GOAL-1: Face Detection and Eye Detection:

In this project, we are going to use well-known classifiers that have been already trained and distributed by OpenCV in order to detect and track a moving face and eye into a video stream.

Cascade Classifiers

The object recognition process (in our case, faces and eyes) is usually efficient if it is based on the features take-over which include additional information about the object class to be taken-over. In this project we are going to use the *Haar-like features* in order to encode the contrasts highlighted by the human face and its spatial relations with the other objects present in the picture. Usually these features are extracted using a *Cascade Classifier* which has to be trained in order to recognize with precision different objects: the faces' classification is going to be much different from the car's classification.

Loading the Classifiers

We are going to load the desired Haar Classifier (e.g. `haarcascade_frontalface_alt.xml` and `haarcascade_eye.xml`) as follows:

```
System.loadLibrary(Core.NATIVE_LIBRARY_NAME);

CascadeClassifier cascadeFaceClassifier = new CascadeClassifier(

"D:/opencv/build/etc/haarcascades/haarcascade_frontalface_default.xml");
        CascadeClassifier cascadeEyeClassifier = new CascadeClassifier(
                "D://opencv/build/etc/haarcascades/haarcascade_eye.xml");
```

Detection and Tracking

Once we've loaded the classifiers we are ready to start the detection; we are going to implement the detection.

Now we can start the detection:

```
MatOfRect faces = new MatOfRect();
cascadeFaceClassifier.detectMultiScale(frameCapture, faces);
```

The `detectMultiScale` function detects objects of different sizes in the input image. The detected objects are returned as a list of rectangles. The parameters are:

- **image** Matrix of the type CV_8U containing an image where objects are detected.
- **objects** Vector of rectangles where each rectangle contains the detected object.

- **scaleFactor** Parameter specifying how much the image size is reduced at each image scale.
- **minNeighbors** Parameter specifying how many neighbors each candidate rectangle should have to retain it.
- **flags** Parameter with the same meaning for an old cascade as in the function cvHaarDetectObjects. It is not used for a new cascade.
- **minSize** Minimum possible object size. Objects smaller than that are ignored.
- **maxSize** Maximum possible object size. Objects larger than that are ignored.

So the result of the detection is going to be in the **objects** parameter or in our case `faces`.

Let's put this result in an array of rects and draw them on the frame, by doing so we can display the detected face are:

```
        for (Rect rect : faces.toArray()) {
                                        Imgproc.putText(frameCapture, "Face", new
Point(rect.x,rect.y-5), 1, 2, new Scalar(0,0,255));

                                        Imgproc.rectangle(frameCapture, new Point(rect.x, rect.y),
new Point(rect.x + rect.width, rect.y + rect.height),
                                                new Scalar(0, 100, 0),3);
                                }
```

The above process, we follow for eye detection and tracking.

```
MatOfRect eyes = new MatOfRect();
cascadeEyeClassifier.detectMultiScale(frameCapture, eyes);

for (Rect rect : eyes.toArray()) {

                                        Imgproc.putText(frameCapture, "Eye", new
Point(rect.x,rect.y-5), 1, 2, new Scalar(0,0,255));

                                        Imgproc.rectangle(frameCapture, new Point(rect.x, rect.y),
new Point(rect.x + rect.width, rect.y + rect.height),
                                                new Scalar(200, 200, 100),2);
}
```

GOAL-2: Pupil Segmentation using BLOB and logging eye location in text file:

For each eye detected, we can find the location of pupil area by using hough circles, simpleblob feature detector or thresholding methods. We have used simpleblob feature detector as it gives the best result.

FileWriter object is used to log eye location in text files.

Extracting the pupil from an image of the eye didn't exactly go so well using the Hough transform. How about blobs? Or rather BLOBs, which are defined as **Binary Large OBject** and refers to a group of connected pixels in a binary image. OpenCV (CV2) actually incorporates a means of finding blobs (an indeterminate shape), in the guise of **SimpleBlobDetector_create()**. How does it work?

Now **SimpleBlobDetector_create()** has a bunch of parameters that can be set – which of course is a doubled edged sword, and requires some tinkering. First it creates several binary images from the original image using the parameters **minThreshold**, **maxThreshold**, and **thresholdStep.** So if minThreshold=0 and maxThreshold=255, it will produce 256 binary images. Then in the binary images pixels (white) are grouped together. Next the centres of the blobs are calculated, and any closer than the parameter **minDistBetweenBlobs** are merged. Finally the centres and radii of the new blobs are computed and returned.
It is also possible to filter blobs by colour, size and shape:

- Colour:
  - **filterByColor=1**, and **blobColor=0** (for dark blobs)
- Area:
  - **filterByArea=1**, **minArea=100**, **maxArea=500** (blobs between 100 and 500 pixels)
- Shape:
  - **filterByCircularity = 1**, then set **minCircularity**, and **maxCircularity** (circle = 1)
  - **filterByConvexity = 1**, then set **minConvexity**, and **maxConvexity**
  - **filterByIntertia=1,** then set **minInertiaRatio**, and **maxInertiaRatio** (circle=1, ellipse = 0→1)

Lots to think about right? But there is nothing too complicated about these parameters. Once they are assigned, a detector can be created, and then blobs detected:

```
detector = cv2.SimpleBlobDetector_create(params)
keyPoints = detector.detect(im)
```

Easy right? Hold on, not so fast. You have to make sure you pick the right parameters. For the pupil, I set up **filterByArea,** and **filterByCircularity.** Considering the pupil is a circular object, circularity makes sense. Usually in an application sense, the size of the image of the eye being taken will be relatively uniform, and hence setting parameters for the upper and lower bounds of

the area of the pupil makes sense. In this case the image is about 600×600 in size. Here is the Python code:

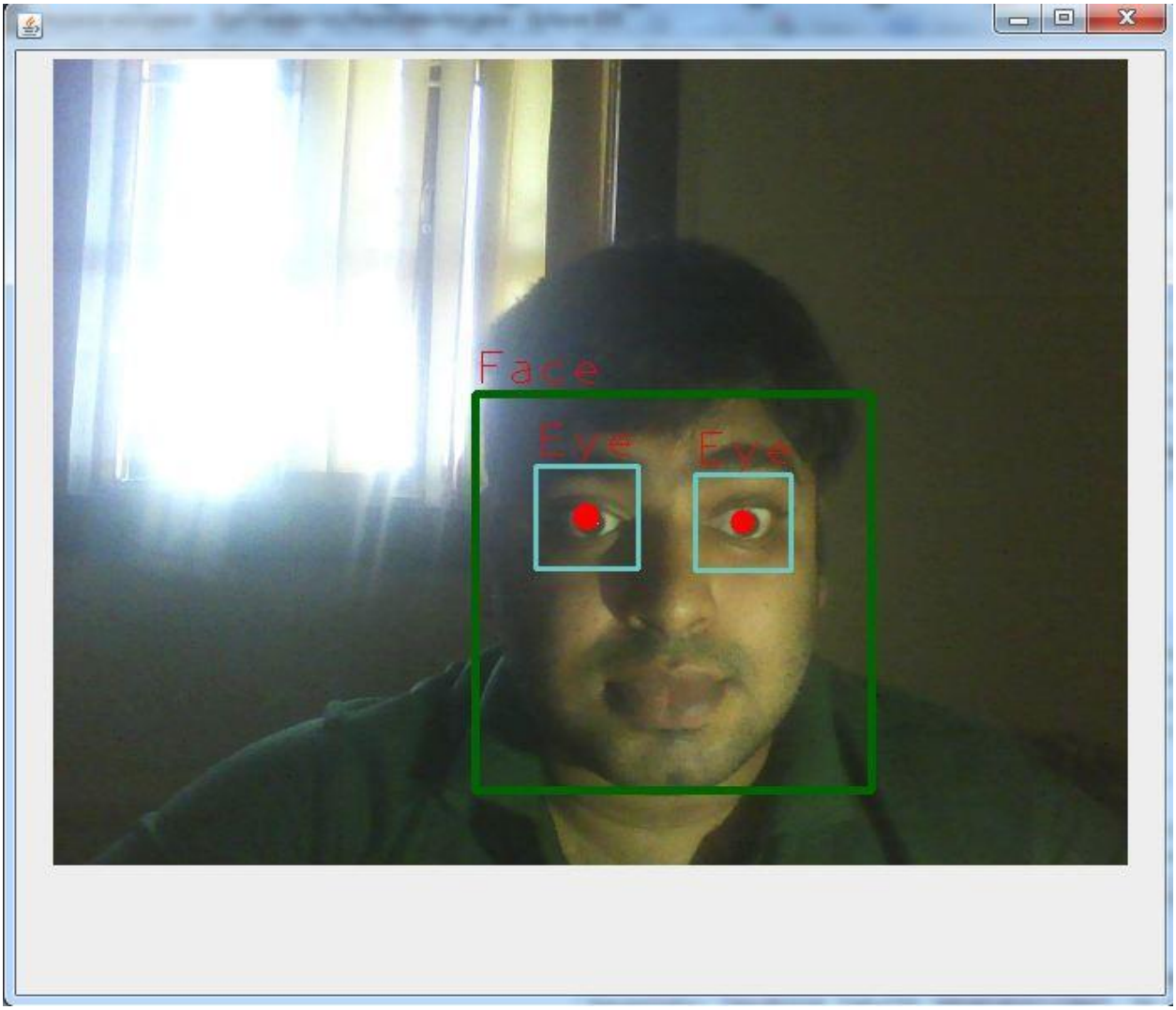
Most of the code is taken up with setting up parameters. Area is set as 3000→6000 pixels, and circularity is set to 0.5. The detector is then applied, and any blobs marked on the image.

Eyes where the contrast between the pupil and the surrounding iris is reduced, due to iris colour, may result in the algorithm failing. There are inherently many issues: (i) pigmentation of the iris: green, gray, brown, hazel; (ii) position of the eyelids, i.e. from the viewpoint of obstruction; (iii) how dilated the pupil is, (iv) interference from things such as make-up, and (v) reflections from light.

Code for the same is given below.

```
rectCrop = new Rect(rect.x, rect.y, rect.width, rect.height);

//Code Snippet for Pupil Tracking
Mat im1= new Mat(frameCapture,rectCrop);

Mat gray1 = new Mat(im1.rows(), im1.cols(), CvType.CV_8SC1);
Imgproc.cvtColor(im1, gray1, Imgproc.COLOR_RGB2GRAY);
Imgproc.Canny(gray1, gray1, 80, 100);
Mat MatOut= new Mat();
MatOfKeyPoint keyPoints = new MatOfKeyPoint();
FeatureDetector fd = FeatureDetector.create(FeatureDetector.SIMPLEBLOB);
// now edit params as you like, and read it back in:
fd.read("C://Users/HOME/Desktop/params.xml"); // wherever you put it.
fd.detect(im1, keyPoints);
org.opencv.core.Scalar cores = new org.opencv.core.Scalar(0,0,255);
org.opencv.features2d.Features2d.drawKeypoints(im1,keyPoints,MatOut,cores,2);
Point pt = new Point();
System.out.println("keypoints: " + keyPoints.toList());

try {
        FileWriter writer = new FileWriter("C://Users/HOME/Desktop/MyFile.txt", true);

        List<KeyPoint> list = keyPoints.toList();
        for (int i=0; i < list.size(); i++){
        pt.x = (int)list.get(i).pt.x + rect.x;
        pt.y = (int)list.get(i).pt.y + rect.y;
        double s=  list.get(i).size;
        int r = (int)(Math.floor(s/8));
        Imgproc.circle(frameCapture, pt, r, new Scalar(0, 0, 255), -3);

        writer.write("Eye Cordinates are:"+"("+pt.x +","+ pt.y+ ")" );
```

```
                writer.write("\r\n");   // write new line
        }
                writer.close();

        } catch (IOException e) {
                e.printStackTrace();
}
```



Thanks.