COST EFFECTIVE IR-FREE EYE TRACKING ON MOBILE DEVICES

by

JACOB KOKO
and
SEBASTIAN FLOREZ

Advisor
DANIEL PLANTE

A senior research proposal submitted in partial fulfillment of the requirements
for the degree of Bachelor of Science
in the Department of Mathematics and Computer Science
in the College of Arts and Science
at Stetson University
DeLand, Florida

Spring Term
2016
# TABLE OF CONTENTS

TABLE OF CONTENTS ...................................................................................i

LIST OF FIGURES ...........................................................................................ii

ABSTRACT ...........................................................................................................iii

1. Introduction .................................................................................................4

2. Related Work ..............................................................................................5

3. Implementation ...........................................................................................8

   a. Iris Detection Improvements ................................................................14

   b. Porting to Mobile Devices ....................................................................18

   c. System Calibration ...............................................................................20

4. Results .......................................................................................................25

5. Conclusion ................................................................................................28

REFERENCES ..................................................................................................vi
LIST OF FIGURES

Figure 1: Image Processing Steps
Figure 2: Calibration Stage Reference Points
Figure 3: Four Quadrant System From Mapped Eye Locations
Figure 4: Iris Coordinates Before Averaging
Figure 5: Iris Coordinates After Averaging
Figure 6: Eye Detections With Peripherals
Figure 7: Front Facing, Multiple Skin Tone Eye Detections
Figure 8: iOS Eye Tracker
Figure 9: Android Eye Tracker.
ABSTRACT

The purpose of our project is to improve upon modern eye tracking technologies that are used for psychological experiments. As it stands, the technology used for such experiments is expensive, non intuitive, and utilizes infrared cameras. We plan to offer an alternative to existing technologies by introducing an effective mobile solution that does not make use of any infrared technology. More specifically, our technology can act as a low cost substitution for eye tracking and data analysis because it only requires a standard mobile device. We believe that our research is applicable to more than just psychological experiments. For example, brand marketing could find substantial use in this technology. This research is largely theoretical in regards to the mathematical algorithms used for eye tracking, but the implementation is quite extensive as well. Therefore, we intend to delegate the work into two separate areas including mathematics and software implementation.
1. Introduction

Many modern eye tracking technologies are large, non intuitive, and expensive. For these reasons, eye trackers are often inaccessible for individuals and small organizations. This project aims to solve that problem by introducing a lightweight, cost effective alternative to modern eye tracking technologies. By operating as a stand alone mobile application, we aim to develop a portable eye tracking system that does not rely on any external hardware. To aid in the implementation, we depend on an open source computer vision library called OpenCV. OpenCV provides a common infrastructure for computer vision applications, containing thousands of optimized algorithms that are applicable for various different use cases within the field of computer vision and machine learning[14].

In our case, we are using OpenCV to aid in tracking a person’s face and eyes. The ability to track a user’s eyes is pivotal to our project, therefore, we are focusing a majority of our efforts on tracking eye movements in real time with as much precision as possible. We made the decision to build our first prototype using a computer’s web camera. After attaining an adequate level of accuracy and efficiency on the web camera, we decided to port the application to mobile devices. To achieve accurate results, our system has been tested under various conditions including different lighting, eye colors, eye peripherals, skin tones, and camera qualities. Moving forward, we intend to offload some of the heavier computations onto an external service to decrease system resource usage. We find this to be the most optimal solution, as hardware acceleration becomes an issue on mobile phones.

Our project aims to introduce a level of simplicity to eye tracking systems by eliminating the need for any external hardware. Many modern eye tracking technologies use infrared light to
produce a corneal reflection that is easily detectable in a video frame. We recognized this
problem early on, and our solution involves applying a series of preprocessing steps to each
video frame. This allows us to achieve a well defined image for processing, without the use of
any external hardware. This is discussed more in the implementation section.

2. Related Work

Eye tracking systems have several use cases that span over various different industries
including user experience and interaction, marketing and consumer research, infant and child
research, psychology, neuroscience, education, clinical research, and human performance [6].
Eye tracking is used within the field of user experience and interaction because it allows
developers to optimize their interfaces based on overall user engagement and activity. By
tracking a user’s gaze as they navigate through an application, developers can pinpoint areas of
the interface that are not performing well. These weak spots can be refined, greatly improving
the user experience of an application. Within marketing, eye trackers are used for measuring a
consumer’s response to a particular message or product packaging. This allows marketers to
develop products that are optimized for their target consumer’s eye, maximizing the overall
potential for sales.

Furthermore, eye tracking technologies have been used extensively within the field of
psychology. It has allowed psychologists to better understand how humans process the
information we encounter visually. For example, sports psychologists use eye trackers to study
professional athletes and their response to stimuli. By doing so, researchers learn about the minds
of professional athletes and how they differ from non professional athletes. Eye tracking
technology is also used by developmental psychologists to get a better understanding of how
children develop both cognitively and perceptually in the early stages of their lives[6].

Ultimately, eye trackers have a variety of use cases making them extremely powerful in a variety of industries.

Today, the leader in producing high quality eye tracking technology is a company named Tobii. Tobii develops a wide range of eye tracking products including a personal peripheral device for laptops, wearable glasses, and specialized speech generating devices that can be controlled with gaze interaction. They first launched in 2001, releasing the first plug and play eye tracking system by 2002 [12]. Since the companies start, it has been on the forefront of creating cutting edge eye tracking technologies for widespread distribution.

**Tobii Pro Glasses 2**

The Tobii pro glasses provide a lightweight solution for eye tracking in real world environments. This tool allows researchers to see what a person is looking at in real time. The glasses have a wide angle HD camera that operates at 25 FPS. To analyze natural eye movement accurately, Tobii created the ‘True View’, consisting of four wide angle HD cameras. This system is often used for research, as it performs with high precision. Tobii’s proprietary 3D modeling is used by the glasses to enable robust eye tracking on any type of eye. The glasses come in a package starting at $14,900. [8]

**Tobii TX300 Eye Tracker**

The Tobii TX300 acts as a standard for eye trackers. It performs with exceptional precision, making it perfect for research. This unit has a gaze sampling frequency of 300 Hz, while allowing for large and fast head movements. The ability
to move one’s head around promotes natural human behavior, improving the quality of the research. If an experiment requires real world interaction, the eye tracking hardware can be removed from the 23” monitor. [7]

**Tobii EyeX Controller**

One of the latest products released by Tobii is the EyeX Controller. This device comes with a SDK for developers, and it starts at $139. Tobii is the leader in developing high end eye tracking systems, and this product allows them to step away from their usual target market by making an affordable eye tracking system that can be used by individuals and small organizations. Although cost effective, the unit performs with high precision at a distance of 18-40”, with a data collection rate of 60 Hz. For our project, we intend to develop a cost effective eye tracking system as well, however, our implementation aims to eliminate the use of any external hardware. [9]

Offering more affordable alternatives, The EyeTribe is a company that provides cost effective eye tracking hardware which includes a software development kit starting at $100. The company provides each buyer with a custom built camera that is meant to be mounted on whatever interface you would like to track eye movement from. This product is unique because the hardware they have developed is extremely small and portable. The hardware can fit on anything from a tablet, to a large computer, making it extensible across multiple devices. The EyeTribe provides an SDK for developers to obtain data and build applications using their eye tracker. The camera has a wide area of view and uses the latest in infrared tracking technology to track the movements of a user’s pupil with high precision. The product works best in an indoor
environment without direct sunlight. With separate modes for collecting gaze data at 30 Hz or 60 Hz, this solution is perfect for individuals seeking a lost cost eye tracking system. [10]

3. Implementation

Our initial implementation makes use of a computer, a web camera, and the OpenCV computer vision library. We arrived at this solution after researching several different eye tracking systems and their implementations. Most eye tracking implementations make use of infrared technology, therefore we were presented with a road block early on. Infrared light makes eye tracking easier because it helps to illuminate several regions of the eye, creating a set of reflections that are easily detectable in a video frame. Using these reflections, a gaze can be calculated using their relative positions [11]. The downside of this technique involves getting an accurate view of the eye. This is very difficult, especially with the introduction of head movement [11].

Our goal is to develop a cost effective eye tracker that does not require any external hardware. Using infrared technology is not feasible when thinking about porting the application to mobile devices. We went through many iterations of our initial paper before deciding on a quadrant based application that maps a user’s eye position to a location on the screen. By doing so, we can determine the variance in eye movement to find out which quadrant their gaze is fixated on. To arrive at such a prototype, we first had to tackle the problem of tracking a person’s eye. This is where OpenCV comes into play, as we used OpenCV to detect a user’s face. Using the facial information, we then extract the eyes from the user. With this data available, we then locate the user’s iris as accurately as possible.
To obtain the user’s face we use OpenCV’s cascade classifier (a machine learning algorithm used to detect faces) and train it with sample images. Once the classifier has successfully detected a user’s face, we then take the result and feed it to another cascade classifier (this one is trained with eye samples) to detect the user’s eyes. Once we have successfully located the user’s eyes, we are able to take those results and perform some preprocessing steps in order to find the iris using a Hough circle transform function provided by OpenCV. Developed by Paul Hough for physics experiments, Hough transforms were originally used for finding lines in an image. The general idea behind the Hough transform is that any point on a binary image is included in the set of all lines [1]. However, the algorithm has since been generalized to recognize other forms, in our case circles. The Hough circle transform is far more complex than a typical Hough transform because each circle is represented by the equation \((x - h)^2 + (y - k)^2 = r^2\), where \((h, k)\) is the center of the circle, and \(r\) is the radius. Therefore, a three dimensional accumulator is required for the circle transform, as opposed to the two dimensional accumulator plane that is typically leveraged by the Hough transform algorithm[13]. The additional dimension results in more expensive computations, ultimately increasing memory usage and execution time[1].

To address this problem, OpenCV uses a complex solution known as the Hough gradient method [3]. First, the image undergoes a canny edge detection phase in order to highlight the frames structural information. Following this step, a slope is calculated for every non zero pixel using a function provided by OpenCV called \(cvSobel()\). This function works by comparing different regions of pixel intensities in order to calculate the x and y sobel derivatives. For every point that falls along the slope within a specified minimum and maximum distance, its value is
summed together in a two dimensional accumulator plane, eliminating the three dimensional problem presented earlier. The potential circles are then selected from the set of all accumulator values, provided that their value is larger than a specified threshold \([1]\). Although this algorithm is less intensive, the Hough gradient has several disadvantages. One major disadvantage stems from the occurrence of nested circles. In this case, only one circle will be stored, and it’s often the larger circle surrounding the smaller one \([1]\). We encountered this problem early on when testing on static images. Although a larger circle was detected, its center was often positioned directly over the iris. For this reason, this particular downfall has not been of major concern.

Another issue with the Hough gradient occurs when it is given an accumulator value that is too low \([1]\). When this occurs, the algorithm takes an extensive period of time to execute as it must process all non zero pixels for every potential circle. This was an issue that we encountered early on, however, our modified binary search resolved this issue.

OpenCV abstracts these implementation details away from the developer by providing an efficient HoughCircles function. The most difficult part of getting the function to operate properly was finding a way to dynamically generate the accumulator threshold value. The accumulator threshold acts as a guideline for the precision of the circle finding algorithm. The larger the value, the higher the probability of finding accurate circular forms. When the accumulator threshold becomes lower, the function returns more circles, but it is likely many of them will be inaccurate. At first, we hard coded values for the threshold but our results were poor. We later discovered that the accumulator threshold value varied per image, meaning it must be generated dynamically. After discussing the problem, we implemented a modified binary search to generate an accurate threshold for each image. The search works by passing
different threshold values to the HoughCircles method. If the method returns more than one
circle, we know that the threshold must be higher than the current value, allowing us to focus on
only larger values. Otherwise, if the method returns no circles, we know that the accumulator
threshold must be lower, allowing us to focus on values that are strictly less than the current
value. Therefore, in each iteration we eliminate half of the potential threshold values, resulting in
an algorithm that has an average complexity of $O(\log n)$. Once we find a threshold that returns
only one circle, we return that value. Otherwise, we keep track of all the threshold values in a
map, and we return the threshold that has the minimum number of circles. This algorithm is
effective, but we plan on refining it in future implementations. It should be noted that the
modified binary search was developed by us for the purpose of the project.

Once we have an image of the eye itself we transform it into an 8-bit single channel
grayscale image, then we apply a histogram equalization to accentuate the light and dark areas of
the image. Histogram equalizations are the most commonly used technique for expanding the
range of an image. Since the image is 8-bit, it has values that range from 0 to 255. When the
values remain uniform in a cluster, the image becomes poor in quality. The histogram
equalization aims to solve this problem by mapping the images original values to a more well
distributed range of values [2]. After the histogram has been applied, we then apply an inverted
threshold that binarizes the image in order to reduce noise and highlight the dark areas. The
inverted binary threshold works by setting any value greater than the threshold equal to zero. If a
value is not greater than the threshold, then it is set to a predefined maximum value [4].

Following this step, we remove white artifacts from the image to make finding the iris
easier by applying a Contrast Limited Adaptive Histogram Equalization (CLAHE )[5]. The
CLAHE is used to correct any data lost from the application of the first histogram equalization. Data loss occurs because histogram equalizations are not applied to a specific region of an image, therefore each point is being mapped to a new uniform distribution. The CLAHE solves this problem by breaking an image into 8x8 grids of tiles. After the tiles are created, a histogram equalization is applied to each tile individually. To account for any noise in the image, a contrast limit is defined and any histogram above the specified limit will be broken down and distributed to other tiles before another histogram equalization is applied. Finally, a bilinear interpolation is applied in order to smooth over the tile borders [5].

For the last step in the preprocessing phase, we apply a gaussian blur in order to further smooth the image. This is a crucial step in order for the Hough circles transform method to work effectively because it is less accurate when noise is present in an image. After all the preprocessing steps have been performed on the image of the eye, we apply a Hough circle transform to find the iris. This process was first performed on static images until we reached a high level of accuracy. After we achieved accurate results on our dataset of static images, we started to feed our program video frames in real time. The preprocessing phase of each image is shown in the figure below.

![Figure 1: Image Processing Steps](image)

After having gone through various static images for testing the initial process for eye detection, we moved forward to real time eye tracking. For this next step, we took what we had
learned from our experimentation with static images and applied it on the frames that are obtained from a computer’s web camera. Each video frame is passed to our algorithm which first finds the user’s face. From the face, the user’s eyes are extracted and cropped out. Furthermore, by using the cropped out eye images, processing for the iris becomes much more efficient as we do not have to process the entire image. This phase of the implementation was fairly easy to transition from static to real time, as the only difference was passing in a video frame to our algorithm. Of course, this presented a major challenge in terms of testing because we could no longer generate diverse datasets within seconds. This means we now had to simulate those environments ourselves using different lighting, and eye types. Therefore, the testing phase of the real time implementation is far more extensive. The next stage of the implementation phase involves developing a calibration interface for the system. This will result in an eye tracker that is tuned dynamically for each user.

With the initial implementation completed, there were two core components to focus on moving forward. First, we needed to iterate on our algorithms to improve the accuracy of the iris tracking process. Second, we needed to focus on resource intensity and reducing the overall requirements of the system, while getting the system integrated onto mobile devices. In regards to mobile, we decided to target the two most common platforms Android and iOS.

**Iris Detection Improvements**

Our initial approach to developing a more accurate product involved looking at the eye regions that are being passed into the classifiers. This was due to realizing that our implementation was returning circles that were slightly too far from the eye itself. This led us to believe that the classifier was being handed too much noise and it was returning too many
outliers which threw off the precision of the eye tracking. The first steps that we took to increase
the accuracy of the tracking was to adjust the regions that were being passed to the classifiers.
We displayed the regions that were being used and fit them to more accurately reflect the actual
eye regions and reduce the noise. This resulted in less outliers being returned and also more
accurate eye regions for iris detection. The classification now more accurately reflected the
user’s eye region and the iris detection portion of the post processing steps gave us more precise
results. This process took numerous iterations because our system uses several randomized
numbers that we arrived at from testing. Unfortunately this was not something that we could
avoid, as the nature of the calculations require these values for accuracy. This was the first step
toward completing the first iteration of iris detection. This improvement snowballed into several
other incremental improvements that pushed our system’s accuracy further, ultimately leading to
the results shown below.
With the adjusted eye regions, our system was drawing the iris in a relatively stable manner but there were still irregularities with iris coordinates. After graphing the results, it was apparent that the system was still affected by the introduction of noise. In this application, noise is often a result of two factors. In a real time application the classifier is never guaranteed to be accurate one hundred percent of the time. This means that it can mistake an object in the background as an eye, resulting in outlier coordinates. Another major concern directly linked to this problem is the environment that the system is deployed in. We found that the environment played a major role in the accuracy of our system. For example, a room with even lighting produces much better results, and having a stationary background reduces classifier error. To account for the noise introduced by these two factors, we decided to average each potential iris coordinate over a fixed set of frames. By averaging the points we smooth over any misleading coordinates detected by the classifier at runtime. Each potential iris point is bucketed into a list depending on which eye it belongs to (left eye or right eye). After a fixed amount of frames have elapsed, in our case eight, an averaging function is passed a list of points for each individual eye. After the points are averaged, an iris location is returned to be displayed on the screen.

We found that the averaging worked extremely well and our iris detection was performing with a relatively high accuracy. To test our results, we plotted a list of iris coordinates that were collected by having a user stare at a fixed location on the screen. The graph below shows that a majority of the coordinates remained within the iris bounds, as we had hoped. However, when the user’s focal point is shifted from the center of the screen, the iris coordinates
become less accurate. The data collection process relies heavily on the environment, and we must also account for user behavior. For example, staring at one fixed point for a long period of time is difficult for humans. The eye tends to jump around, which is bound to happen. This adds noise to our data set, ultimately skewing results. We are also testing without a head mount so slight deviations in head position can also affect the overall results. We attribute most of the outlier coordinates to changes in the environment and user behavior.
Porting to Mobile Devices

After improving the model’s iris detection accuracy, we began to deploy our system onto mobile devices. In particular, we targeted two platforms for development including Android and iOS. The model layer was completed so we did not expect to encounter many challenges. However, getting the system deployed onto iOS devices proved to be more challenging than we originally anticipated. OpenCV provides iOS developers with a framework that ties into their extensive library of algorithms. Unfortunately, the framework is a mixture of the languages C++ and Objective-C(Objective-C++), rather than the native language Objective-C. This introduced several problem right from the start because it increased compile times, while reducing the overall readability of the code base. C++ and Objective-C are fundamentally different programming languages by design, so mixing them is not ideal. I found that it often leads to undefined behavior that becomes increasingly difficult to debug.

One of the challenges introduced by the use of Objective-C++ was manual memory management. Our system was previously written in Java which handles the management of memory for the programmer by using garbage collection. With Objective-C++ the programmer must manually release any memory that is allocated on the heap which is prone to errors and memory leaks. When considering a real time application, memory leaks will scale quickly, degrading the system’s quality within seconds.

The model layer had to be rewritten to conform to OpenCV’s Objective-C++ function calls. The biggest challenge was updating the model to detect and crop the proper face regions.
To do so, it was mostly trial and error as we had done with the web implementation. The cascade classifiers were more sensitive on the mobile device, so I had to pass the eye classifier a minimum and maximum threshold size. Without passing the classifier the threshold sizes for detection, no eyes were returned. This was less of an issue on the web implementation, in which we only set a minimum threshold value. After getting the model converted to Objective-C++, we recognized that its performance was reasonable, but poor compared to its web counterpart. The CPU usage was reaching 150%, causing the iPhone to heat up rather quickly. The face was being detected and drawn properly but the eyes were not always aligned when displayed on the device.

After researching the matter further, we decided to try Apple’s built in face detection class. We made this decision with performance in mind, as we know Apple optimizes its frameworks for the device. Under the hood, this framework interfaces with OpenCV, but it can be leveraged using native Objective-C. Developed as part of Apple’s Core Image framework, the CIFaceFeature class abstracts face detection away from the programmer. A video frame is passed to the Detection class, which returns an array of face features. The face features are then leveraged throughout the model. The detector returns the left eye position, right eye position, face bounds, and mouth position. It also has booleans to indicate whether a particular eye is open or closed. By using the Core Image framework, CPU usage slightly improved, and the device did not heat up as quickly. The detection code became more modular and efficient. However, drawing in real time was still a challenge despite having an optimized feature detector.

Drawing the iris has been a challenge with iOS. The reason for this is because each frame received from the camera must be transformed to conform to the Core Image coordinate system
before it can be displayed on the screen. The Core Image coordinate system sets its point of origin at the bottom left corner, but the video frame is returned as a UIImage, which conforms to a different coordinate system. UIImage is part of UIKit, which has its point of origin at the top left of the screen. The image transformation implementation is trivial, but it results in choppy drawing results. Each facial feature must be translated to conform to the proper coordinate system as well. Apple released a sample project which proved to be a useful starting point for face detection with Core Image, however, it had an abundance of legacy code. When using Apple’s face detection framework rather than OpenCV directly, the way in which video frames are captured changes. Without OpenCV, capturing frames becomes more difficult and the code base becomes quite bulky. New challenges were introduced, such as aspect ratio differences between the video capture, and the actual screen. On the device, head movement greatly affected the system’s accuracy. When the user moves, the eye regions start to bounce around becoming uncentered. The system works best when the device is in a fixed position, and the user does not move. When the environment is tuned properly, the iOS eye tracker performed quite well. There are several improvements to make moving forward. For one, we would like to support other device orientations besides portrait mode. In addition to that, we want to improve the iris drawing accuracy, making it less choppy. Moving forward, it would be beneficial to do a thorough analysis of using OpenCV versus Apple’s Core Image framework for feature detection. Both had their benefits, but ultimately we went with the selection that had the best overall performance from our testing.
Beginning the port to android devices proved to be easier than previously anticipated. This was due to the well integrated OpenCV bindings that the android platform has. Since the platform is built off of java and the initial product was built using java on the desktop the android port was facilitated by the familiarity. The android platform also helps the process of setting up the OpenCV library on the device by providing an application which ensures that the library being used by the device is the current best and most stable for that specific hardware. Of course this does not mean that there were no obstacles to overcome. Unlike a desktop environment the resources that you have at hand are limited. The amount of RAM and CPU that an application
has at hand is throttled by the OS. The pre-processing steps of our algorithm work fine in an environment where the resources are plentiful but it's a different story when you have to manage how much you use and at what time. The phones are especially susceptible to memory leaks and that is something that will quickly take up the memory of the phone leading to “lava” hot phones and quick battery depletion. As such managing the resources used and not allowing the procedure to potentially do damage to user’s phones and dramatically affect the performance of the implementation. With that in mind, although there are issues, the move from desktop to android has been eased by the tightly coupled integration of OpenCV and android. Initially the move from desktop to android started with porting our current logic, this step wasn’t hard because the library bindings are essentially the same. The hard part comes in the form of porting the MVC logic as the mobile platform on android couples view and logic together in the form of activities. It is a bit harder to separate the logic correctly without running the risk of leaking memory. Leaking memory when trying to implement a form of MVC on android happens when references to an activity are passed to other classes and when that activity is paused, and reconstructed via the activity lifecycle that reference is lost. This is mainly due to the implementation of the view lifecycle on the platform, changing the orientation or navigating to a different view and returning causes the activity to be rebuilt and any references to the previous activity for the same view to be lost. As such I have limited the application to run in landscape mode as the frame returned by the OpenCV library is always landscape due to their implementation and it would be too expensive to do image translation in real time. Although there are platform specific issues that must be addressed there are also implementation specific details that should be mentioned. I said before that the library bindings for the android platform
are essentially the same, this is true but the way they are implemented are slightly different. For example loading in the cascade classifier models for the haarcascade and the lbpcascade was moderately time consuming. As opposed to the way they are loading in the java environment, via path to the file on the machine, on android they must be loaded as raw resources and read into memory at the initialization of the program. This step consumed a bit of time as figuring that out took a while as the documentation did not mention the specific steps that had to be taken to load the models in. After that there had to be some changes to the actual classifiers and the relative size of the image that was being classified. Since the logic that had been developed was for a system that had a vast amount of resources at hand the performance was initially horrible on the mobile device, not to mention the “lava” hot problem was very prevalent. To mitigate this issue the regions that were being classified were scaled to reduce performance hits from classification time. We went from classifying the entire frame each time to only classifying a frame with a relative size of 30% to 50%. That change helped initial performance by increasing the framerate from around 1.5 fps (frames per second) to 25 fps (just for head classification), along with reducing needed processing power. The other change that was made was in the form of passing in more accurately refined regions to the eye classifier. That change brought the overall framerate of the application from about 3 fps to 16 fps on average. Those changes made the android application actually feasible as before the app could only be run for a couple seconds before the phone got too hot. After the changes were made the application can be run indefinitely but it still runs the phone hot after a while but nowhere near what it was like before. The increased performance not only allows the application to actually be run but it also makes it user far more user friendly, since increasing the framerate makes it more responsive. There are still
more refinements that can be made to make the android application more precise and actually usable in an real world environment but this initial progress is definitely a huge step in the right direction as currently the entire system is running on an android device without the need for a backend. This gives major insight into the feasibility of eye tracking on mobile devices. Below are images from the android application running the eye tracker.
In order to achieve accurate results on a wide array of eye types, a four quadrant calibration system will be leveraged. This phase of the implementation involves building upon our existing architecture by adding the gaze tracking layer. As such, the calibration stage requires that the current eye tracking algorithms be as accurate as possible. The user’s screen will be divided into four quadrants which we will be used for rudimentary gaze tracking. Before viewing the four quadrants, each user must go through a series of calibration steps which involve having them look at various reference points placed at critical locations on the screen (in this case upper left and right, lower left and right, and the center) as shown below.

**System Calibration**

![Android Eye Tracker](image-url)

*Figure 9: Android Eye Tracker*
We would then store the location of the user’s eyes at the various reference points by using a physical queue from the user to indicate that they are looking at a particular point. In this case a mouse click would trigger the action, however, voice commands would also suffice. At this point, the system will be calibrated for the user and a unique user specific accumulator threshold value will be generated for use by the Hough Circles function provided by OpenCV. With this data in hand, the user can proceed to the four quadrant gaze tracking view. Using the information previously obtained from the calibration stage, we would then map the data obtained from the eyes onto their corresponding positions shown on the screen below.
Figure 3: Four Quadrant System From Mapped Eye Locations
After the points have been mapped to pixel locations on the screen, we can use relative positioning to determine how much a user has adjusted their gaze. This is done by using the eye data points collected during the calibration stage as a frame of reference. The gaze tracking functionality will be a new layer built on top of our current system.

With the knowledge we have gained through the development of our current system, there are several limitations that we are aware of moving forward. Hardware limitation is the biggest of our concern. Currently the system pre processes every video frame by applying complex algorithms at the specified frame rate of the computer’s camera. Although the current system operates on a satisfactory level, it is important to factor in the hardware limitations presented by mobile devices. In its current state, there are several inaccuracies with the eye tracking system. One major concern we was the dynamic generation of threshold values for each video frame. This was leveraged by averaging over frames in multiples of 10. As mentioned above we average the calculations that are received over every ten frames. This allows us to increase accuracy while decreasing computations. Although this does help for precision it adds time for computations to be finished, at least in terms of data collection. Hardware on phones is still a concern as we have mentioned above, keeping track of memory and leveraging performance will be a very real obstacle moving forward.

Hardware acceleration is a big part as to why the algorithms employed by OpenCV have such fast performance. On a standard desktop or laptop, hardware acceleration is rarely an issue, often performed with ease. On the other hand, with mobile devices hardware acceleration becomes an issue of serious concern. Performing computationally intensive computer vision algorithms on a mobile phone will cause the user’s device to heat up very rapidly due an
overworked processor. The graphic engines and processors contained within most modern mobile phones are capable of handling an application of this nature, but the computations performed when processing and displaying images are extremely expensive. To avoid bad performance and damage to hardware, we have decided that offloading the computations onto an external server and returning the results to the phone might be the best option. This would involve setting up an external service hosted on a server that could receive a request from a phone, perform the necessary computations, then return the results of those computations to the phone so that the information can be parsed and displayed to the user.

With the external service in place, certain preprocessing steps can be leveraged and in turn keep us from performing any unnecessary steps on the phones and thus reduce the load. The information passed to the service will have to be handled appropriately in order to avoid the issue of attempting to send too much data. In addition to the data being passed to the external service, it is also important to handle the data being passed back effectively in order to ensure that the results can be easily interpreted and displayed. While adding an external service, we would have to account for any lag time between the communication of the phone and the service, especially if the phone is passing large amounts of data in a short period of time. Typically, external services are accessed via asynchronous network requests from a mobile device. This will become an issue, as it is not possible to execute large network requests in the time that a user adjusts their eye.

Another positive outcome of implementing an external service is that it presents us with an opportunity to display the results from a user’s mobile experience on a computer. We have the ability to do so because the frame information from the mobile phone is being passed to the web
service, and the result is being passed back. Therefore, we can simply pass the result to a computer for displaying the data on a web dashboard. The web dashboard would provide users with a data visualization tool. Adding a backend service will also allow for the possibly of an array of mobile phones that work together in unison to gain a more accurate representation of an individual’s environment. By developing a more accurate representation of an environment, we believe the accuracy of our system will be improved. However, having an array of mobile phones acting as an eye tracking system presents several challenging issues and questions in regards to limitations. One inevitable issue involves the overall aggregation of data passed from each phone, followed by displaying the data in real time. These are all concerns that we are keeping in mind as we continue forward with our project.

4. Results

We were able to develop a system that can track a user’s eyes in real time. Our methodology still requires quite a bit of refinement. The initial phase of this project returned some interesting results as we noticed many differences between sample images. We noted that there were varying results when processing an image that contained peripherals (in this case glasses), shown below.
After recognizing this interesting result, we started to analyze the images at different stages of the preprocessing phase. By doing so, we were able to conclude that the reason for inaccuracy stemmed from the reflection of the glasses, along with the contours that the glasses provide. Initially we believed that skin tone would have an affect on the accuracy of the implementation but there wasn’t enough data to make an assumption about skin color as the sample size was too small. Identifying a difference based on skin tone still requires more extensive testing in order to determine a conclusive result. In our current implementation, skin tone has not presented itself as an issue. Generally, when a user is looking forward (front facing) towards the video camera, their skin tone and eye color did not affect the accuracy of the eye tracker.
Figure 7: Front Facing, Multiple Skin Tone Eye Detections

There was also a noticeable difference when tracking an eye in real time versus tracking them on static images. This was very evident once we switched to the real time environment and it should be noted that this is heavily influenced by a person’s head movement between each frame of the process. As a result, the application is trying to account for head movement, in addition to dynamically generating a threshold value for the circle finding algorithm, resulting in varying results from the computer’s camera. Dynamically finding the threshold between each frame of the application is a very obvious limitation of the project and it is one that we intend to remedy. The issues that arrive when tracking in real time could be diminished once we add a calibration phase to the application. This will allow us to determine a person’s threshold once, eliminating the need to execute the threshold finding algorithm on every video frame. Although the algorithm for finding the threshold is $O(\log n)$ on average, it becomes a major concern when executed repeatedly for the duration of the user’s session. Due to those limitation we were able to make some significant progress and develop a more efficient algorithm. We accounted for the constant applying of the threshold generation and averaging over frames as opposed to displaying the data from every frame. This has resulted in a far more accurate algorithm and thus allowed us to start porting the algorithms over to mobile devices. The changes that we made in our implementation has allowed us to create a system that greatly rivals the previous
implementation. Our current methodology even allows for slight movements in the user’s head. Hopefully the changes create an environment where the precision is close enough to account for slight movements in the user’s eyes.

For further refinement, we want to improve the performance on mobile devices. Currently there are very large gaps in the implementation between the two different mobile platforms. Each platform provides different challenges that must be overcome so that the algorithms can be as accurate on the desktop application as the mobile applications. We also hope to possibly add a backend service that can complement the mobile implementation. That will allow for reduced load on the mobile devices, which will keep battery from being burned up and avoid possible damage to the phone. We also have yet to employ the mobile array of phones and are unaware of the tribulations that come with it, but it is obvious that it would not be easy. At that point the algorithm can hopefully account for some sort of stereoscopic needs. Having a stereoscopic element on top of the current implementation will account for major changes in head movement.

5. Conclusion

Ultimately, we have developed an eye tracking system that operates with high precision within well lit environments. Of course this is compared to our previous implementation. Developing an extensible eye tracking system that works on all types of eyes in any environment will require several further advancements. To achieve a system of this magnitude, we will need to find a more efficient alternative to eliminating the use of infrared cameras. Without infrared, our system becomes inefficient as our program tries to account for the lack of reflection on the
cornea. This issue became more apparent as we moved our system to mobile devices, which have more limited computing resources.
REFERENCES


