Chapter 8.5. Eye Tracking

Did you ever want to control your computer's mouse cursor by simply looking at it? Guiding it from one side of the screen to the other with a flick of your eye, at the speed of thought?

To be clear, we're talking about pupil or iris tracking (see Figure 1), with movements calculated relative to the eye's border. The intention is that as the user's pupil moves to the left, right, up and down, then so will the cursor.

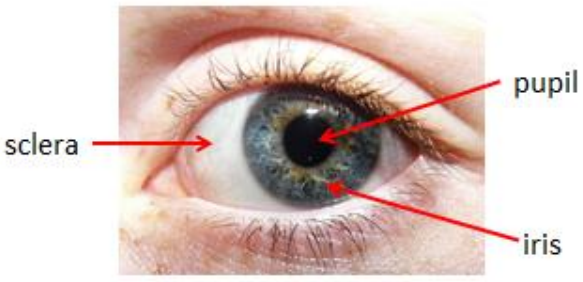

Figure 1. A Typical Eye.

Pupil tracking using a PC/laptop's webcam was the subject of the final year project of my student, Chonmaphat Roonnapak. He examined a variety of tracking techniques, chose the best, and wrote a game that was controlled by pupil/iris movements.

I'll describe a simplified version of his work in this chapter, using a Haar classifier and blob detection for the tracking, and employing the results to move a cursor inside a window. An overview of the approach is shown in Figure 2.

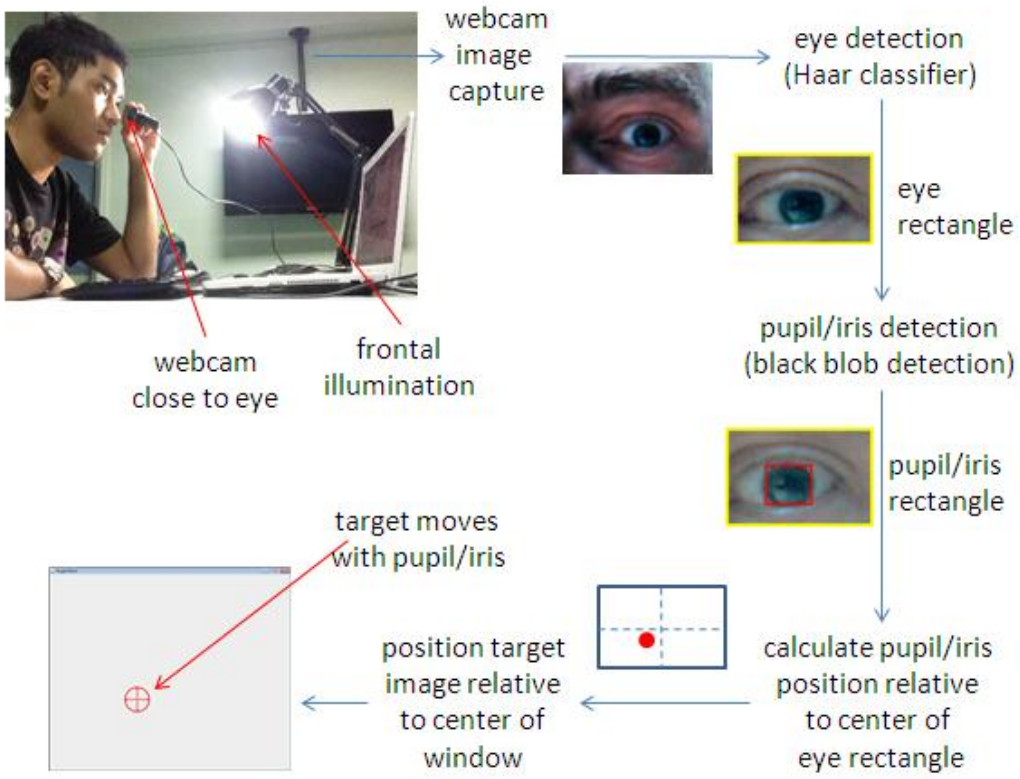

Figure 2. Eye Tracking Stages.

Chonmaphat had to give up on the idea of using his laptop's webcam as the input source, since the quality of the captured images weren't good enough. To reliably track a pupil, the camera has to be positioned close to the eye, and be provided with plentiful, constant illumination (as seen in the top-left picture in Figure 2). The ideal situation would be to attach the webcam to a helmet or cap so that it would stay mostly stationary relative to the eye as the user's head moves.

The image processing has two main parts: first the eye is found inside the webcam image using a pre-existing Haar classifier trained to detect a left eye. The second stage uses a dark colored blob detector to find the pupil, or more usually the iris.

The center of the eye rectangle is treated as the 'origin', and the offset of the center of the pupil/iris rectangle from that origin is calculated. This offset is scaled to generate coordinates relative to the center of the application window (see the bottom right of Figure 2), and the target cursor is drawn at that position.

The EyeTracker application consists of two windows, as in Figure 3.

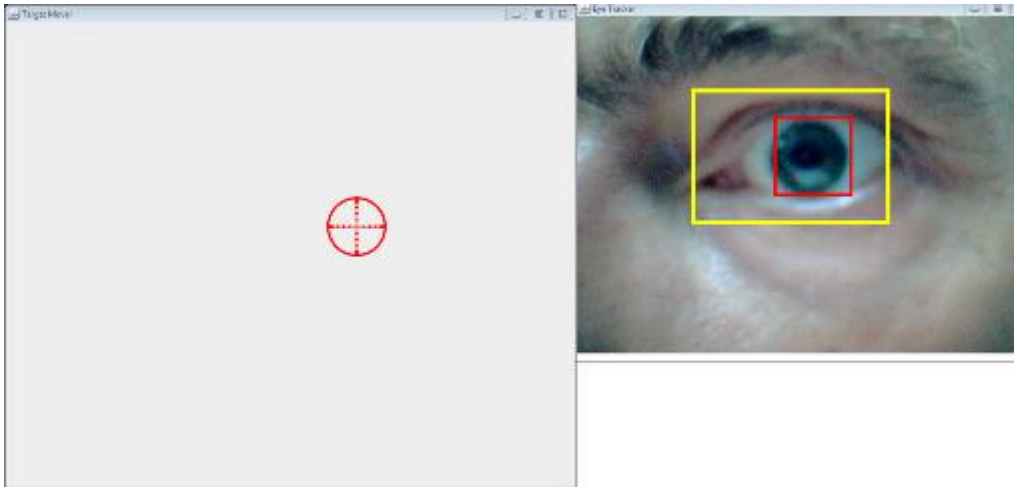

Figure 3. The EyeTracker Application.

The EyeTracker class diagrams are shown in Figure 4, with only class names listed.

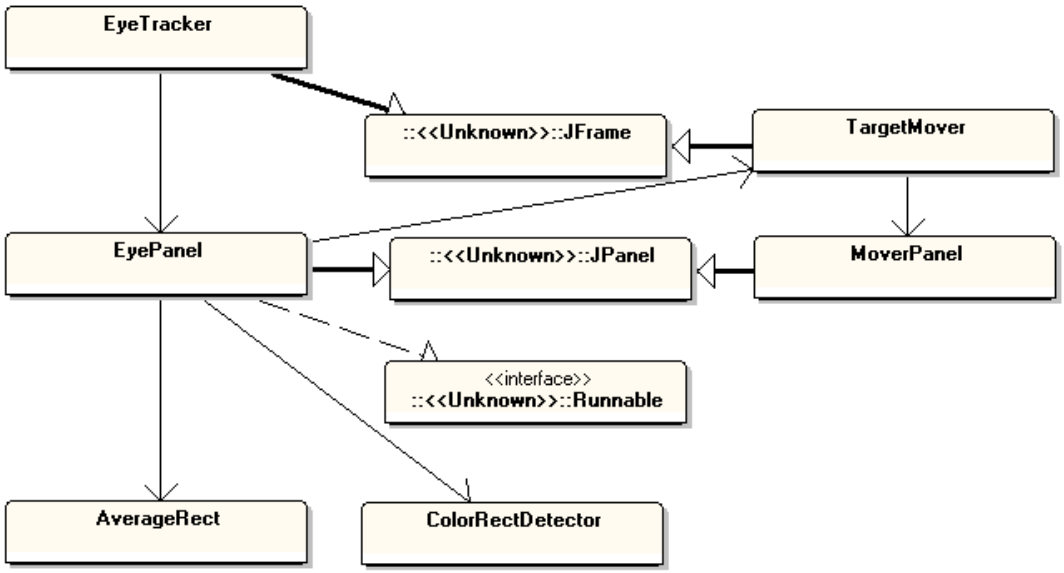

Figure 4. The EyeTracker Class Diagrams.

The window on the left of Figure 3 contains a crosshairs image representing the current cursor position, which moves in response to pupil/iris movements. It is implemented by the TargetMover JFrame and MoverPanel JPanel classes in Figure 4.

The window on the right of Figure 3 shows the current webcam image, with a yellow rectangle around the detected eye and a red rectangle around the pupil/iris. The top-level JFrame is created by the EyeTracker class, but the image processing is carried out in the EyePanel class. Most of this chapter is about the computer vision elements of EyePanel.

The eye and pupil/iris rectangles information are held in two AverageRect objects; each object stores a short sequence of rectangles obtained from the last few webcam

snaps, and returns an average of them when requested. This approach smooths away the inevitable slight shaking of the camera which affect the rectangles' coordinates. The colored blob processing is very similar to that done in Chapter 5 ("Blobs Drumming"), and is carried out by the ColorRectDetector class.

1. Eye Processing

EyePanel spawns a thread for the webcam image processing loop, which has the same general structure as previously: snap a picture, process it, display, sleep, and repeat.

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

private IplImage snapIm = null; // current webcam snap
private volatile boolean isRunning;


public void run()
/* display the current webcam image every DELAY ms
   Each image is processed to find an eye and its pupil/iris
*/
{
  FrameGrabber grabber = initGrabber(CAMERA_ID);
  if (grabber == null)
    return;

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

    snapIm = picGrab(grabber, CAMERA_ID);
    IplImage eyeIm = trackEye(snapIm);   // find the eye
    if (eyeIm != null)
      trackPupil(eyeIm);    // find the pupil/iris
    repaint();

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

The eye detection is handled by the trackEye() method, and the pupil/iris located by trackPupil().

2. Eye Tracking

The eye tracking code uses a pre-existing Haar classifier for a left eye, which comes with the OpenCV distribution. The classifier is loaded at initialization time:

```
// globals
// Haar cascade definition used for eye detection
private static final String EYE_CASCADE_FNM = "eye.xml";
  /* originally called haarcascade_frontalface_alt2.xml
     in C:\OpenCV2.2\data\haarcascades\ and at
     http://alereimondo.no-ip.org/OpenCV/34
  */

private CvMemStorage storage;
private CvHaarClassifierCascade eyeClassifier;


private void initDetector()
{
  // instantiate a classifier cascade for eye detection
  eyeClassifier = new CvHaarClassifierCascade(
                                        cvLoad(EYE_CASCADE_FNM));
  if (eyeClassifier.isNull()) {
    System.out.println("\nCould not load the classifier file: " +
                                            EYE_CASCADE_FNM);
    System.exit(1);
  }
  storage = CvMemStorage.create();
}  // end of initDetector()
```

The classifier is applied to each webcam image, and the resulting bounded box around the eye is stored, and the eye image returned.

```
// globals
private AverageRect eyeAvgRect;    // average bounded box for eye


private IplImage trackEye(IplImage im)
{
  IplImage eyeIm = null;
  CvRect cvEyeRect = findEye(im, eyeClassifier);
  eyeAvgRect.add( scaleRectangle(cvEyeRect) );
                                    // add to other rectangles
  CvRect avRect = eyeAvgRect.get();   // get average
  if (avRect != null) {
    eyeIm = IplImage.create(avRect.width(), avRect.height(),
                                          IPL_DEPTH_8U, 3);
    cvSetImageROI(im, avRect);
    cvCopy(im, eyeIm);
    cvResetImageROI(im);
  }
  return eyeIm;
}  // end of trackEye()
```

The rectangle around the eye isn't processed directly; instead it's added to a sequence of rectangles maintained in the AverageRect object, eyeAvgRect. This object is queried for an average rectangle, calculated from all the rectangles in the sequence,

which has the effect of smoothing out slight movements in the rectangle over time. As new rectangles are added to the AverageRect object, old ones are removed, so the rectangle sequence is kept current.

findEye() starts by converting the webcam image to a grayscale, reducing its size, and equalizing it inside scaleGray(). Then the Haar classifier is invoked:

```
private CvRect findEye(IplImage im,
                       CvHaarClassifierCascade classifier)
{
  IplImage cvImg = scaleGray(im);
  CvSeq eyeSeq = cvHaarDetectObjects(cvImg, 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 eye subimage

  int total = eyeSeq.total();
  if (total == 0)
    return null;
  else if (total > 1) // this case should not happen
    System.out.println("Multiple eyes detected (" + total +
                                          "); using the first");
  CvRect rect = new CvRect(cvGetSeqElem(eyeSeq, 0));
  cvClearMemStorage(storage);
  return rect;
}  // end of findEye()
```

The detector call, cvHaarDetectObjects(), is speeded up by being told to do a quick search for the single biggest matching object. This assumes that the eye dominates the webcam image.

3. Pupil/Iris Tracking

The eye image is passed to the trackPupil() method which finds the bounded box for the pupil/iris.

```
// globals
private static final double PUPIL_SCALE = 3;
    // for increasing pupil movement relative to the eye center

private ColorRectDetector pupilDetector;

private AverageRect pupilAvgRect;  // avg box for pupil/iris
private AverageRect eyeAvgRect;    // avg box for eye

private TargetMover targetFrame;
    // the window whose target is moved by pupil/iris movement


private void trackPupil(IplImage eyeIm)
{
  pupilAvgRect.add( pupilDetector.findRect(eyeIm) );
     // find pupil/iris rect, and add to average rect object
```

```
  // get average eye and pupil rectangles
  CvRect eyeRect = eyeAvgRect.get();
  CvRect pupilRect = pupilAvgRect.get();
     // pupil coords are relative to eye image not webcam image

  if (pupilRect != null) {
    // calculate distance of pupil from center of eye rect
    int xDist = pupilRect.x() + pupilRect.width()/2 -
                                          eyeRect.width()/2;
    int yDist = pupilRect.y() + pupilRect.height()/2 -
                                          eyeRect.height()/2;

    /* scale dists, and convert to percentage positions inside the
       eye rectangle (the values may be < 0 or > 1 due to scaling)
    */
    double xInEye = ((double)eyeRect.width()/2 +
                           xDist*PUPIL_SCALE)/eyeRect.width();
    double yInEye = ((double)eyeRect.height()/2 +
                          yDist*PUPIL_SCALE)/eyeRect.height();

    /* move the target using the pupil's relative
       position inside the eye rectangle */
    targetFrame.setTarget(xInEye, yInEye);
}  // end of trackPupil()
```

The bounded box calculations are performed by a ColorRectDetector object, which I'll explain in the next section.

trackPupil() concentrates on calculating a coordinate for the iris center relative to the eye rectangle. The steps are illustrated by Figure 5.

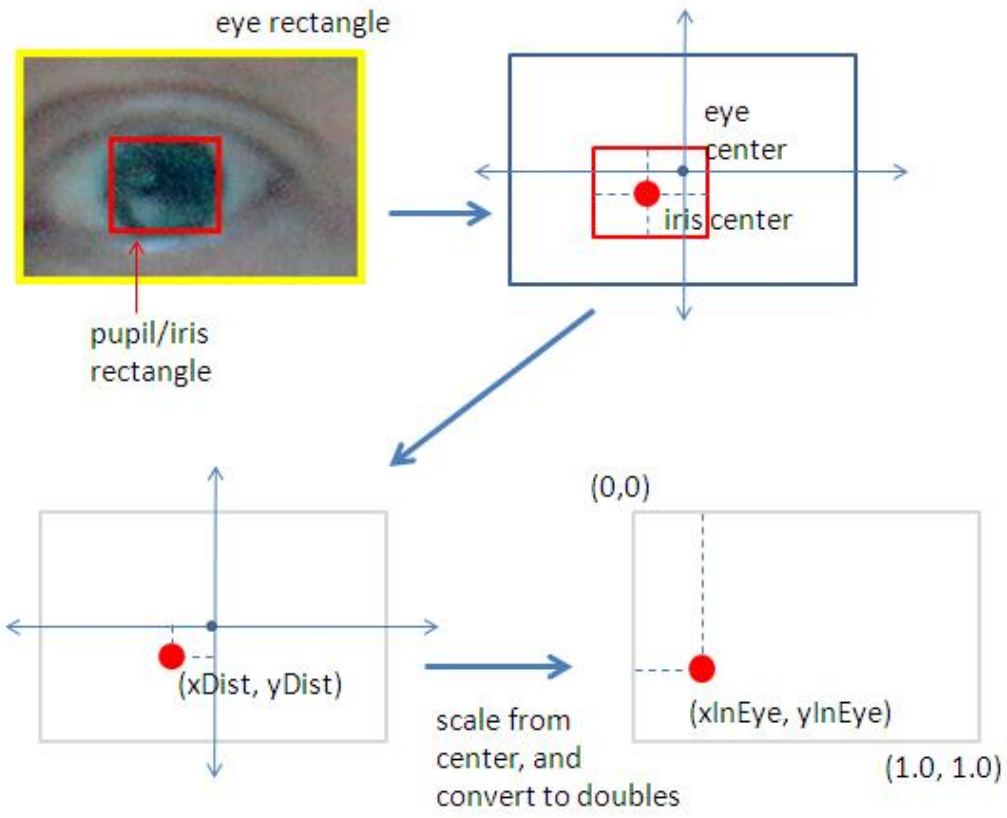

Figure 5. Transforming an Iris Position.

The (xDist, yDist) coordinate in Figure 5 (and in trackPupil()) is the position of the iris relative to the eye rectangle's 'origin'. This position undergoes two further changes to become (xInEye, yInEye) – first the distance from the origin is scaled up so that small eye movements away from the origin are magnified, making the cursor move more quickly towards an edge. Then the position is converted into doubles in the range 0 to 1 relative to the top-left corner of the eye rectangle. This makes it easier to map the point into window coordinates when it's passed to TargetMover.setTarget(). In fact, due to the scaling, the doubles may be less than 0 or greater than 1, but this issue is dealt with by setTarget().

TargetMover.setTarget() sends the position to the setTarget() method in the MoverPanel class, which converts the doubles into panel coordinates:

```
// MoverPanel class globals
private int xCenter, yCenter;    // center of the target image
private int pWidth, pHeight, imWidth, imHeight;
    // dimensions of the panel (p) and of the target image (im)


public void setTarget(double x, double y)
{
  xCenter = (int) Math.round(x*pWidth);    // convert to panel coords
  yCenter = (int) Math.round(y*pHeight);

  // keep the target visible on-screen
  if (xCenter < 0)
```



```
    xCenter = 0;
  else if (xCenter >= pWidth)
    xCenter = pWidth-1;

  // reverse xCenter so left-of-center <--> right-of-center
  xCenter = pWidth - xCenter;

  if (yCenter < 0)
    yCenter = 0;
  else if (yCenter >= pHeight)
    yCenter = pHeight-1;

  repaint();
}  // end of setTarget()
```

The doubles are limited to be between 0 and 1 so the crosshairs target image drawn by repaint() stays visible inside the panel.

Another change is to reverse the x-axis value so that the cursor moves in the same direction as the pupil from the point of view of the user. This is necessary since the webcam is facing the user, so left and right are reversed in its recorded images.

4. Blob Detection

At the start of trackPupil(), ColorRectDetector.findRect() finds a bounded box for the pupil/iris using OpenCV contour detection. The processing is very similar to the blob detection code of Chapter 5 ("Blobs Drumming"), and doesn't use any eye-specific features. The HSV color ranges are for near-black, suitable for finding the pupil (or the iris if the image is dark).

The image is supplied in the call to findRect() which starts by converting it into a HSV format, calculates a threshold image using the HSV ranges, and then uses contours to find the largest bounded box in the threshold image.

```
// globals
// HSV ranges defining the colour
private int hueLower, hueUpper, satLower, satUpper,
            briLower, briUpper;


public CvRect findRect(IplImage im)
{
  int imWidth = im.width();
  int imHeight = im.height();
  IplImage hsvImg = IplImage.create(imWidth, imHeight, 8, 3);
                                    // for the HSV image
  IplImage imgThreshed = IplImage.create(imWidth, imHeight, 8, 1);
                                    // threshold image
  // 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);
    /* do erosion followed by dilation on image to remove
       specks of white & retain size */

  CvBox2D maxBox = findBiggestBox(imgThreshed);
  // store box details in a CvRect
  if (maxBox != null) {
    int xC = (int)Math.round( maxBox.center().x());
    int yC = (int)Math.round( maxBox.center().y());
    int width = (int)Math.round(maxBox.size().width());
    int height = (int)Math.round(maxBox.size().height());
    return new CvRect(xC-width/2, yC-height/2, width, height);
  }
  else
    return null;
}  // end of findRect()
```

findBiggestBox() returns the bounding box for the largest contour in the threshold image.

```
// 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) {
          maxArea = area;
          maxBox = box;
          bigContour = contours;
        }
      }
    }
    contours = contours.h_next();
  }
  return maxBox;
}   // end of findBiggestBox()
```

The value for SMALLEST_BOX was chosen on the assumption that the pupil (or iris) is very close to the camera.