Gaze Detection for Android

Nathan Smith

CSE x0535 Fall 2012 Semester Project Final Report

Abstract

Introduction

For my semester project I chose to implement gaze detection in an Android tablet application.

Related Work

Body

I started my project by researching different gaze detection techniques. It was mentioned early on in a conversation with the Professor that one possible approach would be to detect how elliptical the pupil is. Pupil eccentricity can be a very good indicator of gaze position, however, it requires a high-resolution image and a large, very distinct pupil to work properly.

A Motorola Xoom device was used for development. The Xoom has a 10.1in, 1280x800px resolution with both forward and user facing cameras. The goal was to assume a configuration on the user squarely facing the tablet with their eyes approximately 30cm (~1ft) from the user facing camera and holding the tablet in landscape mode. The setup needed to be well lighted, with some light possibly coming from the tablet screen as well as from the environment. One drawback of too much, or too concentrated light is that it would reflect off the cornea and make it difficult to isolate the pupil. Unfortunately the tablet’s user facing camera was not able to operate in the near-infrared spectrum, restricting usage to the visible range.

Before even detecting the pupil other, more basic questions needed to be answered such as how to detect a face. OpenCV comes with a nice class called CascadeClassifier, which can use classifiers for various features in the form of xml files. The easiest way to detect a face is to use the face classifier, which returns a rectangle within which a face was detected.

The same method can be used for detecting eyes using the eye classifiers. I chose to use separate classifiers for the left and right eyes because it seemed to give a slightly larger bounding box that better included the eye corners than the generic eye classifier. I restricted the region where the classifiers could search for eyes to the upper left/right quadrants respectively to reduce search time.

Unfortunately, both the face and eye classifiers return many results, from which a single region needs to be chosen. I chose to use the largest detected region for each feature since the user’s face would be occupying the majority of the image in the assumed setup. This method seemed to work well, especially for the eyes, which often gave multiple results.

While the end goal of this project was to create a working Android application, I chose to prototype my methods in python using the webcam on my MacBook Pro as input.

As mentioned above, my first approach was to attempt to measure the eccentricity of the pupils. The first step in detecting the region was to threshold the gray image of the eyes. Choosing the correct threshold proved to be more difficult than anticipated. It turned out to be very difficult to reliably separate the pupil from the surrounding iris. Even more difficult was separating the pupil area from the dark shadow of the upper eyelid. Also, there was often non-negligible glare on the cornea. Attempting to threshold this situation often resulted in a large dark area containing the pupil, iris and arc of the eyelid shadow with large holes where light was reflecting. Pushing the threshold up with result in growing the reflection holes until, when the pupil area and shadow were separated, a rather thin crescent remained where pupil should be.

Fitting an ellipse to the resulting binary image gave heavily skewed results. It may have been possible to mask out the reflection holes, but rather than attempting to model the positions of ever-changing reflection patterns as the eyes rotated as well as modeling the appearance of an ellipse on the surface of a rotating sphere projected on the 2d plane of the image, I decided to attempt a different approach.

My new approach was to detect the pupil location relative to the corners of the eyes. The advantage of this approach is that only the eye corners and pupil center needed to be detected rather than a consistent iris blob to be fitted with an ellipse. To detect the eye corners I used the OpenCV GoodFeaturesToTrack function with the minimal eigenvalue of gradient matrices method for corner detection. Many features points were returned in the eye region and I narrowed down the results by only accepting points passing the following logic:

w = eye roi width

h = eye roi height

x, y = point coordinates

( x<(w/5) or x>((w/4)\*3) ) and ( y>int(h\*0.45) and y<int(h\*0.65) )

From the remaining points I selected the leftmost point on the left side and the rightmost point on the right side of each eye. This restriction works well, but the corner detector parameters need to be played with to find the optimal value where points are always found in the desired region without producing too many noise points.

To find the pupil center I used two techniques described by a fellow classmate for both the thresholding and pupil center locating. For the thresholding, the idea was to threshold to a certain percentage of pixels. For this, a histogram of the grayscale eye roi was computed and the value at which 7.5% of the pixels were below was used as the threshold value. All pixels with values above the threshold were set to zero, and all pixels below were set to the max value (255).

For finding the pupil center, the erode function was used on the thresholded image, assuming that the iris/pupil area was the largest blob in the image. The image was eroded, keeping track of the previous step, until nothing remained, then a point on the previous step was assumed to be close to the pupil center. This method works well but could possibly be made better by running HoughCircles on the grayscale image and selecting a circle with a center near the detected pupil location and with a similar size (half the number of steps needed to erode the iris blob).

Once the pupil center was detected it could be compared to the eye center. The eye center is computed by averaging the horizontal and vertical components of the eye corner locations. Setting the eye center as the new origin for the pupil location results in negative horizontal component for looking left and positive for looking right in the camera image and a positive vertical component for looking up and negative for looking down.

These relative pupil locations were compared to calibration points detected in a calibration mode. For this project, the detected relative pupil locations were assumed to have a linear relationship with the gaze angle. Correctly modeling the relationship based on the pupil center as a point on a rotating sphere would result in angle estimation, but the impending deadline of this project excluded this approach from being explored.

To get the slope of the relationship between relative pupil location and gaze angle, left, right, top, and bottom calibration points were used. The left and right points were used to find the yaw slope and top, bottom to find the pitch slope. The yaw and pitch were then found for the pupil centers. Using more calibration points and fitting a curve to the points to find the relationship might obtain a better angle approximation.

These angles could then be used to find the gaze location on the screen based on the assumed user/tablet orientation. Using basic trig functions and comparing to the known screen and tablet dimensions, the position on (or off) the screen could be found.

As mentioned above, calibration points are needed to find the relationship between pupil location and gaze angle. For my application, four points are used:

Left: (0, 400)

Right: (1280, 400)

Top: (640, 0)

Bottom: (640, 800)

Calibration mode consists of:

* Displaying a target on the first point
* Waiting for the user to acknowledge they are looking at the point (tap the screen or press ‘n’ for Python version)
* Collecting a desired number of samples (5)
* Averaging their pupil locations
* Moving to the next point and repeating until all points have been processed

Calibration mode is automatically entered upon starting the app or Python prototype. In the Python version pressing ‘c’ can reactivate it. In the app, it should be a menu item.

All these steps lead up to a gaze location being found that can be used for heat maps, controlling input, etc.

Performance Experiment

Unfortunately, there is very little to show in terms of concrete performance evaluation. In fact, the Android application is not complete at the time of submission. It is missing the basic gaze detection described above and consists of face/eye detection and a partially implemented calibration mode. It does include the gaze location estimation based on gaze angle, but that is useless without gaze angle detection working.

The Python prototype is much further advanced than the Android application containing all of the steps described above with the exception of calibration mode being partially implemented and the rest relying on dummy calibration points until the mode is complete.

The Python version displays a window with the same dimensions as the Xoom tablet, which the user should position centered in the screen below the webcam. Calibration points are displayed in the output window and detected gaze location feedback is provided by the blue dot displayed.

Sheer frame processing speed is lacking for both the Android app and Python prototype with an estimated 3 frames per second being processed in both (keep in mind the Android app is only doing face/eye detection), but this aspect can probably be dramatically improved by optimizing the algorithms and restricting search operations based on previous frames if possible.

Aside from frames processed per second, the gaze estimation is not consistent in multiple respects at this point. Firstly, it often fails to detect locations in every frame. It appears to detect about 1 in 5 frames depending on eye appearance, lighting, etc. This usually comes from failing to find eye corner locations for one or both sides. This may be solvable by tweaking the GoodFeaturesToTrack parameters to give more or better points.

The other common inconsistency comes from the detected gaze location varying (sometimes greatly) despite actually looking at the same target location. I suspect that most of the problem here also lies with detecting eye corners. Sometimes the corner location is not at the eye corner, but rather a nearby ‘noise’ point that happened to be the leftmost or rightmost point depending on the which side is being searched. Sometimes the eye corner detection selected points on the pad arms of my glasses if they were in the eye region.

It may be possible to further restrict the region to look for corners based on previous frames or other eye characteristics. Perhaps edges could be detected and the process could look for a characteristic eyelid arc and choose corner points that lie on that arc.

Face and eye locations generally are consistently detected correctly as long as the eye is open. Opening the eyes wide seems to give better results overall.

Conclusions

I have learned much from this project. The main thing that I have learned is that in theory Computer Vision is fairly intuitive, but in practice it is hard. Thankfully there are libraries out there such as OpenCV that make simple things simple and hard things possible, especially when combined with a dynamic language like Python and libraries like numpy and scipy for easy matrix manipulation. I have also learned that CV algorithms can be computationally expensive and that effort quickly adds up when trying to do high-level tasks.

More specific to this project I have learned that gaze detection is nearly as simple as it may seem at first. It very much depends on environmental variables such as lighting, hardware such as camera quality, and user orientation. My project is nowhere near perfect (let alone complete), but it has been a valuable learning experience and I have developed an interest in many Computer Vision topics.

Future Work

There remains much to be done for this project and many areas that can be improved. Here are some of these area and other ideas I have (many of these topics where mentioned above):

* Model the relationship between relative pupil center and gaze angle as a point on a rotating sphere.
* Be able to use more calibration points.
* Compute ongoing heat map from gaze locations.
* Improve eye corner detection.
* Improve pupil center detection, possibly by using center blob location and approximate iris size to filter detected circles (HoughCircles) to get a better, possibly more consistent pupil center.
* Optimize detection for each step by using results of previous frame as a starting point.
* Finish implementing calibration mode.
* Implement everything that is in the Python prototype in the Android application.
* Android application features:
  + Allow for multiple detection algorithms/versions and let user choose and compare.
  + Implement test mode that displays random points, waits for user acknowledgement, and compares detected gaze location with points shown to analyze performance.
  + Package up app core functionality into a library to be used with other applications either to allow eye based input (potentially cpu/memory limited) or capture face and screen videos for post analysis either in an analysis app or externally.

Acknowledgements

I would like to acknowledge/thank:

* Dr. Flynn for his advice and guidance on this project.
* The ND Center for Research Computing for supplying the Xoom tablet.
* My classmates for their ideas and interest.
* My wife Jessica and son Aaden for giving me time to work on this project.

Citations