Chapter NUI 8.6. Facial Features Recognition

This chapter describes two ways of detecting facial features such as the eyes, eyebrows, nose, mouth, and chin. The first employs Haar cascade classifiers, in much the same way as I coded face detection and eye tracking in earlier chapters. Three examples of what's possible are shown in Figure 1.

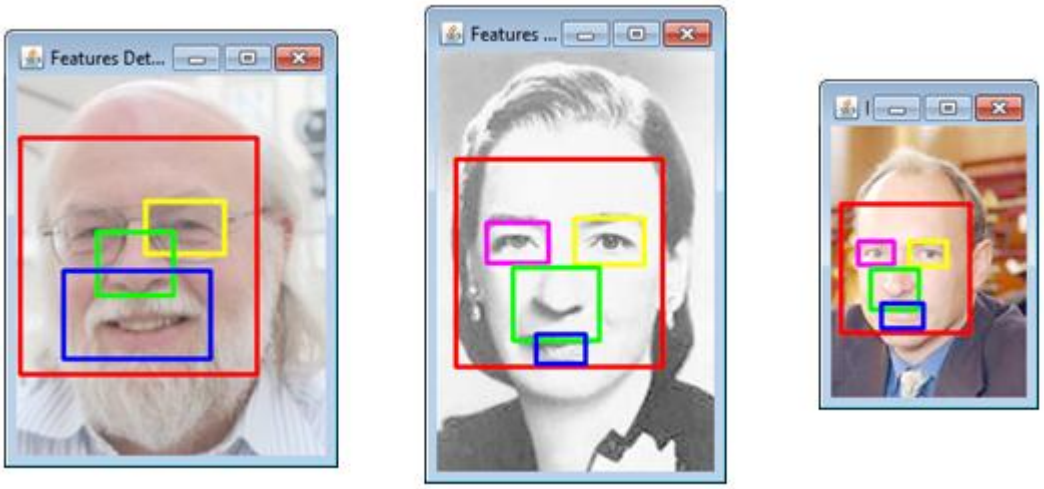


Figure 1. Facial Feature Detection with Haar Cascades.

I'm using the pre-calculated Haar classifiers from the OpenCV download to find the frontal face, left eye, right eye, nose and mouth. I utilize the bounded box details returned by those classifiers to draw colored rectangles on top of the input image.

One drawback is the lack of detail returned by a classifier. For example, it would be useful to have more information about the outline shapes of the nose and mouth. Also, the classifiers aren't that accurate (i.e. overly large rectangles, and the occasionally missed feature, such as James Gosling's left eye). However, I'll explain how detection rates can be improved by using region-of-interest (ROI) cropping on the image.

A fun alternative is the FaceSDK API from Luxand (http://www.luxand.com/facesdk/), which offers a range of capabilities including face detection and recognition, and eye tracking. My interest is in its support for facial features, which are quite a bit superior to what I can code up with Haar cascades. As an example, Figure 2 shows an application using the FaceSDK.

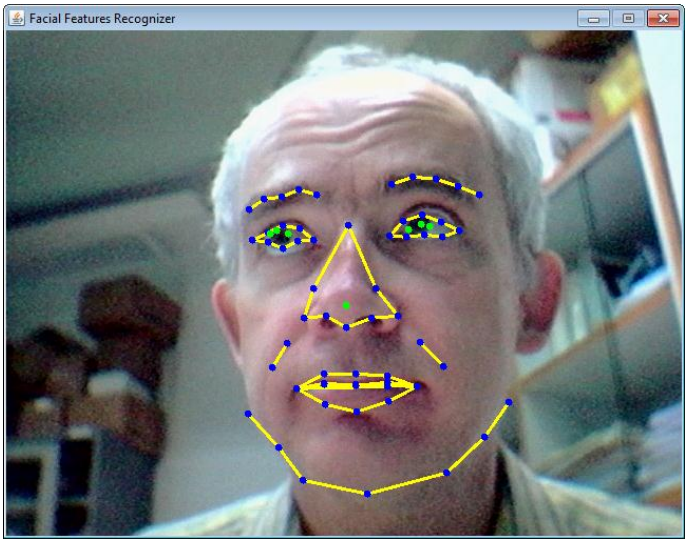


Figure 2. Facial Feature Tracking using FaceSDK.

Webcam snaps are being processed, which explains the image quality difference from Figure 1. FaceSDK represents an analyzed face by 66 coordinates for the eyes, eyebrows, lips, nose, cheek lines, and chin. I've used those points to draw yellow lines and blue and green dots on top of the webcam picture.

The major drawback of FaceSDK is that it isn't free, although you can obtain a trial version that remains fully functional for 6 weeks. Another current weakness is that there isn't a port for Android.

I've used the facial coordinates in two simple examples, shown in Figure 3.

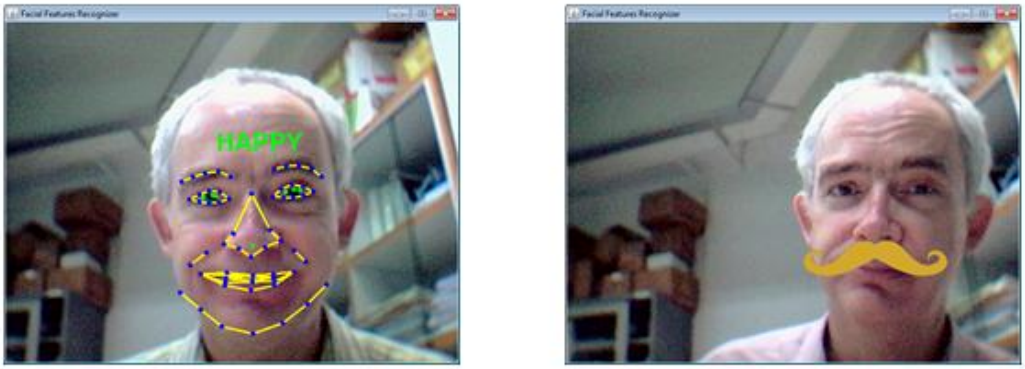


Figure 3. Happy/Sad Detector, and Mustache Augmenter.

The happy/sad detector on the left of Figure 3 utilizes a less-than-sophisticated mouse/nose width ratio calculation to determine if the user is smiling or not, and writes the word "HAPPY" or "SAD" onto the user's brow. The mustache augmenter draws a mustache pinned midway between the user's upper lip and nose.

1. Using Haar Cascades

OpenCV comes with numerous pre-computed Haar cascade classifiers, located in the subdirectory opencv/data/haarcascades/. My version of OpenCV includes classifiers for the face from the front and in profile, both eyes together, eye glasses, the left and right eye separately, nose, mouth, ears, the upper body, lower body, and a standing figure.

Of course, it's possible for a dedicated programmer to implement their own classifiers, as explained in the OpenCV documentation (http://docs.opencv.org/doc/user_guide/ug_traincascade.html), the *Learning OpenCV* textbook (Chapter 13, p.513), and in blog posts by Naotoshi Seo (http://note.sonots.com/SciSoftware/haartraining.html) and in Achu's TechBlog (http://achuwilson.wordpress.com/2011/07/01/create-your-own-haar-classifier-for-detecting-objects-in-opencv/).

My FeaturesDetector application loads an image, converts it to a grayscale, and uses a frontal face Haar classifier to obtain a bounded box for the face.

The odds of a classifier succeeding are increased by focusing the search on a likely area (a region-of-interest or ROI). For that reason, I have the eyes, nose, and mouth classifiers examine the face sub-image rather than the larger original picture. Further ROIs are imposed as facial elements are detected; for instance I employ the detected region for the nose to restrict the search for the mouth by assuming that it's located somewhere below the mid-point of the nose's bounded box.

FeaturesDetector's main() function:

```
// globals
private static final String FACE =
                     "haarcascade_frontalface_alt2.xml";

private static CvMemStorage storage;
private static IplImage drawImg;
      // input color image that has the feature boxes drawn on it


public static void main(String[] args)
{
  if (args.length != 1) {
    System.out.println("Usage: run FeaturesDetector <fnm>");
    System.exit(0);
  }

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

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

  IplImage grayImage = loadGrayImage(args[0]);

  // find the face first
  CvRect faceRect = detectFeature(grayImage, "face", FACE,
                                    0, 0, null, CvScalar.RED);
  if (faceRect == null)
    System.out.println("No face detected");
  else
```



```
    detectFacialFeatures(grayImage, faceRect);

  // draw the input image and any feature boxes
  CanvasFrame canvas = new CanvasFrame("Features Detector");
  canvas.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  canvas.showImage(drawImg);
}  // end of main()
```

The CanvasFrame object created at the end of main() displays the originally loaded image (in color), and any detected features as colored rectangles (e.g. as in Figure 1).

loadGrayImage() loads the image, converting it to a equalized grayscale that can be processed by the Haar detection method.

```
// globals
private static final String FACE_DIR = "faces\\";  // images dir
private static IplImage drawImg;


private static IplImage loadGrayImage(String fnm)
{
  IplImage img = cvLoadImage(FACE_DIR+fnm);
  if (img == null) {
    System.out.println("Could not load " + FACE_DIR+fnm);
    System.exit(1);
  }
  else
    System.out.println("Loaded " + FACE_DIR+fnm);

  drawImg = img.clone();    // store input color image

  // convert to grayscale
  IplImage grayImage = IplImage.create(img.width(), img.height(),
                                       IPL_DEPTH_8U, 1);
  cvCvtColor(img, grayImage, CV_BGR2GRAY);

  cvEqualizeHist(grayImage, grayImage);  // equalize the grayscale
  return grayImage;
}  // end of loadGrayImage()
```

Detecting a Feature

The heart of this example is detectFeature(); it loads a Haar classifier from the named file, and applies it to an image restricted to the ROI area specified by a rectangle.

The method returns a matching feature rectangle *and* draws the rectangle in a specified color onto the original input image stored in drawImg. This drawing step is a little tricky to understand because it requires the (x, y) coordinates of the feature rectangle to be mapped into the coordinate space of the input image

Fortunately, the very first call to detectFeature() in main() doesn't use a selection rectangle, so we can ignore the complexities of coordinate conversion at this stage (but I'll return to them later). The detectFeatures() code:

```
// globals
private static final String HAAR_DIR =
```



```
                "C:\\opencv\\data\\haarcascades\\";

private static CvMemStorage storage;
private static IplImage drawImg;    // original color input image


private static CvRect detectFeature(IplImage im, String featureName,
                String haarFnm, int xF, int yF,
                CvRect selectRect, CvScalar color)
{
  // load the classifier for feature detection
  CvHaarClassifierCascade classifier =
       new CvHaarClassifierCascade(cvLoad(HAAR_DIR + haarFnm));
  if (classifier.isNull()) {
    System.out.println("Could not load the classifier: " +
                              haarFnm + " for " + featureName);
    return null;
  }

  // use selection rectangle to apply a ROI to the im image
  int xSelect = 0;   // default values
  int ySelect = 0;
  if (selectRect != null) {
    cvSetImageROI(im, selectRect);
    xSelect = selectRect.x();
    ySelect = selectRect.y();
  }

  CvSeq featureSeq = cvHaarDetectObjects(im, 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 feature subimage */
  cvClearMemStorage(storage);
  cvResetImageROI(im);

  int total = featureSeq.total();
  if (total == 0) {
    System.out.println("  No " + featureName + " found");
    return null;
  }

  if (total > 1)   // should not happen, but include for safety
    System.out.println("Multiple features detected (" + total +
                    ") for " + featureName + "; using the first");

  CvRect fRect = new CvRect(cvGetSeqElem(featureSeq, 0));
    /* this feature rectangle is defined relative to
       the coordinates of the select rectangle */

  // convert fRect to user's input image coords
  int xDraw = xF + xSelect + fRect.x();
  int yDraw = yF + ySelect + fRect.y();

  // draw feature rectangle on original input image
  if (fRect != null)
    cvRectangle(drawImg, cvPoint(xDraw, yDraw),
         cvPoint(xDraw + fRect.width(), yDraw + fRect.height()),
                  color, 2, CV_AA, 0);
  return fRect;
```



```
}  // end of detectFeature()
```

The call to detectFeature() in main() is:

```
CvRect faceRect = detectFeature(grayImage, "face", FACE,
                                0, 0, null, CvScalar.RED);
```

This sets that the selection rectangle inside detectFeatures() (selectRect) to null, and the xF and yF coordinates to 0. This essentially means that no coordinate conversion needs to be carried out.

detectFeature() is passed the original input image (as a grayscale), and uses a frontal face Haar classifier to find a face. The face's bounding box is stored in the fRect rectangle.

The most complicated lines are probably:

```
int xDraw = xF + xSelect + fRect.x();
int yDraw = yF + ySelect + fRect.y();
```

which convert the feature rectangle coordinates (fRect.x(), fRect.y()) into coordinates relative to the original input image (xDraw, yDraw). In this case though, xF, yF, xSelect, and ySelect are all 0, and so I'll delay explaining the conversion until the next section.

At this point in the execution, the user's original input image (stored in drawImg) will include a red rectangle around the detected face, as in Figure 4.

Figure 4. User's Input Image with a Detected Face.

1.2. Detecting Facial Features

Back in main(), a successful call to detectFeature() will return a feature rectangle for the face, and the facial features inside the face can now be located by calling detectFacialFeatures().

detectFacialFeatures() makes four calls to detectFeature() to find the left eye, right eye, nose, and mouth. All the searches are done using the face image (i.e. the image inside the red rectangle) rather than the original input image, and also utilize selection rectangles to set a ROI for each search.

```
// globals
```



```
private static final String LEFT_EYE =
                          "haarcascade_mcs_lefteye.xml";
private static final String RIGHT_EYE =
                          "haarcascade_mcs_righteye.xml";
private static final String NOSE = "haarcascade_mcs_nose.xml";
private static final String MOUTH = "haarcascade_mcs_mouth.xml";


private static void detectFacialFeatures(IplImage grayImage,
                                                 CvRect faceRect)
{
  int xF = faceRect.x(); // coords relative to grayscale image
  int yF = faceRect.y();
  int wF = faceRect.width();
  int hF = faceRect.height();

  /* extract face image from grayscale image;
     use this for feature detections  */
  IplImage faceImage = IplImage.create(faceRect.width(),
                            faceRect.height(), IPL_DEPTH_8U, 1);
  cvSetImageROI(grayImage, faceRect);
  cvCopy(grayImage, faceImage);

  // detect left and right eyes inside face image
  CvRect selectRect = new CvRect(0,0, wF/2, hF);  //left half of face
      /* selection (x,y) coords are relative to the face image */
  CvRect leRect = detectFeature(faceImage, "left eye", LEFT_EYE,
                           xF, yF, selectRect, CvScalar.MAGENTA);

  selectRect = new CvRect(wF/2, 0, wF/2, hF); // right half of face
  CvRect reRect = detectFeature(faceImage, "right eye", RIGHT_EYE,
                           xF, yF, selectRect, CvScalar.YELLOW);

  // nose detection (below top of eyes)
  int yEye = hF/3;  // guess eye top coord
  if ((leRect != null) && (reRect != null))
    yEye = (leRect.y() < reRect.y()) ? leRect.y() : reRect.y();
                              // choose higher eye
  selectRect = new CvRect(0, yEye, wF, hF-yEye);
  CvRect noseRect = detectFeature(faceImage, "nose", NOSE,
                             xF, yF, selectRect, CvScalar.GREEN);

  // mouth detection  (below nose middle)
  int noseMid = hF/2;  // guess middle of nose coord
  if (noseRect != null)
    noseMid = yEye + noseRect.y() + noseRect.height()/2;

  selectRect = new CvRect(0, noseMid, wF, hF-noseMid);
  detectFeature(faceImage, "mouth", MOUTH,
                             xF, yF, selectRect, CvScalar.BLUE);
}  // end of detectFacialFeatures()
```

Each call to defectFeature() in detectFacialFeatures() uses several coordinates, and conversions between different coordinate systems. I'll explain matters with the aid of Figure 5, which focuses on the search for the right eye

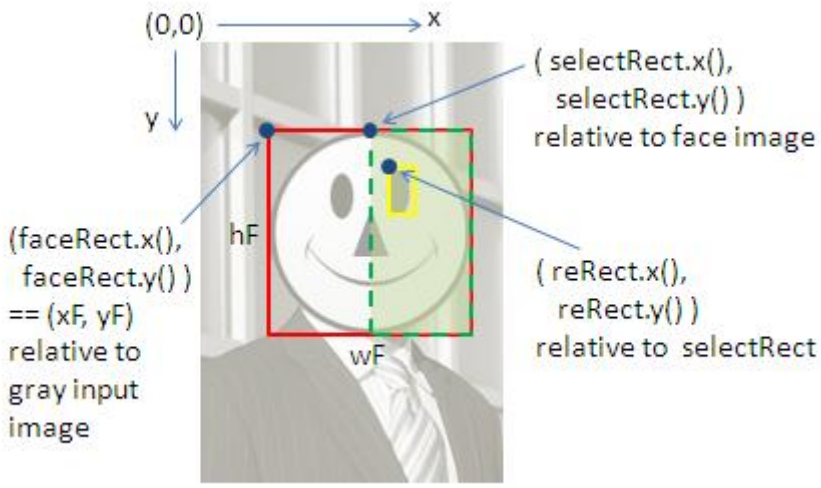


Figure 5. Coordinates involved in finding the Right Eye.

The search for the right eye uses the following lines of code from detectFacialFeatures():

```
int xF = faceRect.x();  // coords relative to grayscale image
int yF = faceRect.y();
int wF = faceRect.width();
int hF = faceRect.height();
     :
selectRect = new CvRect(wF/2, 0, wF/2, hF);    // right half of face
CvRect reRect = detectFeature(faceImage, "right eye", RIGHT_EYE,
                        xF, yF, selectRect, CvScalar.YELLOW);
```

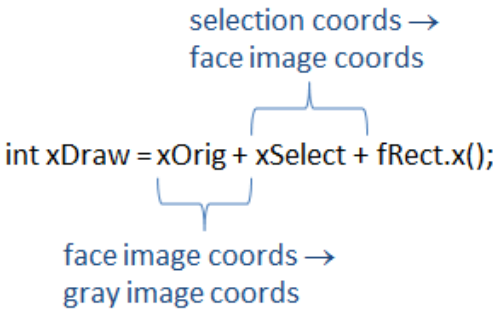
The selection region (selectRect) is restricted to the right half of the face image. Note that the coordinates used by selectRect are defined *relative to the face image*.

detectFeature() returns a feature rectangle (reRect) for the eye; its coordinates are defined *relative to the selection rectangle*!

The hardest part of detectFeature() to understand is the conversion of the reRect coordinate (which is called fRect inside the method) into a coordinate relative to the grayscale image. The relevant lines in detectFeature() are:

```
int xDraw = xF + xSelect + fRect.x();
int yDraw = yF + ySelect + fRect.y();
```

The conversions for the x- and y- values are done in two steps: from selection coordinates to face image coordinates, and from face image coordinates to grayscale input image coordinates. Consider just the x-value conversion:



detectFacialFeatures() is further complicated because each call to detectFeature() uses a different selection region:

- left eye: left half of the face image
- right eye: right half of the face image
- nose: area below the higher of the two eyes
- mouth: area below the mid point of the nose

Unfortunately, this coding approach is necessary otherwise the Haar classifiers would quite likely identify the wrong regions. For example, the downward curve of a lens in a pair of eyeglasses can be mistaken for a mouth.

All these headaches with coordinates conversion disappear when I move over to the FaceSDK.

2. The FaceSDK API

Luxand FaceSDK (http://www.luxand.com/facesdk/) facial detection returns an array of 66 coordinates representing the eyes, eyebrows, nose, cheeks, mouth, and chin. The API is supported across Windows, Mac OS X, and Linux, and includes a DirectShow-compatible webcam library, but I'll keep on using JavaCV in my examples.

Feature detection can utilize multiple processors to increase its speed, and can handle a small amount of head rotation side-to-side (–10..10 degrees), and more around the z-axis (–30..30 degrees). Recognition seems a bit unreliable when the user is wearing glasses, which is why I've removed my spectacles in the tests shown here (e.g. in Figures 2 and 3).

It's necessary to obtain a free license key from the Luxand website, which is used to validate the API when it's installed, and to activate code using the API at runtime. The key expires after 6 weeks, but you can download new ones.

The 66 feature points are numbered from 0 to 65, and also have names. The distribution of feature numbers over the face is shown in Figure 6, which comes from the FaceSDK documentation. The manual also lists the feature names.

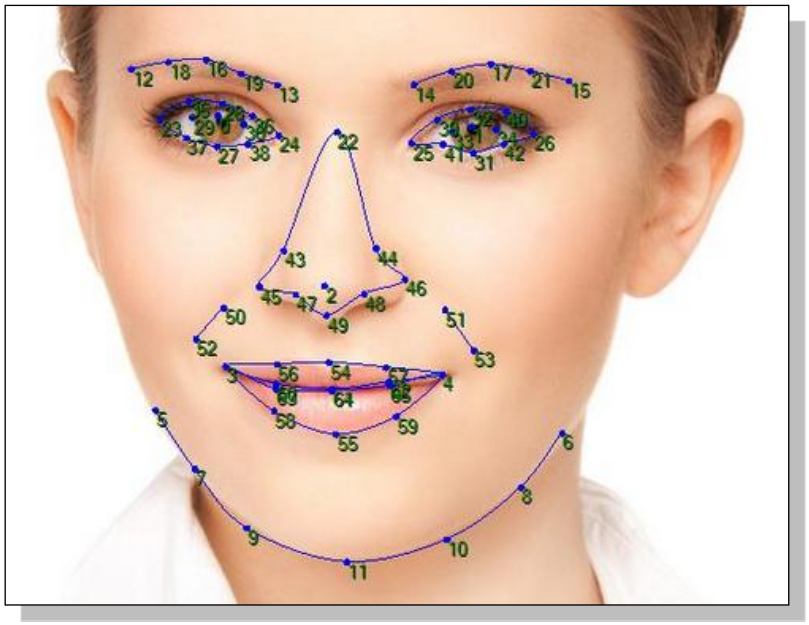


Figure 6. Numbered Feature Points on a Face.

Typical names and values are:

FSDKP_LEFT_EYE	0
FSDKP_RIGHT_EYE	1
FSDKP_LEFT_EYE_OUTER_CORNER	23
FSDKP_LEFT_EYE_INNER_CORNER	24

Feature points with the indices 0, 1, and 2 are significant because they represent the centers of the pupils, and the tip of the nose.

The following code fragment shows how to extract feature points from an image. There are three main steps: 1) converting the image into a FaceSDK HImage object, 2) finding the face region inside the image, and 3) extracting the facial feature coordinates from that region.

```
// 1. convert image to FaceSDK format
HImage imHandle = new HImage();
if (FSDK.LoadImageFromAWTImage(imHandle, image,
          FSDK_IMAGEMODE.FSDK_IMAGE_COLOR_32BIT) != FSDK.FSDKE_OK) {
  System.out.println("Failed to create FaceSDK image");
  return;
}

// 2. find face region
FSDK.TFacePosition.ByReference facePos =
            new FSDK.TFacePosition.ByReference();
if (FSDK.DetectFace(imHandle, facePos) != FSDK.FSDKE_OK) {
  System.out.println("Failed to find a face");
  return;
}

// 3. extract facial features from face region
FSDK_Features.ByReference facialFeatures =
```

```
                                new FSDK_Features.ByReference();
FSDK.DetectFacialFeaturesInRegion(imHandle,
                  (FSDK.TFacePosition)facePos, facialFeatures);

// print out the feature coordinates
for (int i=0; i < FSDK.FSDK_FACIAL_FEATURE_COUNT; i++)
  System.out.println(" Point " + i + " = (" +
                       facialFeatures.features[i].x; + ", " +
                       facialFeatures.features[i].y + ")");

FSDK.FreeImage(imHandle);
```

One surprising omission from the API are methods for easily rendering the feature points onto an image in a style similar to Figure 6. I've implemented this functionality in my FacialRecog application, as shown in Figure 2, and described in the next section.

3. Facial Recognition Using the FaceSDK

My FacialRecog application utilizes the usually coding structure for accessing webcam snaps and rendering them onto a panel. The class diagrams are shown in Figure 7.

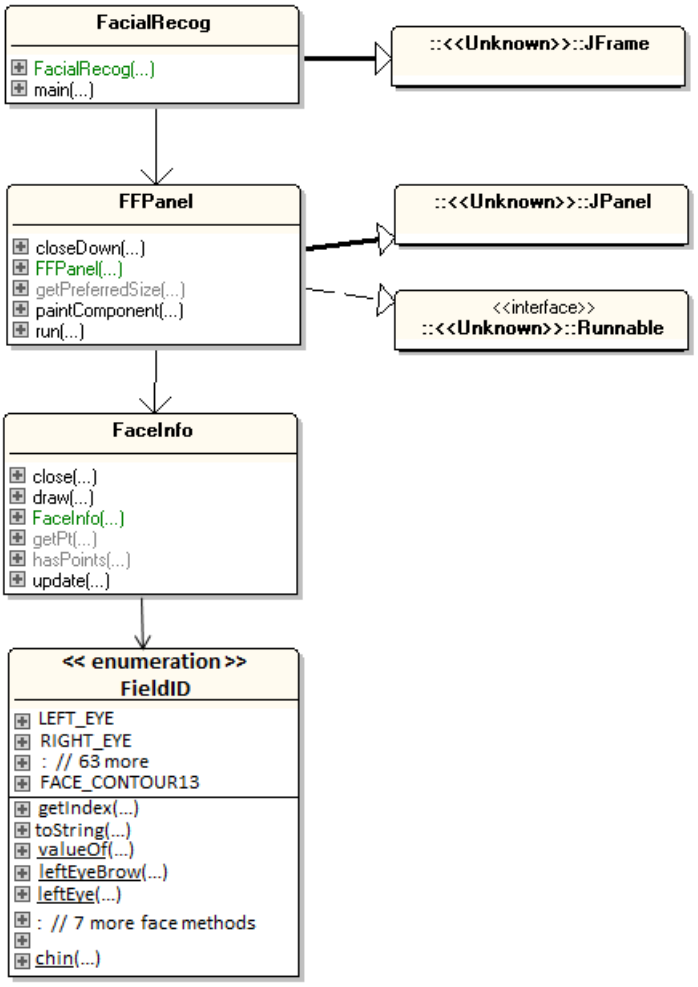


Figure 7. Class Diagrams for the FacialRecog Application.

The core of FFPanel is a threaded loop which uses JavaCV's FrameGrabber to grab a webcam snap, then calculates and draws facial information using the FaceInfo class.

FieldID is an enum version of the FaceSDK face constant names and their index values. My main reason for adding this class was to include a series of static methods for returning arrays of IDs representing all the points comprising a face region. For example, FeatureID.leftEye() returns an array of all the IDs used to define the left eye. This makes it much easier for FaceInfo to draw lines connecting all the points in a region. A minor change in FieldID was to simplify the naming scheme for the feature names, removing the "FSDKP_" prefix. For instance, the FSDKP_LEFT_EYE constant in FaceSDK is called LEFT_EYE in FieldID.

3.1. The Processing Loop in FFPanel

FFPanel's processing loop is located in its run() method, and has the usual structure, except for how termination is handled.

```
// globals
private static final int DELAY = 900;
                  // time (ms) between redraws of the panel

private BufferedImage snapIm = null;  // type changed from IplImage
private volatile boolean isRunning;
private FaceInfo faceInfo;


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

  faceInfo = new FaceInfo();

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

    snapIm = (picGrab(grabber, CAMERA_ID)).getBufferedImage();
      // save the image as a BufferedImage

    faceInfo.update(snapIm);   // update face features
    moodDetector();
    repaint();

    duration = System.currentTimeMillis() - startTime;
    if (duration < DELAY) {
      try {
        Thread.sleep(DELAY - duration);
      }
      catch (Exception ex) {}
    }
  }
  faceInfo.close();
  closeGrabber(grabber, CAMERA_ID);
```

```
  System.exit(0);
}  // end of run()
```

There are three new lines highlighted in bold: first a FaceInfo object is created, then the object is updated inside the loop by being passed the latest webcam image, and finally the object is closed down at the end of run(). There's also a call to moodDetector(), which I'll return to after explaining the internals of FaceInfo.

Timing of the call to FaceInfo.update() indicated that it takes around 800 ms to extract facial details from an image, which is quite a bit slower than the 104 ms quoted in the FaceSDK documentation. However, that figure is for a fast Intel core i7 930 processor with 4 cores. Due to the slowness of my test machine, I set the sleep time (the DELAY value) inside the processing loop to 900 ms.

Terminating the Application

Unlike previous examples, there's a call to System.exit(0) at the end of run(). This replaces the call to System.exit(0) which usually occurs in the windowClosing() method in the top-level JFrame class. The corresponding code in FacialRecog() only calls FacePanel.closeDown() without an exit() call :

```
// window closing listener in FacialRecog
addWindowListener( new WindowAdapter() {
  public void windowClosing(WindowEvent e)
  { facePanel.closeDown(); }   // no exit() call
});
```

The closeDown() method in FacePanel is also changed; it waits an arbitrary large amount of time (5 seconds) to ensure that the processing loop terminates.

```
// in FacePanel
private volatile boolean isRunning;

public void closeDown()
{ isRunning = false;
  try {   // wait for processing loop to finish
    Thread.sleep(5000);
  }
  catch (Exception ex) {}
}
```

This coding change is necessary to prevent the FaceSDK library from crashing at termination time. This approach means that the library isn't closed down except at the end of the processing loop when all facial processing has stopped.

Panel Rendering

The panel's rendering is managed by FFPanel's paintComponent() method:

```
// globals
private BufferedImage snapIm = null; // current webcam snap
private FaceInfo faceInfo;
```



```
private Font msgFont;


public void paintComponent(Graphics g)
// Draw the webcam image, and face features
{
  super.paintComponent(g);
  Graphics2D g2 = (Graphics2D) g;
  g2.setFont(msgFont);

  if (snapIm == null)
    g2.drawString("Initializing webcam, please wait...",
                                              20, HEIGHT/2);
  else {
    g2.drawImage(snapIm, 0, 0, this);
    faceInfo.draw(g2);  // draw facial features
    // reportMood(g2);
    // attachStache(g2);
  }
} // end of paintComponent()
```

Figure 2 shows an example of the how the panel is rendered, and the hard work of drawing lines and points around the facial regions is handled by the call to FaceInfo.draw() which I'll explain in the next section. The commented out calls to reportMood() and attachStache() deal with drawing the 'mood' words and mustache seen in Figure 3. I'll describe their code after I've finished with FaceInfo.

3.2. Collecting Face Information

The creation of the FaceInfo object requires that the FaceSDK library be 'activated' at runtime with a license key. A key can be obtained for free from the Luxand website (http://www.luxand.com/facesdk/).

```
// globals
private static final String FACE_SDK_LICENSE = "XXX...XXX";
    // obtained from http://luxand.com/facesdk/

private Point[] featurePts;    // holds the 66 feature coordinates
private boolean hasPoints = false;


public FaceInfo()
{
  initFaceSDK();

  FSDK.SetFaceDetectionParameters(false, true, 384);
    // DetermineFaceRotationAngle == true for feature detection

  featurePts = new Point[FSDK.FSDK_FACIAL_FEATURE_COUNT];
  for (int i=0; i < FSDK.FSDK_FACIAL_FEATURE_COUNT; i++)
    featurePts[i] = new Point(0, 0);    // dummy values
}  // end of FaceInfo()


private void initFaceSDK()
{
  try {
```

```
    if (FSDK.ActivateLibrary(FACE_SDK_LICENSE) != FSDK.FSDKE_OK){
      System.out.println("License error while activating FaceSDK");
      System.exit(1);
    }
  }
  catch(UnsatisfiedLinkError e) {   // is DLL is in local dir?
    System.out.println("Could not link to FaceSDK library");
    System.exit(1);
  }

  if (FSDK.Initialize() != FSDK.FSDKE_OK) {
    System.out.println("Could not initialize FaceSDK");
    System.exit(1);
  }
}  // end of initFaceSDK()
```

For facial feature recognition to work, it's necessary to turn on face rotation detection via the call to FSDK.SetFaceDetectionParameters() in the constructor.

The main data structure inside FaceInfo is the featurePts[] array, which will hold the 66 feature coordinates calculated by FaceSDK. The coordinates are given dummy values in the constructor, then updated when FaceInfo.update() is called:

```
public synchronized void update(BufferedImage im)
{
  // convert image to FaceSDK format
  HImage imHandle = new HImage();
  if (FSDK.LoadImageFromAWTImage(imHandle, im,
        FSDK_IMAGEMODE.FSDK_IMAGE_COLOR_32BIT) != FSDK.FSDKE_OK) {
    System.out.println("Failed to create FaceSDK image");
    return;
  }
  // printImageSize(imHandle);

  // find face rectangle
  FSDK.TFacePosition.ByReference facePos =
                  new FSDK.TFacePosition.ByReference();
  if (FSDK.DetectFace(imHandle, facePos) != FSDK.FSDKE_OK) {
    System.out.println("Failed to find a face");
   return;
  }
  // printFaceSize(facePos);

  // extract facial features from face region
  FSDK_Features.ByReference facialFeatures =
                      new FSDK_Features.ByReference();
  FSDK.DetectFacialFeaturesInRegion(imHandle,
              (FSDK.TFacePosition)facePos, facialFeatures);

  // store coordinates in featurePts[]
  for (int i=0; i < FSDK.FSDK_FACIAL_FEATURE_COUNT; i++) {
    featurePts[i].x = facialFeatures.features[i].x;
    featurePts[i].y = facialFeatures.features[i].y;
  }
  hasPoints = true;

  FSDK.FreeImage(imHandle);
}  // end of update()
```


update() is almost identical to the FaceSDK code fragment that I showed earlier. The main difference is that instead of printing the detected coordinates, they're stored inside the featurePts[] array. After the array has been filled, the global boolean hasPoints is set to true.

update() is synchronized so that its changes to the featurePts[] array can not be mixed up with the drawing of those points which occurs in the synchronized draw() method.

I used the two commented-out print methods in update() when I was debugging the code. They print details about the FaceSDK image and the detected face:

```
private void printImageSize(HImage imHandle)
// print the size of the FaceSDK version of the image
{
  int imWidthRef[] = new int[1];
  int imHeightRef[] = new int[1];
  FSDK.GetImageWidth(imHandle, imWidthRef);
  FSDK.GetImageHeight(imHandle, imHeightRef);
  System.out.println("  (width, height): (" +
             imWidthRef[0] + ", " + imHeightRef[0] + ")");
}  // end of printImageSize()


private void printFaceSize(FSDK.TFacePosition.ByReference facePos)
// print the face position and angle
{
  int left = facePos.xc - facePos.w/2;
  int top = facePos.yc - facePos.w/2;    // no h (?)
  System.out.println("   face (left, top): (" + left + ", " +
                       top + "); angle: " + facePos.angle);
}  // end of printFaceSize()
```

3.3. Accessing a Point

The FaceInfo.getPt() method returns a specified point when passed a feature ID:

```
public Point getPt(FeatureID id)
{ if (!hasPoints)
    return null;
  return new Point( featurePts[id.getIndex()]);
}
```

For example, the following call returns the coordinate for the bridge of the user's nose:

```
Point noseBridge = faceInfo.getPt(FeatureID.NOSE_BRIDGE);
```

FeatureID.NOSE_BRIDGE is one of 66 names defined in the FeatureID enumeration.

Inside getPt(), FeatureID.getIndex() looks up the index value associated with the ID. Inside FeatureID, each ID has an associated value, as the class fragment shows:

```
public enum FeatureID
{
  LEFT_EYE(0),     // left pupil
  RIGHT_EYE(1),    // right pupil
```

```
      :  // more IDs and values, not shown
  NOSE_BRIDGE(22),
      :  // more IDs and values, not shown

  private int index;

  private FeatureID(int i)
  {  index = i; }

  public int getIndex()
  {  return index;  }

      :  // more methods, not shown
}  // end of FeatureID enum
```

I copied the IDs and index values from the FaceSDK documentation, although I simplified the IDs by leaving off their "FDSK_" prefix.

3.4. Drawing Face Regions and Points

Figure 8 shows a close-up of the lines and points drawn over the user's face by FaceInfo.draw().

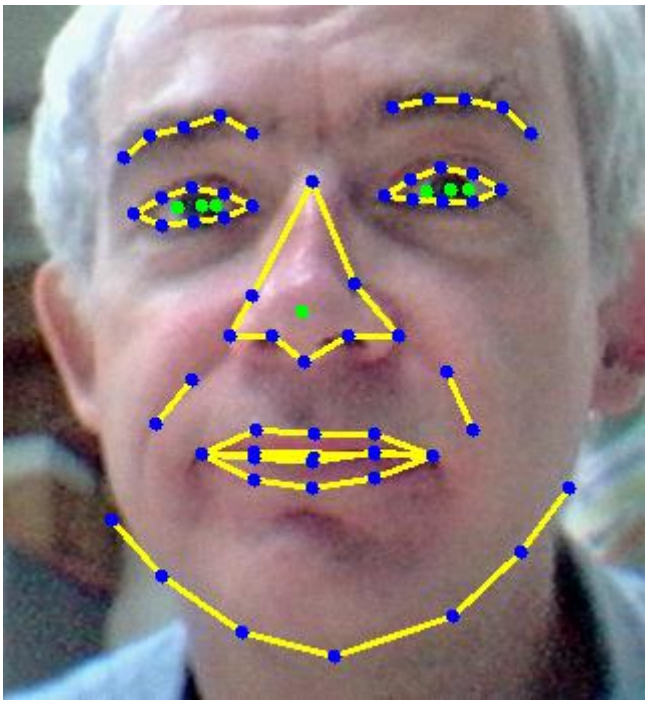


Figure 8. Lines and Points Drawn on a Face.

There are two kinds of colored dots: the points linked by lines are blue, while single coordinates are green. The connected points define ten face regions: the left and right eyebrows, left and right eyes, the nose, left and right cheek lines, upper and lower lips, and the outline of the chin. There are seven green points – three for each eye, and one for the tip of the nose. The three eye coordinates are for the left and right edges of the pupil, and its center.

To simplify the drawing task, the FeatureID enum contains ten static methods that return arrays of face region IDs: leftEyeBrow(), leftEye(), rightEyeBrow(), rightEye(), nose(), leftCheek(), rightCheek(), topLip(), bottomLip(), and chin(). For example, FeatureID.leftEyeBrow() is defined as:

```
// in the FeatureID enum
public static FeatureID[] leftEyeBrow()
{  return new FeatureID[] {
      LEFT_EYEBROW_INNER_CORNER,
      LEFT_EYEBROW_MIDDLE_RIGHT,
      LEFT_EYEBROW_MIDDLE,
      LEFT_EYEBROW_MIDDLE_LEFT,
      LEFT_EYEBROW_OUTER_CORNER };
}
```

These static methods make it considerably easier to implement a concise version of FaceInfo.draw():

```
// in the FaceInfo class
public synchronized void draw(Graphics2D g2)
// draw the face features using lines and dots
{
  if (!hasPoints)
    return;

  // draw lines & blue points for the 10 face regions
  connectPoints(g2, FeatureID.leftEyeBrow(), false);
  connectPoints(g2, FeatureID.leftEye(), true);

  connectPoints(g2, FeatureID.rightEyeBrow(), false);
  connectPoints(g2, FeatureID.rightEye(), true);

  connectPoints(g2, FeatureID.nose(), true);

  connectPoints(g2, FeatureID.leftCheek(), false);
  connectPoints(g2, FeatureID.rightCheek(), false);

  connectPoints(g2, FeatureID.topLip(), true);
  connectPoints(g2, FeatureID.bottomLip(), true);

  connectPoints(g2, FeatureID.chin(), false);

  // draw  7 green dots for pupils, irises, nose tip
  g2.setColor(Color.GREEN);

  drawPoint(g2, FeatureID.LEFT_EYE);
  drawPoint(g2, FeatureID.LEFT_EYE_LEFT_IRIS_CORNER);
  drawPoint(g2, FeatureID.LEFT_EYE_RIGHT_IRIS_CORNER);

  drawPoint(g2, FeatureID.RIGHT_EYE);
  drawPoint(g2, FeatureID.RIGHT_EYE_LEFT_IRIS_CORNER);
  drawPoint(g2, FeatureID.RIGHT_EYE_RIGHT_IRIS_CORNER);

  drawPoint(g2, FeatureID.NOSE_TIP);
}  // end of draw()
```


The relevant array of feature IDs is passed to connectPoints() for each face region. For instance, the call to draw the left eyebrow's lines and points is:

```
connectPoints(g2, FeatureID.leftEyeBrow(), false);
```

connectPoints()'s boolean argument indicates whether the points should be drawn as a closed polygon or not; the left eyebrow lines do not form a polygon, so the argument is false.

The relevant ID is passed to drawPoint() for each of the single points. For example, the nose tip dot is drawn by calling:

```
drawPoint(g2, FeatureID.NOSE_TIP);
```

connectPoints() draws a series of yellow lines, linked by blue points:

```
private void connectPoints(Graphics2D g2, FeatureID[] ids,
                                          boolean isPolygon)
{ // draw the yellow lines
  g2.setColor(Color.YELLOW);
  g2.setStroke(new BasicStroke(3));
  for(int i=0; i < ids.length-1; i++)
    drawLine(g2, ids[i], ids[i+1]);

  if (isPolygon)    // draw a line back to the first point
    drawLine(g2, ids[ids.length-1], ids[0]);

  // draw the blue dots
  g2.setColor(Color.BLUE);
  for(int i=0; i < ids.length; i++)
    drawPoint(g2, ids[i]);
}  // end of connectPoints()
```

Implicit in the coding of connectPoints() is the assumption that the IDs are listed in a suitable drawing order inside the supplied IDs array.

drawLine() and drawPoint() are passed feature IDs, so must look up the associated index values and access the global featurePts[] array before calling Graphics methods.

```
// globals
private Point[] featurePts;

private void drawLine(Graphics2D g2, FeatureID fromID,
                                        FeatureID toID)
// draw a line between two feature IDs
{ Point fromPt = featurePts[fromID.getIndex()];
  Point toPt = featurePts[toID.getIndex()];
  g2.drawLine(fromPt.x, fromPt.y, toPt.x, toPt.y);
}

private void drawPoint(Graphics2D g2, FeatureID id)
// draw a point for the feature ID
{ Point pt = featurePts[id.getIndex()];
  g2.drawOval(pt.x-2, pt.y-2, 4, 4);
}
```


3.5. Detecting the User's Mood

Figure 2 shows the optional mood detection feature of my FaceRecog application. Figure 9 shows the detection of 'happiness' and 'sadness', but this time I've commented out the FaceInfo.draw() call in the code so that feature lines and points aren't displayed.

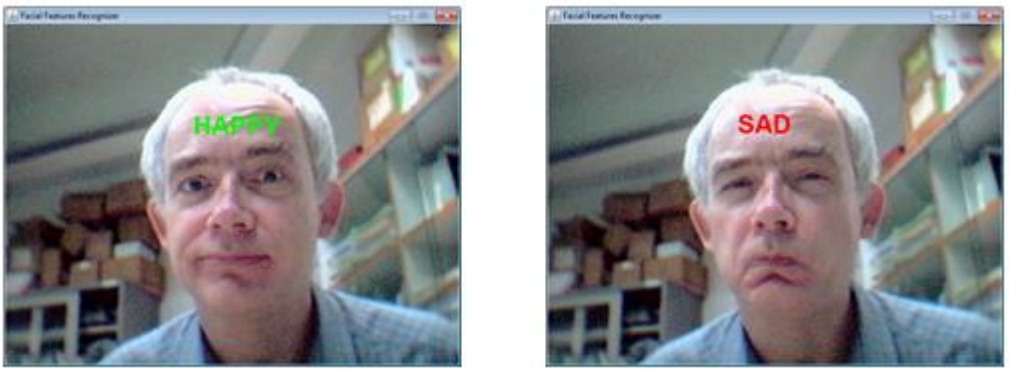


Figure 9. A Happy and Sad User.

Of course, this is **not** a serious implementation of mood analysis. It's only intended to show how feature point information can be used in a simple example. In fact, it might be more accurate to label the faces in Figure 9 as "drunk" and "constipated".

So how exactly is this miracle of mood interpretation implemented?

moodDetector() is called in FFPanel.run():

```
   :
faceInfo.update(snapIm);   // update face features
moodDetector();
  :
```

moodDetector() calculates the ratio of the mouth width to the nose width, and then decides that the user is happy if the mouth is sufficiently wider than the nose. If the mouth is narrower than the nose (i.e. scrunched up) then the user is deemed to be sad. These states are recorded by setting the global booleans isHappy and isSad,

```
// in FFPanel
// globals used for mood detection using mouth/nose 'analysis'
private static final double SMALL_MOUTH = 1.1;
private static final double WIDE_MOUTH = 1.4;

private FaceInfo faceInfo;

private boolean isSad = false;
private boolean isHappy = false;


private void moodDetector()
{
  if (!faceInfo.hasPoints())
    return;

  // calculate nose width
  Point noseLeft = faceInfo.getPt(FeatureID.NOSE_LEFT_WING_OUTER);
```



```
  Point noseRight = faceInfo.getPt(FeatureID.NOSE_RIGHT_WING_OUTER);
  double noseWidth = noseLeft.distance(noseRight);

  // calculate mouth width
  Point mouthLeft = faceInfo.getPt(FeatureID.MOUTH_LEFT_CORNER);
  Point mouthRight = faceInfo.getPt(FeatureID.MOUTH_RIGHT_CORNER);
  double mouthWidth = mouthLeft.distance(mouthRight);

  double mouthRatio = mouthWidth/noseWidth;

  // convert ratio into a mood boolean setting
  isSad = false;   // reset
  isHappy = false;
  if (mouthRatio <= SMALL_MOUTH)
    isSad = true;
  else if (mouthRatio >= WIDE_MOUTH)
    isHappy = true;
}  // end of moodDetector()
```

Note that moodDetector() starts with a call to FaceInfo.hasPoints() to determine if there are any available feature points. This is a common safeguard in any code that accesses point information.

The mood is rendered by a call to reportMood() in FFPanel.paintComponent():

```
  :
g2.drawImage(snapIm, 0, 0, this);
// faceInfo.draw(g2);
reportMood(g2);
// attachStache(g2);
  :
```

I've commented out the call to FaceInfo.draw() so the panel looks like Figure 9.

reportMood() uses the isHappy and isSad booleans to decide which string to draw. I use the nose bridge coordinate along with small offsets upwards and to the left to position the string on the user's forehead.

```
private void reportMood(Graphics2D g2)
{
  if (!faceInfo.hasPoints())
    return;

  Point noseBridge = faceInfo.getPt(FeatureID.NOSE_BRIDGE);
  if (isSad) {
    g2.setColor(Color.RED);
    g2.drawString("SAD", noseBridge.x-45, noseBridge.y-60);
  }
  else if (isHappy) {
    g2.setColor(Color.GREEN);
    g2.drawString("HAPPY", noseBridge.x-50, noseBridge.y-60);
  }
}  // end of reportMood()
```


3.6. Sporting a Moustache

Why wait weeks for a suitably luxurious mustache to grow, when you can enjoy one in seconds? Figure 3 shows the mustache in place, and it stays attached between the user's upper lip and the bottom of the nose as he (or she) moves around.

A mustache graphic is loaded at startup, and FFPanel.paintComponent() is modified so that attachStache() is called:

```
  :
g2.drawImage(snapIm, 0, 0, this);
// faceInfo.draw(g2);
// reportMood(g2);
attachStache(g2);
  :
```

The coding is so simple that I've put all the processing inside the attachStache() drawing method, whereas with the mood detector I separated the calculation and drawing elements between two methods.

The attachStache() code:

```
// globals
private FaceInfo faceInfo;
private BufferedImage mustacheIm;


private void attachStache(Graphics2D g2)
// draw moustache mid-way between nose bottom and mouth top
{
  if (!faceInfo.hasPoints())
    return;

  Point mouthTop = faceInfo.getPt(FeatureID.MOUTH_TOP);
  Point noseBottom = faceInfo.getPt(FeatureID.NOSE_BOTTOM);
  int xC = (mouthTop.x + noseBottom.x)/2;
  int yC = (mouthTop.y + noseBottom.y)/2;

  if (mustacheIm != null)
    g2.drawImage(mustacheIm, (xC - mustacheIm.getWidth()/2),
                             (yC - mustacheIm.getHeight()/2), this);
}  // end of attacheStache()
```

Note that the size of the mustache graphic never changes. It could be adjusted by calculating the width of some face element, such as the mouth, to estimate the user's distance from the screen and therefore obtain a scaling factor for the image.