NUI Chapter 7. Face Detection and Tracking

[**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.]

This chapter explains how a face can be tracked. As with earlier examples, I'll grab frames from the webcam, and draw them rapidly onto a panel. At the same time, a detector analyzes the frames to find a face and highlight it in the panel. The application, called FaceTracker, is shown in Figure 1.

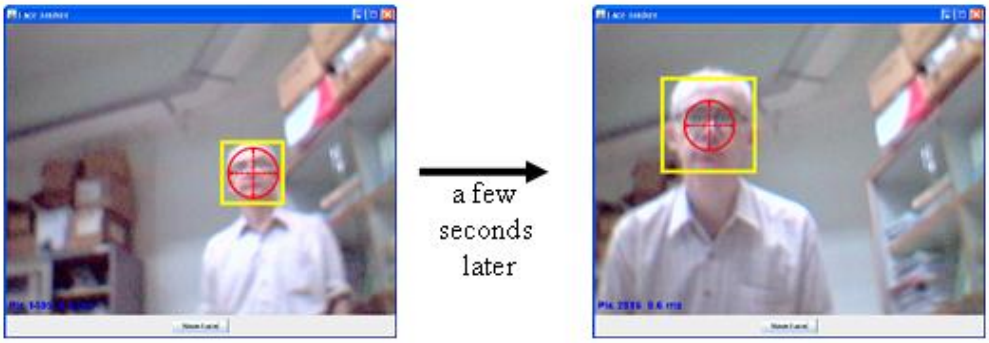

Figure 1. Face Tracking over Time.

The tracker draws a yellow rectangle around the face, and red crosshairs centered inside the rectangle.

The detection code is fast when there's a face present in the image (around 40ms), but may take substantially longer to decide there's no face (as much as 200ms). Two important aspects of the coding are finding ways to speed up the detection, and making sure that lengthy detection processing don't slow down the rest of the program (in particular, the rendering of successive images onto the panel).

The next chapter will extend the processing to *recognize* the tracked face. The distinction between face detection and recognition is that recognition returns a name for the face.

Detection is carried out by a Haar classifier, pre-trained to find facial features (when viewed front-on). The classifier's training requires a great deal of time, but thankfully I can skip that stage because I'm using a face classifier that's already part of OpenCV.

1. Face Detection

The FaceDetection.java example described in this section reads an image from a file, locates all the faces in the picture. It draws yellow rectangles around them, then writes the modified image out to a new JPEG file. Figure 2 shows an example image before and after the faces have been identified.

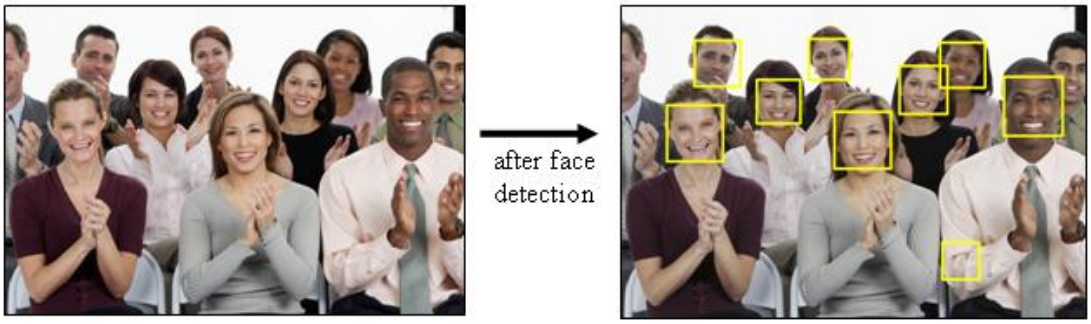

Figure 2. Finding Faces in an Image.

The face detection uses the same Haar classifier that I'll be employing later on in my tracker application, and Figure 2 highlights some of the strengths and weaknesses of the approach. Multiple faces can be found easily and quickly, but only if a face is almost level and almost completely visible. For instance, the classifier failed to label the men at the left and right edges of the image because too much of their faces are missing or obscured. Also, a Haar classifier can often return false positives – highlighted areas which are not faces. This can be seen in the right hand image of Figure 2, where a crinkled shirt elbow is misidentified.

The code for FaceDetection.java:

```
import com.googlecode.javacv.*;
import com.googlecode.javacv.cpp.*;
import com.googlecode.javacpp.Loader;

import static com.googlecode.javacv.cpp.opencv_core.*;
import static com.googlecode.javacv.cpp.opencv_imgproc.*;
import static com.googlecode.javacv.cpp.opencv_highgui.*;
import static com.googlecode.javacv.cpp.opencv_objdetect.*;


public class FaceDetection
{
  private static final int SCALE = 2;
     // scaling factor to reduce size of input image

  // cascade definition for face detection
  private static final String CASCADE_FILE =
                       "haarcascade_frontalface_alt.xml";

  private static final String OUT_FILE = "markedFaces.jpg";


  public static void main(String[] args)
  {
    if (args.length != 1) {
```



```
      System.out.println("Usage: run FaceDetection <input-file>");
      return;
    }

    // preload the opencv_objdetect module to work around a known bug
    Loader.load(opencv_objdetect.class);

    // load an image
    System.out.println("Loading image from " + args[0]);
    IplImage origImg = cvLoadImage(args[0]);

    // convert to grayscale
    IplImage grayImg =
                cvCreateImage(cvGetSize(origImg), IPL_DEPTH_8U, 1);
    cvCvtColor(origImg, grayImg, CV_BGR2GRAY);

    // scale the grayscale (to speed up face detection)
    IplImage smallImg = IplImage.create(grayImg.width()/SCALE,
                          grayImg.height()/SCALE, IPL_DEPTH_8U, 1);
    cvResize(grayImg, smallImg, CV_INTER_LINEAR);

    // equalize the small grayscale
    cvEqualizeHist(smallImg, smallImg);

    // create temp storage, used during object detection
    CvMemStorage storage = CvMemStorage.create();

    // instantiate a classifier cascade for face detection
    CvHaarClassifierCascade cascade =
           new CvHaarClassifierCascade(cvLoad(CASCADE_FILE));
    System.out.println("Detecting faces...");
    CvSeq faces = cvHaarDetectObjects(smallImg, cascade, storage,
                            1.1, 3, CV_HAAR_DO_CANNY_PRUNING);

    cvClearMemStorage(storage);

    // draw thick yellow rectangles around all the faces
    int total = faces.total();
    System.out.println("Found " + total + " face(s)");
    for (int i = 0; i < total; i++) {
      CvRect r = new CvRect(cvGetSeqElem(faces, i));
      cvRectangle(origImg, cvPoint( r.x()*SCALE, r.y()*SCALE ),
                       cvPoint( (r.x() + r.width())*SCALE,
                                (r.y() + r.height())*SCALE ),
                       CvScalar.YELLOW, 6, CV_AA, 0);
          // undo image scaling when calculating rect coordinates
    }

    if (total > 0) {
      System.out.println("Saving marked-faces version of " +
                                    args[0] + " in " + OUT_FILE);
      cvSaveImage(OUT_FILE, origImg);
    }
  }  // end of main()

}  // end of FaceDetection class
```

The image preprocessing consists of three steps: conversion of the color input image to grayscale (necessary for the subsequent equalization and Haar detection), scaling to reduce the size of the image (and thereby reduce the detection time), and grayscale

equalization. Equalization examines the image's range of grayscale values and widens them to cover more of the total range from black to white. The result is an image with larger contrasts between similarly shaded areas, which makes object detection easier later on.

1.1. Background on Haar Classification

Although I've mentioned Haar classification a few times, I haven't explained how it finds faces. In fact, I've been selling it short, because it can be used for much more than just face detection. A Haar classifier can be trained to detect most types of 'blocky', fairly rigid objects, such as cars, motorbikes, and parts of the human body such as the eyes and mouth. It's less great at recognizing structures with tree-like branches such as hands, or smooth shapes containing very little texturing, lines, or varying sub-regions.

Good training involves using many *thousands* of high-quality positive images. For face detection, this means close-up pictures of heads which all have a very similar front-facing pose with little background variation. Eyes, noses, and mouths should all be in roughly the same position across all the pictures, and the images should be the same size. You also need to train the classifier with a similar number of negative images (pictures without faces).

As you might suspect, all this training may take tens of hours, or even longer! The good news is that OpenCV comes with several pre-trained Haar classifiers for different objects, including faces, relieving me of any trainer work. You'll find the classifiers in the OpenCV download, in the directory <OpenCV>\data\haarcascades\. A source for other classifiers is http://alereimondo.no-ip.org/OpenCV/34.

If you really want to train your own classifier, then there's a good description of the necessary steps in Chapter 13 of *Learning OpenCV* by Bradski and Kaehler. Online resources include Naotoshi Seo's excellent tutorial at http://note.sonots.com/SciSoftware/haartraining.html

One reason for the Haar classifier's speed is that it looks for features represented by *rectangular groups* of pixels (some typical examples are shown in Figure 3) rather than individual pixels.

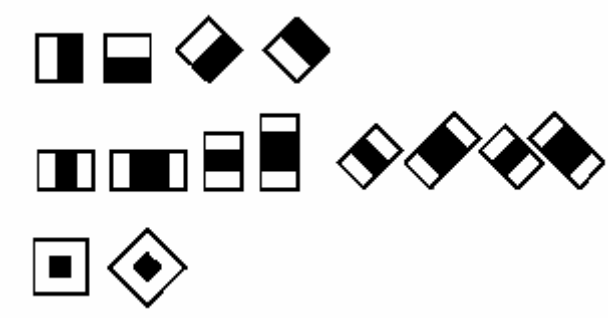

Figure 3. Some Common Haar Features.

These rectangular patterns can be scaled so that different feature sizes can be found using the same approach. Detection is made faster by converting the picture into an *integral* image, where a given (x, y) pixel contains the sum of all the original picture's

pixel intensities in the rectangle between (0, 0) and (x, y). This makes checking for feature rectangles a fast, simple operation involving only a few additions and subtractions. The Haar features that are rotated by 45 degrees (see Figure 3 for some examples) require the original picture to be rotated before being converted to an integral.

The use of integrals is fast and efficient but calculating the 100,000 possible features in even a small image is still too time-consuming, Fortunately, its possible to drastically reduce the number of features that need to be tested to decide whether an image contains an object (e.g. a face). Feature testing is organized into a *cascade* (something like a binary tree), where the cascade's root node contains the test that has proven the best at finding an object during training. If an image isn't rejected by this test then it's passed down the cascade to the second-best test, and so on. Only if an image reaches the end of all the tests without being rejected, is it deemed to contain an object.

The main drawback of Haar classification is the relatively high negative hit rate – an object is detected which isn't really in the image. We can see that in Figure 2, where the face detector decides somebody's elbow is a face.

Haar classification was first developed by Paul Viola and Michael Jones, and so is sometimes known as Viola-Jones detection. A copy of their paper can be downloaded from http://research.microsoft.com/en-us/um/people/viola/Pubs/Detect/violaJones_CVPR2001.pdf. OpenCV documentation on Haar classification is located at http://opencv.willowgarage.com/documentation/c/objdetect_cascade_classification.html

1.2. Using the Haar Classifier

If you look back at the FaceDetection.java example, the code related to the classifier is:

```
// cascade definition for face detection
private static final String CASCADE_FILE =
                        "haarcascade_frontalface_alt.xml";

         :
// instantiate a classifier cascade for face detection
CvHaarClassifierCascade cascade =
             new CvHaarClassifierCascade(cvLoad(CASCADE_FILE));

CvSeq faces = cvHaarDetectObjects(smallImg, cascade, storage,
                             1.1, 3, CV_HAAR_DO_CANNY_PRUNING);
```

The classifier information is stored as an XML file in the local directory, and loaded by the CvHaarClassifierCascade() constructor. I copied the XML file from <OpenCV>\data\haarcascades\. Some developers recommend using haarcascade_frontalface_alt2.xml instead, but I found no difference between them in my tests.

cvHaarDetectObjects() finds rectangular regions in the input image that contain objects recognized by the classifier. The returned data structure is a sequence (list) of these rectangles.

The classifier scans the image several times at different scales, controlled by the fourth argument (the value 1.1 in the code above). Increasing the scale factor (e.g. to 1.2) will make the classifier run faster, but also increase the chance that it might miss a feature at a certain size.

The fifth argument (the value 3 in the example above) is the number of overlapping detections needed before a region is deemed to contain an object. Reducing this value will increase processing speed, but increases the chance of negative hits.

The Canny pruning argument specifies that regions with no lines are to be skipped, thereby speeding up the search.

1.3. Accessing the Face Information

The CvSeq JavaCV data type is something like a Java ArrayList of Rectangle objects in this example, and is manipulated in an analogous way. A loop iterates through the sequence, accessing each rectangle:

```
int total = faces.total();
for (int i = 0; i < total; i++) {
  CvRect r = new CvRect(cvGetSeqElem(faces, i));
  cvRectangle(origImg, cvPoint( r.x()*SCALE, r.y()*SCALE ),
                  cvPoint( (r.x() + r.width())*SCALE,
                           (r.y() + r.height())*SCALE ),
                  CvScalar.YELLOW, 6, CV_AA, 0);
    // undo image scaling when calculating rect coordinates
}
```

JavaCV's CvRect data type is similar to Java's Rectangle class – it stores the top-left hand corner of the rectangle as a (x, y) coordinate, and the rectangle's width and height. These are drawn onto the original image (the one loaded from the file), using OpenCV's cvRectangle() drawing function. One tricky aspect is to remember to undo the scaling of the rectangle values, so they match the original image's unscaled dimensions.

2. The Face Tracker

My tracker application (see Figure 1) captures webcam snaps with JavaCV's FrameGrabber and then performs face detection using code similar to the previous section. The class diagrams for the application are given in Figure 4.

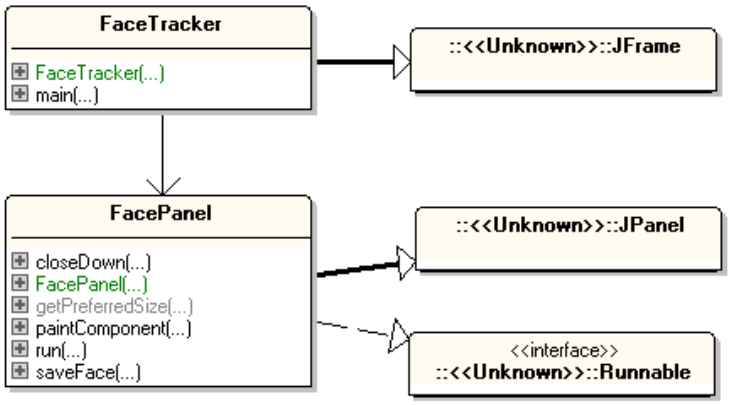

Figure 4. Class Diagrams for the FaceTracker Application.

I won't bother explaining the top-level FaceTracker class – it's a standard JFrame which creates the FacePanel object, and a button. Pressing the button, labeled as "Save Face", makes FacePanel save the currently highlighted face (i.e. the subimage inside the yellow rectangle) to a file.

The FacePanel class spends much of it's time inside a threaded loop which repeatedly grabs an image from the webcam and draws it onto the panel until the window is closed. FacePanel differs from similar panel classes in earlier examples in one important way. Since face detection is such a time consuming process, it is farmed out to a separate thread that uses a mixture of Java 2D and JavaCV. The rest of this chapter will describe these aspects in more detail.

2.1. Initializing the Detector

When cvHaarDetectObjects() is eventually called, it has two prerequisites that I can deal with at start-up time: I load the classifier's XML file, and create dynamic storage which will be allocated as the function progresses. This occurs in the initDetector() method, called from FacePanel's constructor:

```
// globals
// classifier for face detection
private static final String FACE_CASCADE_FNM =
                              "haarcascade_frontalface_alt.xml";
                            // "haarcascade_frontalface_alt2.xml";

private CvHaarClassifierCascade classifier;
private CvMemStorage storage;
private CanvasFrame debugCanvas;


private void initDetector()
{
  // instantiate a classifier cascade for face detection
  classifier = new CvHaarClassifierCascade(cvLoad(FACE_CASCADE_FNM));
  if (classifier.isNull()) {
    System.out.println("\nCould not load: " + FACE_CASCADE_FNM);
    System.exit(1);
  }
```



```
  storage = CvMemStorage.create();
     // create storage used during object detection

  // debugCanvas = new CanvasFrame("Debugging Canvas");
}  // end of initDetector()
```

The code for the creation of a debugCanvas object is commented out. It was used during debugging to show the intermediate stages in an image's transformation. CanvasFrame is a useful way of quickly displaying an image without creating additional GUI elements in the Swing application.

2.2. The Display Loop

The FacePanel() constructor invokes a thread which starts the webcam display loop inside run(). The method is similar to what we've seen before, except when it passes the snapped image to trackFace() for processing (shown in bold in the following):

```
// globals
private static final int DELAY = 100;
                   // time (ms) between redraws of the panel
private static final int CAMERA_ID = 0;

private static final int DETECT_DELAY = 500;
               // time (ms) between each face detection
private static final int MAX_TASKS = 4;
               // max no. of tasks that can be waiting to be executed

private IplImage snapIm = null;
private volatile boolean isRunning;

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

// used for thread that executes the face detection
private AtomicInteger numTasks;
       // used to record number of detection tasks
private long detectStartTime = 0;


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

  long duration;
  isRunning = true;

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

    snapIm = picGrab(grabber, CAMERA_ID);

    if (((System.currentTimeMillis() - detectStartTime) >
                                                DETECT_DELAY) &&
```



```
        (numTasks.get() < MAX_TASKS))
      trackFace(snapIm);
    imageCount++;
    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()
```

Face detection, even after various speed optimizations, can still take a 200ms to fail to find anything. Such a lengthy delay would severely affect run()'s display loop, which is meant to draw a new image onto the panel roughly every DELAY (100) ms.

I get around that problem by utilizing a separate thread to execute the work inside trackFace() (see below for details), allowing the display loop to progress without delay. Altogether, FaceTracker utilizes three threads – the GUI event thread, a thread containing the display loop in run(), and a face detection thread inside trackFace().

Due to the time-consuming nature of trackFace()'s work, I limit its call frequency to once every DETECT_DELAY (500) ms. I'll explain the other part of the if-test around the trackFace() call – the test of the numTasks atomic integer – shortly.

trackFace()'s threaded nature means that its call in run() will return almost immediately, before the detection task has been completed. As a consequence, the duration calculated inside run() doesn't include detection time.

2.3. Tracking a Face in a Thread

The simplest way of implementing a threaded detection task is to fire off a new thread each time an image needs to be analyzed. This is almost certainly not a good idea since we don't know whether the underlying OpenCV library (i.e. the C code inside OpenCV's DLLs) is capable of dealing with multiple detection tasks being carried out *at the same time*.

It's hard to test OpenCV's robustness in the face of concurrency, since any problems depend on how multiple calls overlap in their use of global data structures, DLLs, and the underlying OS. It's better to avoid the problem altogether by enforcing a restriction that only one detection task can execute inside the detection thread at a time; pending tasks will have to queue up to wait their turn.

Since Java 5, it's been easy to create threads with this kind of behavior, by using an ExecutorService object to manage a single threaded executor:

```
// global
private ExecutorService executor;

// in the FacePanel() constructor
```



```
executor = Executors.newSingleThreadExecutor();
```

The factory method, Executors.newSingleThreadExecutor(), creates an executor consisting of a single worker thread taking tasks off an unbounded queue one at a time. Tasks are guaranteed to execute sequentially, and no more than one task will be active at any given time.

One way of improving this execution scenario is to limit the length of the task queue, since we don't want an unbounded number of detection tasks waiting to be processed. The queue length can be limited by using an atomic integer as a counter to record the number of tasks currently on the queue:

```
// globals
private static final int MAX_TASKS = 4;
       // max no. of tasks that can be waiting to be executed

private AtomicInteger numTasks;
       // used to record number of detection tasks


// in the FacePanel constructor()
numTasks = new AtomicInteger(0);
```

nunTasks is atomic since both the webcam display and detection threads are able to modify it. I don't want problems to arise if they try to manipulate the integer at the same time.

The second half of the if-test around the call to trackFace() implements the bounded queue requirement:

```
if (((System.currentTimeMillis() - detectStartTime) >
                                          DETECT_DELAY) &&
    (numTasks.get() < MAX_TASKS))
  trackFace(im);
```

trackFace() is only called if there are less than MAX_TASKS (4) tasks associated with the executor – one running and three waiting.

2.4. Detecting a Face

The face detection code inside trackFace() is in a run() method. It's invocation is added to the executor's queue as a pending task when trackFace() is called:

```
// globals
private IplImage grayIm;
private volatile boolean saveFace = false;
                  // set by the "Save Face" button


private void trackFace(final IplImage img)
{
  grayIm = scaleGray(img);
  numTasks.getAndIncrement();
      // increment no. of tasks before entering queue
```



```
  executor.execute(new Runnable() {
    public void run()
    {
      detectStartTime = System.currentTimeMillis();
      CvRect rect = findFace(grayIm);
      if (rect != null) {
        setRectangle(rect);
        if (saveFace) {
          clipSaveFace(img);
          saveFace = false;
        }
      }
      long detectDuration =
               System.currentTimeMillis() - detectStartTime;
      System.out.println(" detect time: " + detectDuration + "ms");
      numTasks.getAndDecrement();
           // decrement no. of tasks since finished
    }
  });
}  // end of trackFace()
```

The hard work of face detection by the Haar classifier is hidden away in findFace(), which returns a single JavaCV rectangle object. This information is stored by setRectangle() for later rendering onto the panel, and the clipped face is saved if the "Save Face" button has been pressed.

The task counter, numTasks, is incremented outside the run() method since I want to record the number of tasks queuing as well as the one currently executing. However the counter is decremented at the end of the task (the last line of run()).

The Haar classifier requires a grayscale image, which is generated by scaleGray() before the thread starts. scaleGray() also reduces the image's size, to speed up the processing, and equalizes it. The code is very similar to that performed in the earlier FaceDetector example.

The time taken by the classifier is printed to standard output. On my slow test machine, finding a face usually took 20-50ms while failing to find one could take between 70-200ms. These times indicate that I could increase the detection activation frequency which is currently set at once every DETECT_DELAY (500) ms.

2.5. Finding a Face

The findFace() method calls the Haar classifier, and extracts a single rectangle from the result.

```
private CvRect findFace(IplImage grayIm)
{
  // Haar classification
  CvSeq faces = cvHaarDetectObjects(grayIm, classifier, storage,
                 1.1, 1, CV_HAAR_DO_ROUGH_SEARCH |
                         CV_HAAR_FIND_BIGGEST_OBJECT );
     /* speed things up by searching for only a single,
        largest face subimage */

  int total = faces.total();
  if (total == 0) {
```



```
    System.out.println("No faces found");
    return null;
  }
  else if (total > 1)
    System.out.println("Multiple faces detected (" + total
                                        + "); using the first");
  else
    System.out.println("Face detected");

  CvRect rect = new CvRect(cvGetSeqElem(faces, 0));  //get rectangle

  cvClearMemStorage(storage);
  return rect;
}  // end of findFace()
```

The arguments of the cvHaarDetectObjects() call are a little different from those in my earlier FaceDetection.java example. The fifth argument sets the number of overlapping detections needed before a region is deemed to contain an object to only 1 (it was 3 in FaceDetection). Reducing this value increases processing speed, but increases the chance of negative hits.

The final argument is an OR'ed combination of CV_HAAR_DO_ROUGH_SEARCH and CV_HAAR_FIND_BIGGEST_OBJECT which signals that only the largest object need be returned, and that a faster search is preferred. Canny pruning isn't included since it interacts unfavorably with the rough search setting.

During debugging, it was useful to display the JavaCV image utilized in findFace(). I added the following lines at the start of the method:

```
// show the grayscale
debugCanvas.showImage(grayIm);
debugCanvas.waitKey(0);
```

2.6. Saving a Rectangle

setRectangle() extracts the face rectangle's coordinates ((x, y), width, height) from the JavaCV data structure and stores them in a Java Rectangle object. In the process, the data is enlarged, so it has the same scale as the original snapped image.

```
// globals
private static final int IM_SCALE = 4;
private static final int SMALL_MOVE = 5;

private Rectangle faceRect;   // holds coords of highlighted face


private void setRectangle(CvRect r)
{
  synchronized(faceRect) {
    int xNew = r.x()*IM_SCALE;
    int yNew = r.y()*IM_SCALE;
    int widthNew = r.width()*IM_SCALE;
    int heightNew = r.height()*IM_SCALE;

    // calculate movement of new rectangle compared to previous one
    int xMove = (xNew + widthNew/2) -
                        (faceRect.x + faceRect.width/2);
```



```
    int yMove = (yNew + heightNew/2) -
                        (faceRect.y + faceRect.height/2);

    // report movement only if it is 'significant'
    if ((Math.abs(xMove)> SMALL_MOVE) ||
        (Math.abs(yMove) > SMALL_MOVE))
      System.out.println("Movement (x,y): (" +
                                xMove + "," + yMove + ")" );
    faceRect.setRect( xNew, yNew, widthNew, heightNew);
  }
}  // end of setRectangle()
```

Perhaps the most mysterious aspect of setRectangle() is its use of a synchronized block. It's present because there's a possibility that faceRect can be used in two threads at the same time. The face detection thread calls setRectangle() and clipSaveFace() which both access faceRect, while the Java GUI thread needs it for drawing.

The movement of the current face rectangle compared to the last one is calculated in setRectangle(), but not used elsewhere in the application. I included the code since such information would be useful in more complex face tracking applications.

2.7. Rendering the Highlighted Face

Figure 5 shows a typical rendering of the highlighted face in FaceTracker.

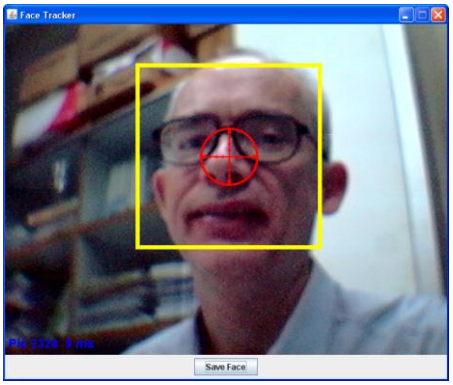

Figure 5. A Highlighted Face.

The panel contains four elements: the webcam image in the background, a yellow rectangle, a red crosshairs image, and statistics written at the bottom left corner.

All rendering is done through calls to the panel's paintComponent():

```
// global
private IplImage snapIm = null;   // current webcam snap


public void paintComponent(Graphics g)
{
  super.paintComponent(g);
  Graphics2D g2 = (Graphics2D) g;

  g2.setFont(msgFont);
```



```
  // draw the image, stats, and detection rectangle
  if (snapIm != null) {
    g2.setColor(Color.YELLOW);
    g2.drawImage(snapIm.getBufferedImage(), 0, 0, this);
    String statsMsg = String.format("Snap Avg. Time:  %.1f ms",
                              ((double) totalTime / imageCount));
    g2.drawString(statsMsg, 5, HEIGHT-10);
                      // write statistics in bottom-left corner
    drawRect(g2);
  }
  else  {  // no image yet
    g2.setColor(Color.BLUE);
    g2.drawString("Loading from camera " + CAMERA_ID +
                                          "...", 5, HEIGHT-10);
  }
} // end of paintComponent()
```

drawRect() is in charge of drawing the yellow rectangle and the crosshairs:

```
// global
private Rectangle faceRect;     // coords of the highlighted face

private void drawRect(Graphics2D g2)
{
  synchronized(faceRect) {
    if (faceRect.width == 0)
      return;

    // draw a thick yellow rectangle
    g2.setColor(Color.YELLOW);
    g2.setStroke(new BasicStroke(6));
    g2.drawRect(faceRect.x, faceRect.y,
                          faceRect.width, faceRect.height);

    int xCenter = faceRect.x + faceRect.width/2;
    int yCenter = faceRect.y + faceRect.height/2;
    drawCrosshairs(g2, xCenter, yCenter);
  }
}  // end of drawRect()
```

drawRect() uses a synchronized block for the same reason as setRectangle() earlier – I don't want its access to the rectangle information to be affected by other threads.

Standard Java2D code is used to draw the rectangle, replacing my earlier use of JavaCV's cvRectangle() in FaceDetection.java.

drawCrosshairs() draws a pre-loaded PNG image (see Figure 6) so it's centered at the given coordinates.

Figure 6. The Crosshairs Image.

