hue range is 140-179, the saturation is 182-255, and the brightness covers the complete scale 0-255 which means that it will have no effect on which pixels are assigned to a blob.

The right-hand display panel shows the current webcam image, converted from RGB to HSV format. This allows the user to see how a HSV colored card differs from the surrounding environment.

The left-hand panel applies the ranges to the HSV image as upper and lower threshold bounds. The result is that pixels inside all the three ranges appear white, while colors outside are mapped to black. Figure 5 doesn't show it clearly but the threshold image contains several patches of white pixels. These are collected into blobs using JavaCV functions, and the largest blob is highlighted with a bounding box drawn in yellow.

The user's task is to adjust the HSV sliders until the card is consistently highlighted in the threshold image independent of its position or orientation. Those HSV ranges can then be saved to a text file by the user pressing the "Save Settings" button.

HSVSelector was run twice for BlobsDrumming – once to obtain ranges for the red card (as shown above), and again to calculate settings for the blue card.

It can be quite difficult finding good ranges, so I used Shervin Emami's HSV Color Wheel program (http://www.shervinemami.co.cc/colorWheelHSV.7z). The wheel allows me to find HSV values that roughly match a card's color, and I used these as starting values for the HSVSelector sliders.

Figure 6 shows the class diagrams for HSVSelector.
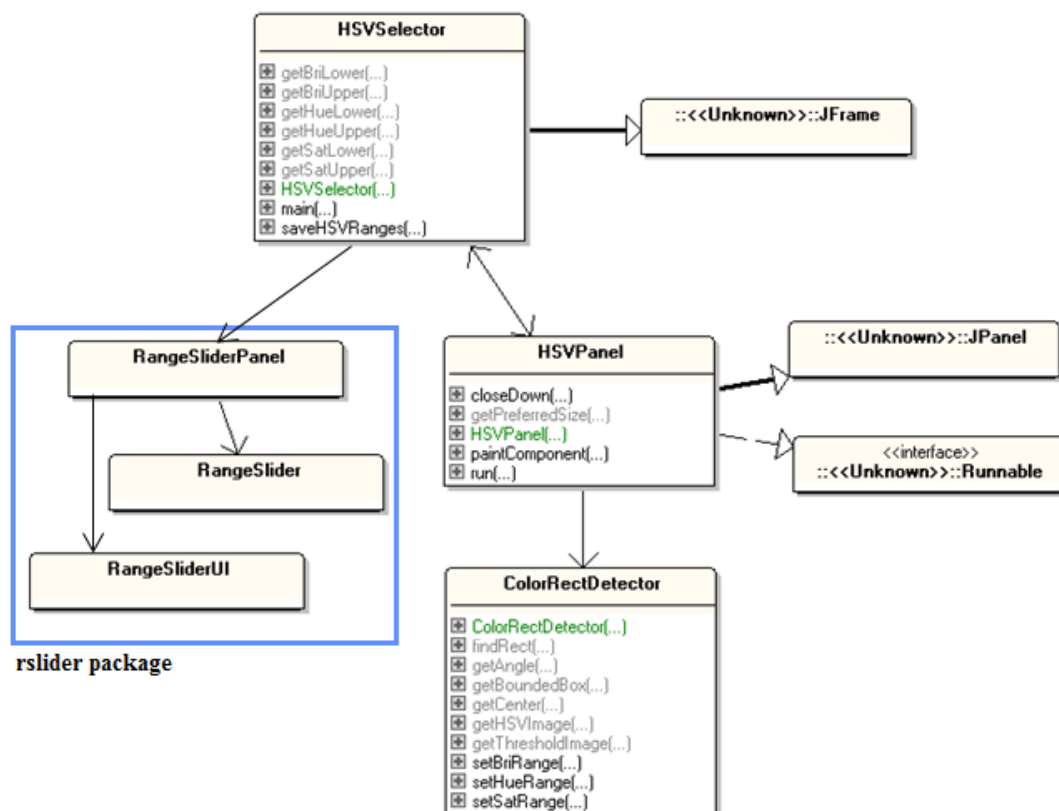

© Andrew Davison 2013

Figure 6. Class Diagrams for HSVSelector.

The HSVSelector class is the top-level JFrame, and uses HSVPanel to display the threshold and HSV images side-by-side in a single panel. The GUI uses three instances of the RangeSliderPanel class (a subclass of JPanel) as the hue, saturation, and brightness sliders. The webcam image is obtained using JavaCV's FrameGrabber inside HSVPanel's thread, and the hard work of blob detection is done by ColorRectDetector.

The Range Slider

RangeSlider was originally developed by Ernie Yu, and is explained in his blog at http://ernienotes.wordpress.com/2010/12/27/creating-a-java-swing-range-slider/. His GUI component addresses Java's JSlider limitation of only having a single thumb for adjusting its value. Yu's slider has two thumbs which permits the user to select a range.

This book isn't about GUI component design, so I'll skip the details of his implementation, which subclasses JSlider. The other important class, RangeSliderUI, extends BasicSliderUI to paint two thumbs on the component and handle user events. My contribution is quite minor: I added a subclass of JPanel (RangeSliderPanel) so I could include two textfields along with Yu's range slider, and surround the lot with a title border.

The following RangeSliderDemo program is a JFrame containing a single range slider panel, as shown in Figure 7.
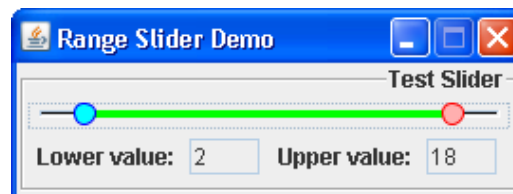

Figure 7. A RangeSlider Example.

RangeSliderDemo shows how Yu's slider can be initialized and used:

```
import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;

import rslider.*;   // the Range Slider package


public class RangeSliderDemo extends JFrame
{
  private RangeSliderPanel rsPanel;
  private RangeSlider slider;


  public RangeSliderDemo()
  {
    super("Range Slider Demo");
```

© Andrew Davison 2013

```
    Container c = getContentPane();
    c.setLayout(new BorderLayout());

    rsPanel = new RangeSliderPanel("Test Slider", 0, 20, 2, 18);
      // title; slider min, max; current thumb settings

    c.add(rsPanel, BorderLayout.CENTER);

    // listen for slider changes, and update the slider panel
    slider = rsPanel.getRangeSlider();
    slider.addChangeListener(new ChangeListener() {
      public void stateChanged(ChangeEvent e)
      {
        int lower = slider.getValue();
        int upper = slider.getUpperValue();
        rsPanel.updateLabels(lower, upper);
        System.out.println("Current lower-upper: " +
                                  lower + " - " + upper);
      }
    });

    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLocationRelativeTo(null);
    pack();
    setResizable(false);
    setVisible(true);
  }  // end of RangeSliderDemo()

  // -------------------------------

  public static void main(String[] args)
  { new RangeSliderDemo(); }

}  // end of RangeSliderDemo class
```

The RangeSliderPanel constructor sets its border title, the minimum and maximum slider values, and the current thumbs settings:

```
RangeSliderPanel rsPanel =
    new RangeSliderPanel("Test Slider", 0, 20, 2, 18);
```

The RangeSlider object inside RangeSliderPanel must have a ChangeListener attached to it, which will fire whenever one of the slider's thumbs is moved by the user. The new thumb values are accessed by RangeSlider.getValue() and RangeSlider.getUpperValue(), and must be explicitly passed to the RangeSliderPanel to update its two textfields.

4. Detecting Blobs

My ColorRectDetector class performs blob detection for HSVSelector (and for BlobsDrumming). First the hue, saturation, and brightness (HSV) ranges are set, then the ColorRectDetector object is passed an image for analysis. The detector records the coordinates of the bounding box around the biggest blob with those color ranges. It

     © Andrew Davison 2013

also stores the rectangle's center, and the angle that its longest edge makes with the horizontal. In HSVSetting, the bounding box's coordinates are used to draw the yellow box on top of the threshold image in Figure 5.

The ColorRectDetector() constructor creates two empty global IplImage objects, one for the HSV version of the webcam image (shown on the right of Figure 5), and one for its binary threshold (on the left of Figure 5). The creation of these objects require image dimensions, which are supplied to the constructor. Default initial HSV ranges are also set.

```
// globals
private static final int NUM_POINTS = 4;  // no of coords in box

// default HSV initial slider ranges
private static final int HUE_LOWER = 0;
private static final int HUE_UPPER = 179;
          // the Hue component ranges from 0 to 179 (not 255)

private static final int SAT_LOWER = 0;
private static final int SAT_UPPER = 255;

private static final int BRI_LOWER = 0;
private static final int BRI_UPPER = 255;


// HSV ranges defining the color being detected by this object
private int hueLower, hueUpper, satLower, satUpper,
                                  briLower, briUpper;


// JavaCV elements
private CvMemStorage storage;
private IplImage hsvImg;        // HSV version of webcam image
private IplImage imgThreshed;  // threshold for HSV settings

// bounded box details
private int[] xPoints, yPoints;
private Point center;
private int angle;        // to the horizontal (in degrees)



public ColorRectDetector(int width, int height)
{
  hsvImg = IplImage.create(width, height, 8, 3);    // for HSV image
  imgThreshed = IplImage.create(width, height, 8, 1);
                                        // threshold image
  storage = CvMemStorage.create();

  // storage for the coordinates of the bounded box
  xPoints = new int[NUM_POINTS];
  yPoints = new int[NUM_POINTS];
  center = new Point();
  angle = 0;

  // set default HSV ranges
  hueLower = HUE_LOWER;  hueUpper = HUE_UPPER;
  satLower = SAT_LOWER;  satUpper = SAT_UPPER;
  briLower = BRI_LOWER;  briUpper = BRI_UPPER;
```

```
}  // end of ColorRectDetector()
```

OpenCV uses a non-standard range for hues, from 0 to 179 (rather than to 255). The 179 comes from the way that the HSV color space can be drawn as a cylinder, with the hue represented by an angle around the vertical axis. Instead of spreading the hue around the cylinder's entire circumference (0 to 359 degrees), OpenCV employs a range that covers half of that (0 to 179 degrees). This is explained more fully in "HSV Color Format in OpenCV" by Shervin Emami at http://www.shervinemami.co.cc/colorConversion.html.

The HSV ranges in ColorRectDetector can be modified via three set methods:

```
public void setHueRange(int lower, int upper)
{  hueLower = lower;
   hueUpper = upper;
}

public void setSatRange(int lower, int upper)
{  satLower = lower;
   satUpper = upper;
}

public void setBriRange(int lower, int upper)
{  briLower = lower;
   briUpper = upper;
}
```

4.1. Finding a Rectangular Blob

ColorRectDetector.findRect() is the entry point for blob detection. First, the supplied IplImage is converted to HSV-format. This is thresholded by cvInRangeS() using the HSV ranges to create a binary image whose pixels are white for colors inside all those ranges. The largest blob inside the binary is found, and it's bounding box's attributes are stored.

```
// globals
private boolean foundBox = false;


public boolean findRect(IplImage im)
{
  if (im == null)
    return false;

  // convert to HSV
  cvCvtColor(im, hsvImg, CV_BGR2HSV);

  // threshold image using supplied HSV settings
  cvInRangeS(hsvImg, cvScalar(hueLower, satLower, briLower, 0),
                     cvScalar(hueUpper, satUpper, briUpper, 0),
                                                 imgThreshed);

  cvMorphologyEx(imgThreshed, imgThreshed, null, null,
                                         CV_MOP_OPEN, 1);
  /* erosion followed by dilation to remove specks of white
     while retaining the image's size */
```

© Andrew Davison 2013

```java
    CvBox2D maxBox = findBiggestBox(imgThreshed);

    // extract box details
    if (maxBox != null) {
      foundBox = true;
      extractBoxInfo(maxBox);
    }
    else
      foundBox = false;

    return foundBox;
}  // end of findRect()
```

findBiggestBox() uses the JavaCV function cvFindContours() to create a list of contours. For my binary threshold image, a contour is a region (or blob) of white pixels. Each blob is approximated by a bounding box, and the largest is returned.

```java
// globals
private static final float SMALLEST_BOX =  600.0f;
      // ignore detected boxes smaller than SMALLEST_BOX pixels


private CvBox2D findBiggestBox(IplImage imgThreshed)
{
  CvSeq bigContour = null;

  // generate all the contours in the threshold image as a list
  CvSeq contours = new CvSeq(null);
  cvFindContours(imgThreshed, storage, contours,
                         Loader.sizeof(CvContour.class),
                         CV_RETR_LIST, CV_CHAIN_APPROX_SIMPLE);

  // find the largest box in the list of contours
  float maxArea = SMALLEST_BOX;
  CvBox2D maxBox = null;
  while (contours != null && !contours.isNull()) {
    if (contours.elem_size() > 0) {
      CvBox2D box = cvMinAreaRect2(contours, storage);
      if (box != null) {
        CvSize2D32f size = box.size();
        float area = size.width() * size.height();
        if (area > maxArea) {  // record big contour
          maxArea = area;
          maxBox = box;
          bigContour = contours;
        }
      }
    }
    contours = contours.h_next();  // move to next contour
  }

  // if (bigContour != null)
  //   extractContourInfo(bigContour);   // explained later

  return maxBox;
}  // end of findBiggestBox()
```

cvFindContours() can return different types of contours, collected together in different kinds of data structures. I generate the simplest kind of contours, storing them in a linear list which can be searched with a while loop.

After some experimentation, I placed a lower bound on the bounded box size of 600 square pixels which filters out small boxes surrounding patches of image noise. This means that findBiggestBox() may return null if it doesn't find a large enough box.

The advantage of findBiggestBox() returning a bounding box rather than just a shapeless blob, is that shape calculations are easier. extractBoxInfo() records the box's coordinates, its center, and the angle its longest side makes to the horizontal.

```java
// global bounded box details
private static final int NUM_POINTS = 4;  // no of coords in box

private boolean foundBox = false;
private int[] xPoints, yPoints;
private Point center;
private int angle;      // to the horizontal (in degrees)


private void extractBoxInfo(CvBox2D maxBox)
{
  // store the box's center point
  CvPoint2D32f boxCenter = maxBox.center();
  center.x = Math.round( boxCenter.x());
  center.y = Math.round( boxCenter.y());

  CvPoint2D32f box_vtx =  new CvPoint2D32f(NUM_POINTS);
        // allocate native array using an integer as argument
  cvBoxPoints(maxBox, box_vtx);

  // store the box's corner coordinates
  for (int i = 0; i < NUM_POINTS; i++) {
    box_vtx.position(i);    // use position() method
    xPoints[i] = (int)Math.round( box_vtx.x() );
    yPoints[i] = (int)Math.round( box_vtx.y() );
  }

  angle = calcAngle(xPoints, yPoints);  // store horizontal angle
}  // end of extractBoxInfo()
```

These coordinates, center, and angle can be accessed outside ColorRectDetector by calls to suitable get methods:

```java
public Polygon getBoundedBox()
{  return ((foundBox) ?
            new Polygon(xPoints, yPoints, NUM_POINTS) : null);  }

public Point getCenter()
{  return ((foundBox) ? center : null);  }

public int getAngle()
{  return angle;  }
```

The foundBox boolean will be set to false if no suitable bounded box was found by findRect().

4.2. Calculating the Box's Angle

The CvBox2D object passed to extractBoxInfo() already contains an angle, which would seem to make an entire calcAngle() function a bit superfluous. It's a lot easier just to write:

```
angle = (int) Math.round( Math.toDegrees( maxBox.angle() ));
```

This code fragment converts the box's angle from radians to an integer number of degrees. Unfortunately, this angle isn't quite what I need. It represents the bounded box's angle to the vertical, but I want the angle the *longest* edge makes to the horizontal. This means that I must examine the bounded box's edges inside calcAngle() to find the longest one, and calculate its angle to the horizontal. The algorithm is illustrated by Figure 8.
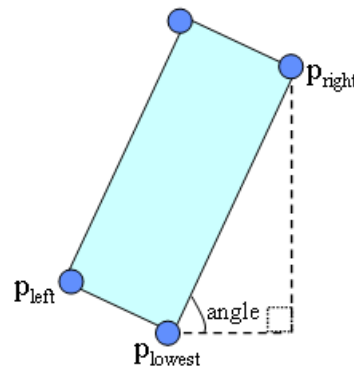

Figure 8. Finding the Horizontal Angle of a Box's Longest Edge.

calcAngle() locates the point in the bounding box with the largest y-value (i.e. p_{lowest} in Figure 8). It then calculates the lengths of the adjacent edges, to find the longest (p_{lowest} to p_{right} in Figure 8). Its angle to the horizontal is easily obtained.

```
private int calcAngle(int[] xPts, int[] yPts)
{
  // find index of point with 'largest' y-value (lowest on screen)
  int idxLowest = 0;
  int yLowest = -1;
  for (int i = 0; i < NUM_POINTS; i++) {
    if (yPts[i] > yLowest) {  // further down has larger y-coord
      yLowest = yPts[i];
      idxLowest = i;
    }
  }

  // get neighboring point indicies
  int idxRight = (idxLowest+1)%4;
  int idxLeft = (idxLowest+3)%4;      // same as -1 but remains +ve

  // calculate length^2 of neighboring sides
  float x = xPts[idxLowest];
  float y = yPts[idxLowest];
```

```
        float xRight = xPts[idxRight];
        float yRight = yPts[idxRight];
        float rightLen2 = (yRight-y)*(yRight-y) + (xRight-x)*(xRight-x);

        float xLeft = xPts[idxLeft];
        float yLeft = yPts[idxLeft];
        float leftLen2 = (yLeft-y)*(yLeft-y) + (xLeft-x)*(xLeft-x);

        // store info about pt along longest side from lowest pt
        int longIdx;
        float xLong, yLong;
        if (rightLen2 > leftLen2) {     // right side is longest
          longIdx = idxRight;
          xLong = xRight;
          yLong = yRight;
        }
        else {  //left side is longest
          longIdx = idxLeft;
          xLong = xLeft;
          yLong = yLeft;
        }

        // calculate angle of longest side to the horizontal
        double radAngle = Math.atan2( (double)(y-yLong),
                                      (double)(xLong-x) );
        return (int) Math.round( Math.toDegrees(radAngle) );
}  // end of calcAngle()
```

A less long-winded approach for finding the angle of the longest edge, is with *moments*. I used spatial moments back in Chapter 4 to find the center of gravity (COG) of a binary image. The same technique can be applied to a contour to find its center (or centroid). I can also calculate second order mixed moments, which give information about the spread of pixels around the centroid. Second order moments can be combined to return the orientation (or angle) of the contour's major axis relative to the x-axis.

Referring back to the moments notation from Chapter 4, the m() moments function is defined as:

$$\text{m}(p, q) = \sum_{i=1}^{n} I(x, y) x^p y^q$$

The function takes two arguments, p and q, which are used as powers for x and y. The I() function is the intensity for a pixel defined by its (x, y) coordinate. *n* is the number of pixels that make up the shape.

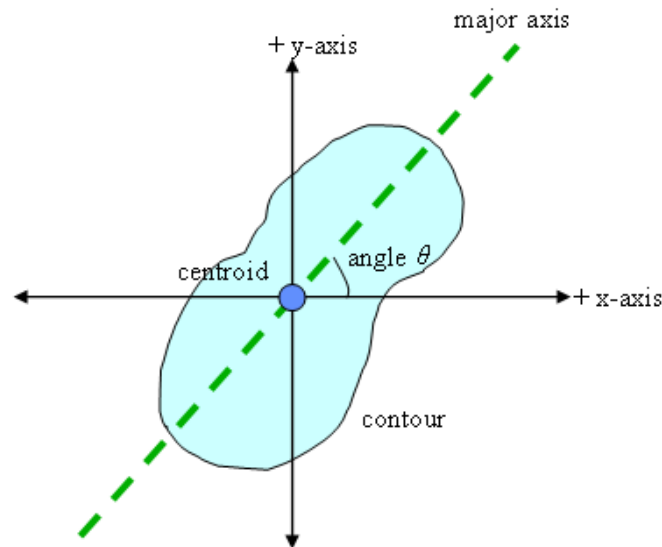If I consider a contour like the one in Figure 9, then *q* is the angle of its major axis to the horizontal.

Figure 9. A Contour and its Major Axis Line.

In terms of the m() function, it can be shown that:

$$\tan 2\theta = \frac{2 \cdot m(1,1)}{m(2,0) - m(0,2)}$$

The extractContourInfo() method shown below uses spatial moments to obtain the contour's centroid, and then utilizes cvGetCentralMoment() to calculate the major axis angle according to the above equation:

```
private void extractContourInfo(CvSeq contour)
// use the contour to get the center and angle
{
  CvMoments moments = new CvMoments();
  cvMoments(contour, moments, 1);

  // for center of gravity
  double m00 = cvGetSpatialMoment(moments, 0, 0);
  double m10 = cvGetSpatialMoment(moments, 1, 0);
  double m01 = cvGetSpatialMoment(moments, 0, 1);

  if (m00 != 0) {    // calculate center
    int xCenter = (int) Math.round(m10/m00);
    int yCenter = (int) Math.round(m01/m00);
    System.out.println("COG: (" + xCenter + ", " + yCenter + ")" );
  }

  // for angle of major axis to the horizontal
  double m11 = cvGetCentralMoment(moments, 1, 1);
  double m20 = cvGetCentralMoment(moments, 2, 0);
  double m02 = cvGetCentralMoment(moments, 0, 2);

  double theta = 0.5 * Math.atan2(2*m11, m20-m02);  // axis eqn
  int thetaDeg = (int) Math.round( Math.toDegrees(theta));
  System.out.println("moment angle: " + thetaDeg);
```

© Andrew Davison 2013

```
}  // end of extractContourInfo()
```

extractContourInfo() can be called by findBiggestBox() after the biggest contour has
been found (the call is commented out in the findBiggestBox() code above).

The contour moment angle is almost the same as the bounded box angle, except that it
has a different sign because the positive y-axis of the contour runs down the screen.

The math behind the major axis angle equation is described in the computer vision
lecture slides for binary images by Robyn Owens at
http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/OWENS/LECT2/node
3.html. Moments in OpenCV are explained in more depth in "Simple Image Analysis
by Moments" by Johannes Kilian at http://serdis.dis.ulpgc.es/~itis-
fia/FIA/doc/Moments/OpenCv/OpenCV_Moments.pdf.

5. Using the ColorRectDetector

HSVSelector uses the same looping rendering thread as I've described before, this
time located in HSVPanel (see Figure 6):

```
// globals
private static final int DELAY = 100;
                  // time (ms) between redraws of the panel

private static final int IMG_SCALE = 2;
        // scaling applied to webcam image before blob detection

private static final int CAMERA_ID = 0;

private HSVSelector top;  // top-level GUI
private volatile boolean isRunning;

// used for the average ms snap time information
private int imageCount = 0;
private long totalTime = 0;

private ColorRectDetector rectDetector;


public void run()    // in HSVPanel
{
  FrameGrabber grabber = initGrabber(CAMERA_ID);
  if (grabber == null)
    return;

  IplImage snapIm;
  IplImage scaleImg = IplImage.create(WIDTH/IMG_SCALE,
                                      HEIGHT/IMG_SCALE, 8, 3);
  long duration;
  isRunning = true;

  while (isRunning) {
   long startTime = System.currentTimeMillis();

    // update detectors HSV settings
    rectDetector.setHueRange(top.getHueLower(), top.getHueUpper());
```

```
    rectDetector.setSatRange(top.getSatLower(), top.getSatUpper());
    rectDetector.setBriRange(top.getBriLower(), top.getBriUpper());

    snapIm = picGrab(grabber, CAMERA_ID);    // take a snap
    imageCount++;
    cvResize(snapIm, scaleImg);
    rectDetector.findRect(scaleImg);
    repaint();

    duration = System.currentTimeMillis() - startTime;
    totalTime += duration;
    if (duration < DELAY) {
      try {
        Thread.sleep(DELAY-duration);
      }
      catch (Exception ex) {}
    }
  }
  closeGrabber(grabber, CAMERA_ID);
}  // end of run()
```

The loop iterates every DELAY milliseconds, and passes a *scaled* version of the image to ColorRectDetector.findRect(). Scaling speeds up the processing time of the blob detector.

The GUI sliders' current HSV ranges are passed to the ColorRectDetector object in each loop iteration. This ensures that any GUI changes affects the blob processing.

5.1. Rendering the Blob

Figure 5 shows the visual elements rendered into the panel – the threshold and HSV images, a bounded box for the blob, and the usual statistics. This is handled by HSVPanel's paintComponent():

```
public void paintComponent(Graphics g)     // in HSVPanel
{
  super.paintComponent(g);
  Graphics2D g2 = (Graphics2D) g;
  g2.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
                   RenderingHints.VALUE_INTERPOLATION_BILINEAR);

  if (rectDetector != null)
    drawImages(g2);

  writeStats(g2);
} // end of paintComponent()
```

drawImages() positions the threshold and HSV images side-by-side and draws a bounded box polygon on top of the threshold image.

```
// globals
private ColorRectDetector rectDetector;


private void drawImages(Graphics2D g2)
{
```

```
    int threshWidth = 0;
    BufferedImage threshIm = rectDetector.getThresholdImage();
    if (threshIm != null) {
      g2.drawImage(threshIm, 0, 0, this);    // draw threshold
      threshWidth = threshIm.getWidth();

      Polygon boxPoly = rectDetector.getBoundedBox();
      if (boxPoly != null) {
        g2.setPaint(Color.YELLOW);
        g2.drawPolygon(boxPoly);    // draw box on threshold image
      }
    }

    // display HSV image to the right of the threshold image
    BufferedImage hsvIm = rectDetector.getHSVImage();
    if (hsvIm != null)
      g2.drawImage(hsvIm, threshWidth, 0, this);
  }  // end of drawImages()
```

The two images, and the polygon, are obtained by calling get methods in ColorRectDetector.

5.2. Storing HSV Information

The HSVSelector GUI includes a "Save Settings" button (see Figure 5), which stores the HSV ranges into a text file. The BlobsDrumming application described in the next section loads two of these settings files – one for the blue card, one for the red.

The code attached to the button writes the current hue, saturation, and brightness ranges into a text file spread over three lines:

```
// in HSVSelector
// globals
private int hueLower, hueUpper, satLower, satUpper,
                                   briLower, briUpper;


public void saveHSVRanges(String fnm)
// write out three lines for the lower/upper HSV values
{
  try {
    PrintWriter out = new PrintWriter( new FileWriter(fnm));
    out.println("hue: " + hueLower + " " + hueUpper);
    out.println("sat: " + satLower + " " + satUpper);
    out.println("val: " + briLower + " " + briUpper);
    out.close();
    System.out.println("Saved HSV ranges to " + fnm);
  }
  catch (IOException e)
  {  System.out.println("Could not save HSV ranges to " + fnm); }
}  // end of saveHSVRanges()
```

6. Drumming with Blobs

The BlobsDrumming application generates percussion sounds based on how the user holds two colored cards up in front of the webcam. The bounded boxes for the cards are detected, and treated as virtual 'drum sticks' which trigger percussion beats.
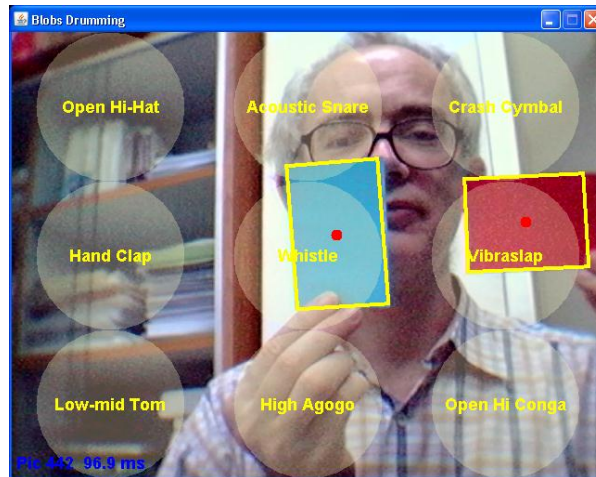

Figure 10. Blob Drumming (a repeat of Figure 1)

The webcam image is overwritten by nine translucent 'drumming' circles (see Figure 10), which dictate what sort of percussion instrument will be set off by a card.

The orientation of a card affects the tempo, with a faster beat assigned to a rectangle rotated away from the vertical. Since the red card shown in Figure 10 is almost horizontal, the vibraslaps occur very rapidly.

Figure 11 gives the UML class diagrams for the application, which reuse several classes from earlier examples.
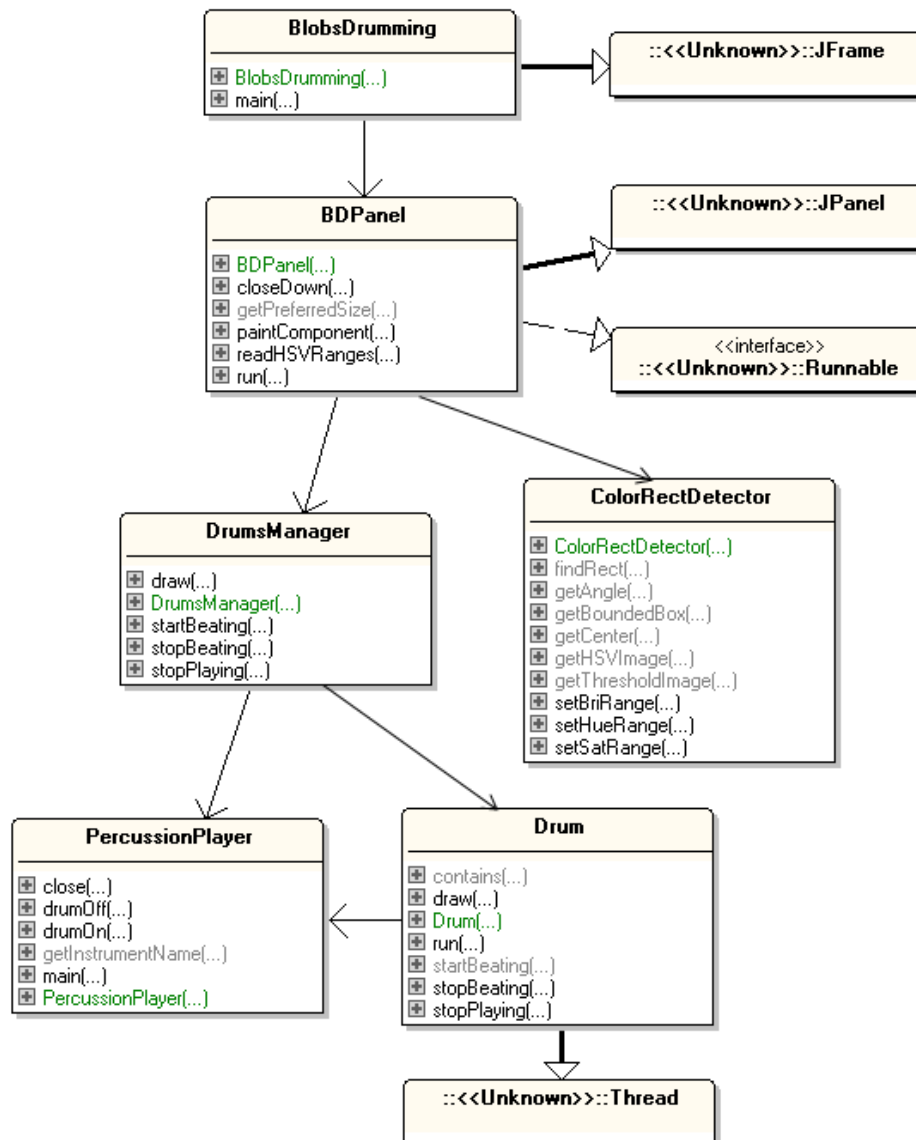
Figure 11. Class Diagrams for BlobsDrumming.

The BlobsDrumming class is the top-level JFrame, and uses BDPanel to display the annotated webcam image.

The webcam image is obtained using JavaCV's FrameGrabber in BDPanel, and the blob detection is done by two instances of ColorRectDetector, one looking for the biggest red rectangle, the other for a light blue one.

The DrumsManager is in charge of nine percussion instruments, each represented by a Drum instance. Audio generation is via calls to a single PercussionPlayer object shared by all the drums. The Drums are threaded, so the methods in PercussionPlayer are synchronized to avoid race conditions when its drumOn() and drumOff() are called simultaneously.

6.1. Drums Creation

When DrumsManager creates the Drum objects, each is supplied with a (x, y) position, width, and height, which they use to draw a translucent circle and text on the BDPanel's panel at render time. The DrumsManager calculates this positional data so that its nine drums are laid out on the panel in a grid shape dictated by the NUM_ROWS and NUM_COLS constants.

```java
// globals
private final static int NUM_ROWS = 3;
private final static int NUM_COLS = 3;

private PercussionPlayer player;
private Drum[] drums;

private int numSticks;
private Drum[] currDrums;    // points to currently playing drum


public DrumsManager(int width, int height, int num)
// width and height of the panel
{
  int colWidth = width/NUM_COLS;    // drawing area for one drum
  int rowWidth = height/NUM_ROWS;
  numSticks = num;      // no. of drums that can be playing at once

  player = new PercussionPlayer();

  // initialize each drum
  drums = new Drum[NUM_ROWS*NUM_COLS];
  int xCoord = 0;      // (xCoord, yCoord) is top-left
  int yCoord = 0;      // of each drum drawing area
  int i = 0;
  for (int row=0; row < NUM_ROWS; row++) {
    xCoord = 0;
    for (int cols=0; cols < NUM_COLS; cols++) {
      drums[i] = new Drum( PercussionPlayer.getInstrumentName(i),
                           xCoord, yCoord,
                           colWidth, rowWidth, player);
      drums[i].start();
      xCoord += colWidth;
      i++;
    }
    yCoord += rowWidth;
  }

  // initialize currently playing drums array
  currDrums = new Drum[numSticks];
  for (int j=0; j < numSticks; j++)
    currDrums[j] = null;   // no drums playing yet
} // end of DrumsManager()
```

DrumsManager also creates a currDrums[] array which will store references to the drums currently being played. The array indicies act as "drum stick" indicies, so the drum stored in currDrums[0] is the one currently being hit by stick 0. The two colored cards are the "drum sticks".

© Andrew Davison 2013

Each Drum object executes in its own thread so that multiple drum beats can be generated at the same time.

6.2. Starting and Stopping Drum Beats

Drum beating is controlled through the DrumsManager methods startBeating() and stopBeating(). startBeating() is called with four arguments: a stick index (which corresponds to the card triggering the beat), the (x, y) coordinate of the card's center, and its angle to the vertical.

startBeating() will initiate a beat if the (x, y) coordinate is over a new drum, and the previous drum beat (if one exists) is stopped. Alternatively, startBeating() may adjust the beat rate of the current drum if the (x, y) coordinate is over the same drum as before but the card's angle to the vertical has changed.

```
// globals
private Drum[] drums;
private int numSticks;
private Drum[] currDrums;      // points to currently playing drums


public void startBeating(int sIdx, int x, int y, int angle)
{
  if ((sIdx < 0) || (sIdx >= numSticks)) {
    System.out.println("No stick with that index (" + sIdx+ ")");
    return;
  }

  if ((currDrums[sIdx] != null) &&
      (currDrums[sIdx].contains(x, y)))  // still in old drum area?
    currDrums[sIdx].startBeating(x, y, angle);  //change beat rate
  else {
    if (currDrums[sIdx] != null) {  // just left old drum area?
      currDrums[sIdx].stopBeating();
      currDrums[sIdx] = null;
    }
    for(Drum drum: drums) {       // inside a new drum area?
      if (drum.startBeating(x, y, angle)) {
        currDrums[sIdx] = drum;
        break;
      }
    }
  }
}  // end of startBeating()
```

stopBeating() uses the supplied stick index to make the corresponding currDrums[] drum stop beating.

```
public void stopBeating(int sIdx)
{
 if ((sIdx < 0) || (sIdx >= numSticks)) {
    System.out.println("No stick with that index (" + sIdx+ ")");
    return;
  }

  if (currDrums[sIdx] != null) {
```

```
        currDrums[sIdx].stopBeating();
        currDrums[sIdx] = null;
    }
}  // end of stopBeating()
```

6.3. Drawing the Drums

As Figure 10 illustrates, BDPanel must draw the webcam image, the statistics, nine drum circles, and highlighted boxes around the two cards. It delegates the rendering of the drum circles to DrumsManager, by calling its draw() method.

```
// in DrumsManager
public void draw(Graphics g)
{ for(Drum drum: drums)
    drum.draw(g);
}
```

DrumsManager passes the drawing task to each Drum object.

6.4. Creating a Single Drum.

The Drum() constructor is supplied with the name of its percussion instrument, a reference to the PercussionPlayer for generating sounds, and coordinate and size details used at render time.

```
// globals
private static final int MAX_DELAY = 250;   // for drum beats in ms

private String drumName;
private int radius, xCenter, yCenter;
private Font msgFont;

// drum beating
private PercussionPlayer player;
private volatile boolean isPlaying = true;
private volatile boolean drumIsBeating = false;
private int repeatDelay = MAX_DELAY;
                        // time between drum beats (in ms)


public Drum(String name, int x, int y, int width, int height,
                                       PercussionPlayer p)
{ drumName = name;
  player = p;

  radius = (width < height) ? width/2 : height/2;
  xCenter = x + width/2;
  yCenter = y + height/2;

  msgFont = new Font("SansSerif", Font.BOLD, 18);
} // end of Drum()
```

An important part of a Drum object's state are two booleans, isPlaying and drumIsBeating. isPlaying signals whether the drum is active (which it should be until

© Andrew Davison 2013

the BlobsDrumming application is closed down), and drumIsBeating states whether the drum is beating or not. Initially, a drum is silent so drumIsBeating is set to false.

DrumsManager creates a Drum object, and immediately starts its thread, which consists of a simple loop.

```
// global
private static final int MAX_DELAY = 250;   // for drum beats in ms
private static final int BEAT_LENGTH = 200;
                             // time of a drum beat (in ms)

private int repeatDelay = MAX_DELAY;


public void run()        // in Drum class
// keep repeating a drum beat
{
  while (isPlaying) {
    if (drumIsBeating) {
      player.drumOn(drumName);
      wait(BEAT_LENGTH);
      player.drumOff(drumName);

      wait(repeatDelay);   // time between drum beats
                           // (this value can vary)
    }
  }
}  // end of run()
```

Each iteration plays a single drum beat and waits an amount of time set by repeatDelay. Initially, each drum will silently loop, since drumIsBeating is false.

6.5. Starting (and Stopping) a Drum Beating

The drum begins beating when its startBeating() method is called.

```
// globals
private int xHit = -1;    // hit location on the drum
private int yHit = -1;
private volatile boolean drumIsBeating = false;


public boolean startBeating(int x, int y, int angle)
{
  double ratio = radiusRatio(x, y);
  if (ratio > 1.0)   // outside drum circle
    return false;

  repeatDelay = angle2Delay(angle);    // adjust repeat delay
  xHit = x; yHit = y;    // set hit coord
  drumIsBeating = true;
  return true;
}  // end of startBeating()
```

startBeating() is passed the card's center point, and begins by checking if it falls within the drum's circle. If it does then the angle of the card from the vertical is

translated into a new repeatDelay value. The center coordinate is saved as (xHit, yHit), and drum beating is turned on.

The position and size of the drum circle are set when the drum is constructed, and radiusRatio() calculates the card's distance from the drum's center as a fraction of the drum's radius. If the fraction exceeds 1, then the card is outside the circle.

```
// globals
private int radius, xCenter, yCenter;

private double radiusRatio(int x, int y)
{
  int xDist = x - xCenter;
  int yDist = y - yCenter;
  return Math.sqrt(xDist*xDist + yDist*yDist)/radius;
}  // end of radiusRatio()
```

The mapping from angle to repeatDelay could be implemented in many ways. I divide the drum circle into five regions, as shown in Figure 12.
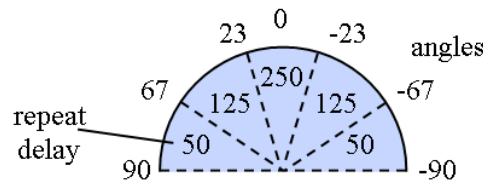

Figure 12. Converting a Vertical Angle to Repeat Delay

The repeat delay is decreased as the vertical angle increases, which causes the drum to beat faster.

```
// global
private static final int MAX_DELAY = 250;   // for drum beats in ms

private int angle2Delay(int angle)
{
  int ang = Math.abs(angle);
  int rate = MAX_DELAY;    // default is slowest rate
  if (ang < 23)
    rate = MAX_DELAY;
  else if (ang < 67)
    rate = 125;
  else if (ang <= 90)
    rate = 50;
  return rate;
}  // end of angle2Delay()
```

Stopping the beat requires that the drumIsBeating boolean is set to false, and the hit coordinate assigned a default position of (-1, -1) outside the drum circle.

```
public void stopBeating()
{
  xHit = -1; yHit = -1;
```

```
    drumIsBeating = false;
}
```

6.6. Drawing a Drum

Figure 10 shows that a drum is made up of three visual elements – a translucent white circle, the name of the percussion instrument positioned at the center of the circle, and a red dot if the drum is beating.

The hardest part is calculating the coordinate where the drum name should be written.

```
// globals
private static final int HIT_SIZE = 12;
private static final Color TRANS_PALE = new Color(255, 255, 200, 75);
                        // translucent whitish color for the drum

// drum info
private String drumName;
private int radius, xCenter, yCenter;
private int xHit = -1;    // hit location on the drum
private int yHit = -1;

// for drawing the drum name
private Font msgFont;
private FontMetrics fm = null;
private int xNamePos, yNamePos;

private volatile boolean drumIsBeating = false;


public void draw(Graphics g)     // in the Drum class
// draw drum and hit location (xHit,yHit)
{
  drawDrum(g);

  if (drumIsBeating) {
    g.setColor(Color.RED);       // draw hit location as a red circle
    g.fillOval(xHit-HIT_SIZE/2, yHit-HIT_SIZE/2, HIT_SIZE, HIT_SIZE);
  }
}  // end of draw()


private void drawDrum(Graphics g)
// draw drum as a circle containing its name at its center
{
  if (fm == null) {   // initialize drum name position using font
    fm = g.getFontMetrics(msgFont);
    xNamePos = xCenter - fm.stringWidth(drumName)/2;
    yNamePos = yCenter + fm.getAscent() - (fm.getAscent() +
                                      fm.getDescent())/2;
  }

  g.setColor(TRANS_PALE);      // draw a translucent circle
  g.fillOval(xCenter-radius, yCenter-radius, radius*2, radius*2);

  // draw name of drum in the center of the circle
  g.setColor(Color.YELLOW.brighter());
  g.setFont(msgFont);
  g.drawString(drumName, xNamePos, yNamePos);
```

```
}  // end of drawDrum()
```

The font metric, fm, will be null the first time that drawDrum() is called, which causes it to be calculated, along with the drawing position (xNamePos, yNamePos) for the instrument name.

The text positioning code could be more sophisticated. As it stands, a very long instrument name may extend beyond the drum circle, and overlap adjacent circles.

6.7. Bringing Drums and Detectors Together

The BDPanel class links the drums and detectors. It uses DrumsManager to create the drums, and manage communicate with them, and creates two instances of ColorRectDetector to detect the red and blue cards.

As usual, the important code is located inside the class' run() method, where it grabs, processes, and renders the webcam image.

```
// in BDPanel
// globals
private static final int IMG_SCALE = 2;
                        // scaling applied to webcam image

private static final int NUM_DETECTORS = 2;
            /* each detector will find a colored rectangle
               in the image */

private DrumsManager drummer;   // manages all the drums


public void run()
{
  FrameGrabber grabber = initGrabber(CAMERA_ID);
  if (grabber == null)
    return;

  // initialize drums display and rectangles detectors
  drummer = new DrumsManager(WIDTH, HEIGHT, NUM_DETECTORS);
       // create drums, and use NUM_DETECTORS sticks to hit them

  initDetectors(WIDTH/IMG_SCALE, HEIGHT/IMG_SCALE);

  IplImage scaleImg = IplImage.create(WIDTH/IMG_SCALE,
                                     HEIGHT/IMG_SCALE, 8, 3);
  long duration;
  isRunning = true;
  while (isRunning) {
    long startTime = System.currentTimeMillis();

    snapIm = picGrab(grabber, CAMERA_ID);
    imageCount++;
    cvResize(snapIm, scaleImg);
    updateDetectors(scaleImg);
    repaint();

    duration = System.currentTimeMillis() - startTime;
    totalTime += duration;
    if (duration < DELAY) {
```

```
        try {
          Thread.sleep(DELAY-duration);
        }
        catch (Exception ex) {}
    }
  }

  closeGrabber(grabber, CAMERA_ID);
  drummer.stopPlaying();    // stop the drums playing
}  // end of run()
```

run() looks pretty much the same as usual since most of the drum and detector code is hidden inside other methods, such as updateDetectors(). Initially the method creates a DrumsManager instance, and calls initDetectors() to create two detectors. The DrumsManager is supplied with the dimensions of the entire panel, whereas the detectors take scaled dimensions because they'll be processing scaled webcam images.

initDetectors() creates two ColorRectDetectors, initializing their HSV ranges by reading values from text files. The files (redHSV.txt and blueHSV.txt) were previously created using the HSVSelector application described earlier.

```
// globals
private static final int NUM_DETECTORS = 2;
            /* each detector will find a colored rectangle
               in the image */

private ColorRectDetector[] detectors;
private boolean haveDetectors = false;


private void initDetectors(int recWidth, int recHeight)
{
  detectors = new ColorRectDetector[NUM_DETECTORS];

  detectors[0] = new ColorRectDetector(recWidth, recHeight);
  readHSVRanges("redHSV.txt", detectors[0]);

  detectors[1] = new ColorRectDetector(recWidth, recHeight);
  readHSVRanges("blueHSV.txt", detectors[1]);

  haveDetectors = true;
}  // end of initDetectors()
```

The ordering of the detectors in the detectors[] array means that the red card will correspond to stick 0 and the blue card will be stick 1.

6.8. Updating the Detectors and the Drums

updateDetectors() passes the scaled webcam image to each of the detectors. If a detector finds a rectangular colored blob, then its center and angle are passed to the drums manager.

```
// globals
private static final int IMG_SCALE = 2;  // scaling for webcam image
```

```
private static final int NUM_DETECTORS = 2;

private ColorRectDetector[] detectors;
private DrumsManager drummer;


private void updateDetectors(IplImage scaleImg)
{
  Point center;
  for (int i=0; i < NUM_DETECTORS; i++) {
    if (detectors[i].findRect(scaleImg)) {  // blob found?
      center = detectors[i].getCenter();   // start beating
      drummer.startBeating(i, center.x*IMG_SCALE,
                              center.y*IMG_SCALE,   // undo scaling
                (detectors[0].getAngle() - 90));
                            // change to vertical angle
    }
    else  // blob not found; stop drum beating for that detector
      drummer.stopBeating(i);
  }
}  // end of updateDetectors()
```

The image passed to the detector is scaled down by IMG_SCALE, so the returned center must be enlarged before being passed to the drums manager. Also, the angle returned by the detector is the rectangle's angle to the horizontal; subtracting 90 converts it to a vertical offset.

The drums manager decides which drum to start beating. However, if a card isn't found then the manager is told to stop whatever drum is assigned to its stick index.

6.9. Drawing the Detectors and Drums

BDPanel's paintComponent() draws the webcam image, the camera statistics, multiple drum circles, and highlighted boxes around the two cards.

```
// globals
private IplImage snapIm;              // current webcam snap
private DrumsManager drummer;         // manages the drums
private boolean haveDetectors = false;


public void paintComponent(Graphics g)  // in BDPanel class
{
  super.paintComponent(g);
  Graphics2D g2 = (Graphics2D) g;
  g2.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
                    RenderingHints.VALUE_INTERPOLATION_BILINEAR);

  if (snapIm!= null)
    g2.drawImage(snapIm.getBufferedImage(), 0, 0, this);

  if (drummer != null)
    drummer.draw(g2);       // draw all the drums

  if (haveDetectors)
    drawBoxes(g2);          // draw the blob boxes

  writeStats(g2);
```

© Andrew Davison 2013

```
} // end of paintComponent()
```

DrumsManager draws the drums, which leaves drawBoxes() to render the bounded boxes around the cards.

```
// globals
private static final int IMG_SCALE = 2;
                         // scaling applied to webcam image

private ColorRectDetector[] detectors;
                         // for detecting the colored rects


private void drawBoxes(Graphics2D g2)
/* draw yellow outline boxes around the locations of the
   detected colored boxes */
{
  g2.setPaint(Color.YELLOW);
  g2.setStroke(new BasicStroke(4));  // thick yellow pen

  Polygon bbox;
  for(ColorRectDetector detector : detectors) {
    bbox = detector.getBoundedBox();
    if (bbox != null) {
      for (int i = 0; i < bbox.npoints; i++) {
        // scale pts back to full size
        bbox.xpoints[i] *= IMG_SCALE;
        bbox.ypoints[i] *= IMG_SCALE;
      }
      g2.drawPolygon(bbox);      // draw bounding box onto panel
    }
  }
}  // end of drawBoxes()
```

If a bounded box has been found, then it will be returned as a Polygon by ColorRectDetector.getBoundedBox(). This is drawn with Graphics.drawPolygon() after its coordinates have been scaled up.