EyeDROID: Android eye tracking system

Daniel Garcia
IT University of Copenhagen
Copenhagen, Denmark
dgac@itu.dk

Ioannis Sintos
IT University of Copenhagen
Copenhagen, Denmark
isin@itu.dk

ABSTRACT
Current eye tracking systems usually delegate computational intensive processing to a remote or local server, thus reducing the mobility of the user. With the emergence of mobile and wearable devices a new possibility for eye tracking has appeared alongside. However, implementing an eye tracking system on such devices implies new challenges not currently present on the ones implemented on stationary machines, such as mobility and limited available resources. This paper presents EyeDroid, an Android platform video-based head mounted eye tracker. Unlike other eye tracking systems, EyeDroid performs all its processing workload in a mobile device and sends the resulting coordinates of the incoming video streaming to a network client. The system was evaluated in therms of speed, energy consumption and accuracy. The results were a processing frame rate of 6.25fps that could be improved by replacing the camera driver, a battery lifetime of approximate 4.5 hours and an accuracy of 90.88%. For this reason, it can be concluded that EyeDroid provides an efficient solution for mobility issues present in current eye-tracking systems, therefore it could be used along mobile and wearable devices.

Author Keywords
EyeDroid; Eye tracking; Android; OpenCV; JLPF; ITU;

ACM Classification Keywords
Human-centered computing: Ubiquitous and mobile computing; Computing methodologies: Computer graphics

General Terms
Algorithms; Design; Experimentation; Measurement; Performance

1. INTRODUCTION

Due to the emergence of everyday life wearable and mobile devices, novel system interaction techniques are required, suchlike eye tracking. However, this techniques face some challenges when trying to achieve the primary goal of such devices, which is mobility. Mobility can be seriously impacted when used interaction techniques require heavy processing done in devices that are not powerful enough, either in terms of computation capabilities or battery life, meaning that the device would need to be connected to an external machine to delegate the computational intensive tasks or because batteries need to be recharged. According to the cyber foraging scenario, even when using wireless connections the mobility is limited to its range.

Eye tracking has been studied widely in the past years and applied to different fields, such as assisting technologies and augmented reality, between others. By extending or replacing a system's input with an eye tracking interface, new possibilities arise to improve the users experience.[5] For instance, gaze tracking data can be used to explicitly control the cursor in a mobile handheld device in a more natural way [2] or to implicitly recognize an activity the user is performing

As mentioned above, eye tracking techniques can be used along mobile and wearable devices, but current mobile eye trackers need a remote or local server to perform the image processing, thus reducing the mobility of the user. Because even if a wireless technology is used to transfer the eye images to a server, the volatility of the network and limited battery life are still challenging. For this reason, analyzing the eye images on a mobile device that can be carried by the user can be a big advantage of mobile gaze tracking systems.

1.1 Eye tracking challenges on mobile devices

The main challenge when implementing systems on mobile and wearable devices is the need of developing low resource consuming algorithms that can be executed fast enough and have an acceptable accuracy level. Such is the case of eye tracking, which requires image analysis techniques that are computationally intensive and therefore it is difficult to perform this kind of processing on a device itself due to the lack of optimal resources, such as limited battery life, processing power and network capabilities. For this reason, efficient algorithms able to perform the required image analysis are needed in order to avoid delegating computational workload to an external computer, as is usually done in current eye tracking systems.

1.2 EyeDroid

EyeDroid is a mobile Android platform eye tracking system designed to be used with a head mounted camera. EyeDroid receives video streaming from the user's eye as input, process it and sends the resulting 2-axis coordinates to a networked client. Process described in figure 1.

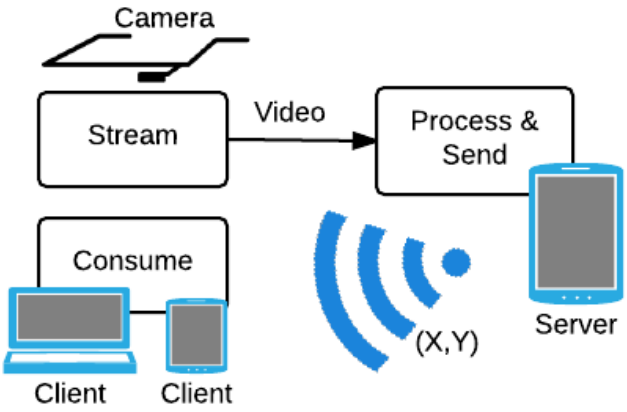

Figure 1. EyeDroid system architecture. Video is streamed from the head mounted camera to the mobile phone where the frames are processed. The resulting pupil coordinates are sent to the connected clients over the network.

Unlike other eye tracking systems which use a stationary processing server, EyeDroid performs all its processing workload in a mobile device and sends the resulting coordinates to a network client. For this reason, EyeDroid supports mobility when used along wearable and mobile devices.

1.3 Paper Overview

The remainder of this paper is organized as follows. Next section summarizes previous studies on eye tracking and mobile image processing. Section 3 describes the main challenges on doing video-based eye tracking on mobile devices and the methodology followed during the process, starting by designing a low resource consumption processing framework which allowed to decompose the image processing algorithm into different steps that can be executed in parallel, following with the optimization of the pupil detection algorithm and finally its evaluation. The proposed system is then introduced in section 4. In section 5 an evaluation is done by comparing execution speed on different algorithm configurations, comparing mobile device battery usage against popular applications and measuring the accuracy of the system. In section 6 a discussion is done pointing out advantages and disadvantages of the proposed solution, potential improvements and future work. Finally, the conclusion and results are presented in section 7.

2. RELATED WORK

2.1 Eye tracking

Depending on the available technology, two basic types of eye tracking systems can be distinguished, electro-oculography and video-based. Electro-oculography can be less unobtrusive because small electrodes can be positioned around the eye of the user, but specialized hardware is needed. In contrast, video based techniques can be used even along regular cameras that can be placed either close to user for remote recording or head mounted. Eye tracking using a video camera has been extensively studied in the literature, particularly in the field of Human-Computer Interaction (HCI). Some example implementations are discussed bellow.

2.1.1 Stationary tracking systems

Sewell [13] presented a system for real-time gaze tracking using a standard remote web-cam without the need for hardware modification. Additionally, it describes the methodology used for pupil detection, which relies first on user's face detection, eye region image cropping and finally, pupil detection. Because a regular web-cam image resolution and/or appropriate lighting to find the pupil might be inadequate, an extra computation to determine the gaze using neural networks was used. Even tough this approach resulted to be very accurate compared to other gaze tracking systems, could be implemented using non-specialized hardware and be unobtrusive, it needed extensive calibration and heavy computation. For these reasons, EyeDroid, which also uses a low-cost camera, was implemented following a mobile approach.

Other eye tracking systems have been developed, such as MobiGaze [10]. This research project tried to provide a new way of interaction with the mobile device by building a remote eye tracker. By attaching a stereo camera on a handheld device, it was able to extract the position of the gaze and use this information as input to the device.

2.1.2 Mobile tracking systems

A wide variety of mobile trackers have been developed before, such is the case of the open source Haytham project [8] gaze tracking system. The technique used on this software to detect the pupil is based on predicting a region of Interest (ROI), applying image filters, removing outliers and blob detection. A similar pupil detection technique to Haytham project was used along some optimizations for EyeDroid implementation.

Though its intrusiveness, head mounted eye trackers provide higher accuracy than remote trackers and can support user mobility, such as the case of Kibitzer [1], a wearable gaze-based urban exploration system. Kibitzer used computer vision techniques to track the eye of the user. It suggests the usage of a head mounted camera in a bike helmet, along with an Android mobile device and backpack-held laptop. First, the camera sends the captured image to the processing laptop via USB cable, afterwards the computer sends the eye data to the mobile client through a socket-based API. In a similar way, the openEyes eye tracker [7] is proposed. Their solution provides both a head mounted camera and a set of open-source software tools to support eye tracking. OpenEyes was intended to be mobile, therefore the processing unit was carried on a backpack. However, in both head mounted scenarios unobtrusiveness level is low due to the size of the processing units carried in the back of the user.

2.2 Image processing on mobile devices

Even an algorithm designed to solve a specific problem, such as eye detection, requires high computational resources. To meet the constraints of the computational budget provided by mobile devices, developers either trade-off on quality or invest more time into optimizing the code for specific hardware architectures. For this reason, existent technologies have been optimized to support computer vision techniques on mobile

devices. Such is the case of OpenCV library [11] which provides GPU acceleration for low-level image-processing functions and high-level algorithms. [12]

Several applications have been successfully implemented using the OpenCV library along with the Android native development kit (NDK) and proved to work efficiently in mobile devices, such as the face recognition for smart photo sharing research project [14] and PicoLife, a computer vision-based gesture recognition and 3D Gaming system for Android Mobile Devices [9]. Similar to this mentioned projects, OpenCV was used on EyeDroid system to provide real-time video processing.

The most common approaches followed by the mentioned systems involved video-based solutions using OpenCV along mobile trackers because of the potentially high accuracy that can be obtained with low calibration and the minimization of resource consumption. As mobile and wearable devices can now be equipped with cameras and more computing power, thus the demand for computer-vision applications is increasing, such as eye tracking. Even though several mobile eye tracking systems have been developed, no solution provides truly mobility to the user as EyeDroid.

3. METHOD

Given the challenges of video-based eye tracking systems and its usage along with mobile and wearable devices, such as limited battery life, processing power, network capabilities and mobility, this project followed an iterative incremental process with four iterations to overcome them.

The first iteration consisted on designing a low resource consumption processing core that allowed the decomposition of the image processing algorithm for eye tracking used in Haytham project into several steps that could be run in parallel. This processing core was also designed to be a platform independent external library to encourage future portability.

In the second iteration the processing core was implemented and tested, resulting in the Java Lightweight Processing Framework (JLPF) [3]. At the end of this iteration, JLPF was imported to the Android platform. A first prototype was build using a mock processing algorithm.

As part of the third iteration, the image processing algorithm for eye tracking was implemented using the Android native support for C++ on top of the system core (JLPF). The algorithm was decomposed into several steps, allowing different parallel execution configurations.

The forth and final iteration consisted on the algorithm execution configuration selection and the system evaluation according to speed, battery consumption and accuracy.

4. EYEDROID EYE TRACKER

EyeDroid is an Android platform eye tracker which is intended to be used along with a head mounted camera. Because of its low resource consumption, a smartphone can be used as hosting device, offering a truly mobile solution as all the required equipment for the system can be carried by the user. Unlike other eye tracking systems, all the image processing is done on the device itself without the need of delegating the task to an external processing server. The input to the system consists of real time video of one of the user's eyes provided by a camera that is directly connected to the device. The output of the system is sent to any TCP/IP client that can consume the produced pupil coordinates. EyeDroid eye tracker can be seen in figure 2.

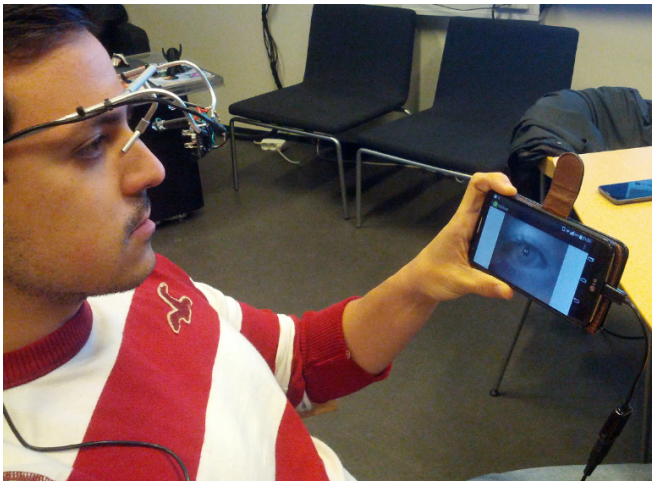

Figure 2. EyeDroid physical hardware.

4.1 Design decisions

Below, the most important decisions during design and implementation are presented.

- Implement an independent processing framework as basis. The Java Lightweight Processing Framework (JLPF) was designed to be the core of the final system, independent from any image processing process and the Android platform. As a consequence, the core could be implemented and tested separately. Additionally, the actual eye tracking system was built on top of the processing core, providing portability for future implementations.
- Use an architectural framework that allowed experimentation on the image processing algorithm. The Pipes and filters architectural framework was used on the processing core in order to support flexible experimentation on the image processing algorithm. This approach allowed the eye tracking algorithm to be decomposed into several steps (filters) connected by pipes to define the execution order, ran and tested under different execution configurations, both sequential and parallel, until the optimal was found. The steps on which the algorithm was decomposed and the parallel execution policies were decided based on experimentation. Once a prototype was build, both the algorithm and its scheduling policy were tuned up until the most efficient configuration was found.
- Passive consumer-producer pattern on architecture pipes and filters interaction. Filters consume frames to be processed from the pipes. In order to achieve lower resource consumption, passive consumer implementations of the architectural filters were implemented and made them

to fit a variety of scheduling execution policies, such as the sequential execution of the filters on a single thread and the parallel execution of the filters on many threads. The producer-consumer pattern reduces the computational overhead produced by active consumers and simplifies workload management by decoupling filters that may produce or consume data at variable rates.[4]

- Transform Haytham algorithm into a parallel execution. Since performance was a key issue for the system, a sequential implementation of the algorithm had a throughput penalty, specially when regular video frame ratio had to be processed.

- Most recently computed frame for region of interest (ROI) prediction. The algorithm uses feedback from the most recently processed frame to predict the ROI around the eye, on which the pupil is more likely to be found, for subsequent frames. As described bellow in the subsection *Image processing algorithm*, the feedback from frame N does not necessarily affect frame N+1 because the gap between frames feedback can be of more than one when several frames are being processed in parallel. This accuracy issue was considered as acceptable since the maximum difference between two frames can be configured by setting a constant pipe capacity size.

  When executing in parallel, even though each step could potentially run on a different thread, there is no deterministic execution of the individual steps which can lead to erroneous feedback.

- Android NDK usage. Android native development kit (NDK) was used for implementing the image processing algorithm instead of the regular Android SDK because of its performance boost.

- Generic use of input and output to the processing core. This decision made it easy for the processing core to interact with different kind of inputs and outputs. For instance, it is transparent to the core whether frames are provided by the network, a file or a camera. This approach facilitates testing, evaluation and future implementations.

4.2 Hardware

The hardware requirements in the current implementation of the EyeDroid eye tracker are an Android mobile device (minimum API level 15) and a head mounted USB 2.0 infrared camera connected directly to the phone. EyeDroid hardware is shown in figure 3. The recommended camera resolution is 640x480px. Because the Android platform does not provide support to connect an external USB camera, the OS needs to own root access to the phone and use customized camera video drivers. On EyeDroid, open source third party drivers were used.[6]

Because the number of people already owing a smartphone is large and the rest of the hardware needed is a USB camera and a simple head support, the system could be potentially used to support every day tasks.

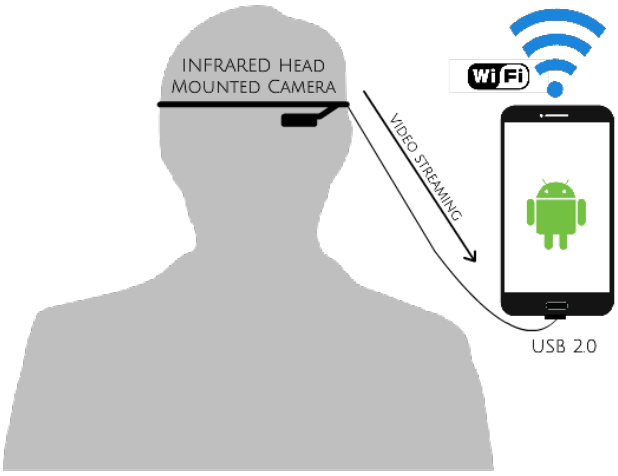

Figure 3. EyeDroid hardware. An infrared USB 2.0 camera is connected to the android device. This mobile server publish the resulting coordinates through a Wi-Fi connection.

4.3 Software

4.3.1 Java Lightweight Processing Framework (JLPF)

According to the design decisions, pipes and filters design pattern (or pipeline) was used as the main architectural framework. Since variable decomposition was needed to test different algorithm configurations in order to be optimized, this design provided flexibility to experiment with different parameters to customize the processing steps required to perform eye tracking.

The Java Lightweight Processing Framework(JLPF) was built as an external library in the first iteration of the development process with the notion in mind that it should be platform independent and be able to perform any kind of processing and not just image processing. The idea behind the design was to decouple as much as possible the whole algorithm and it's scheduling execution policy. Since the target platform is Android running on a mobile device performance was a key issue. This design allowed for a fully configurable algorithm in terms of decomposability of the steps and how these steps should be scheduled for execution on the available processing resources, instead of a monolithic algorithm that would perform poorly. Finally, in order to divide the algorithm in steps of equal execution time, the composite pattern was implemented to allow composition of individual steps.

The software architecture of the JLPF can be seen in figure 4. The component can be reused for any kind of processing just by implementing the IOController class for a specific platform, the IOProtocolReader and IOProtocolWriter to specify how to read a frame and how to return the result respectively.

4.3.2 Image processing algorithm

Since performance is important due to the lack of available resources (compared to a stationary eye tracking system), an important decision was to use the Android NDK support for C++ instead of the regular Android SDK for java. This decision allowed the algorithm code to run directly on the processing resources and access system libraries directly, unlike Java which would run on a virtual machine. Moreover, this

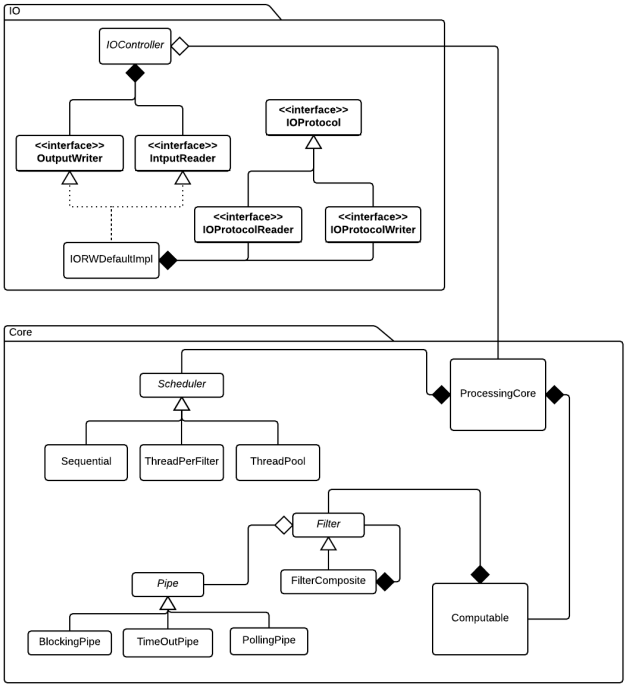

Figure 4. Java Lightweight Processing Framework (JLPF) software architecture.

allowed for independent development and testing of the processing algorithm that was later imported to the main Android application.

For the image processing the OpenCV library was used. Below are listed the individual steps of the actual algorithm, how they were composed and scheduled in order to optimize the algorithm execution (figure 5). It should be noted that each frame passes through all the steps in the exact same order as it was originally provided.

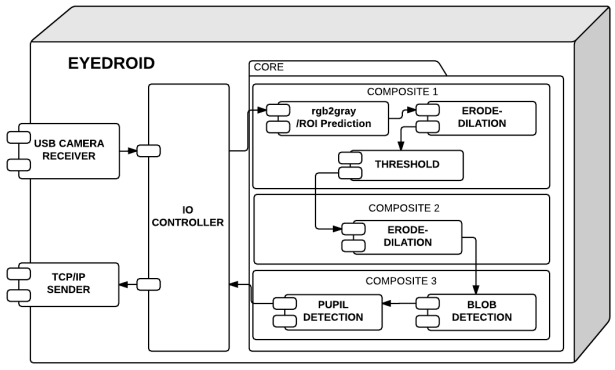

Figure 5. EyeDroid software components. Eye tracking algorithm inside the core is decomposed into steps (filters) and connected by pipes (arrows). Each composite is executed on a separate thread.

1. Eye Region of interest(ROI). The first time a frame is received, a constant ROI is defined in the center of the image (400x350 px), covering the whole eye of the user. This region is used to look for the existence of the user pupil on a smaller image than the original one in order to minimize the processing overhead. This constant initial ROI is later reduced (200x150px) and moved closer to a previously position where the pupil was found than the center of the image. Some eye tracking systems, such as Haytham project, uses an immediate previous frame on the streaming sequence (in case the pupil was found) to define new ROI's to increase accuracy. Following this approach, a sequential algorithm execution is needed because each frame depends on the processing completion of the exactly previous one.

   This paper now proposes a simple technique to estimate subsequent ROI's that allows the parallelization of the algorithm. Once the pupil is found on previous processed frames, the ROI is reduced and moved to the most recently computed pupil coordinates (the last frame which it's processing is completed) and not the position where the pupil is in the immediate previous frame. This technique was adopted based on the assumption that if the algorithm execution is fast enough, even when processing different frames in parallel, the ROI of the current frame being processed is very similar to the most recently computed one. In case that previous coordinates has not been computed yet, the default constant ROI is used again. Because each frame is now independent from the previous one, parallelization of the algorithm steps can be done. Finally, if the pupil is not found then the default constant ROI is used until an appearance of the pupil is detected again. A brief example is shown in figure 6.

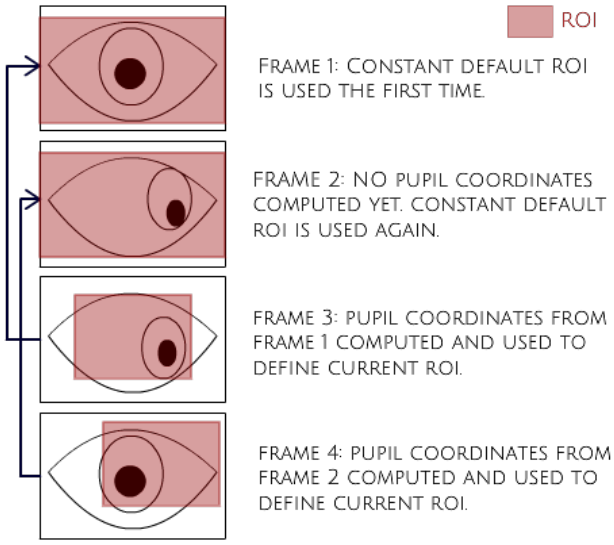

Figure 6. ROI prediction technique based on the most recently computed frame

2. RGB to Grey conversion. The second step of the algorithm converts the original image ROI into gray scale. This reduces the processing overhead as there is one byte per pixel instead of three.

3. Erode-dilation. Edore (3 times) and dilation (2 times) is performed both before and after the threshold step. This step is used in order to smooth the corners in the image blobs.

4. Thresholding. The exact type of thresholding used was binary inverted. The result of this operation was a new image where the most dark parts of the original image were converted to black while the most bright parts of the image were converted to white. This way the pupil is represented now as a black blob in the image while removing any unnecessary data in the rest of the image. The thresholding lower limit value (70) was determined by experimentation, selecting the lowest value that kept the pupil as a black blob. Below this selected value, the pupil would be converted to white pixels along with the rest of the image.

5. Erode-dilation. Edore (3 times) and dilation (2 times) is done in case the thresholding step detected other dark blobs in the image except the pupil. By using erode in the output image any small dark blobs are shrinked until they disappeared. Dilation was used to bring the pupil blob back to its original size. This step was necessary in order to remove blob outliers.

6. Blob detection. After each frame is processed by all the previous steps, the result is a white image with black blobs in it. This makes it easier for the detection method to find circles. The method used in this step is the HoughCircles from the OpenCV library, giving as parameters a minimum radius of 20px and a maximum of 50px.

7. Pupil coordinates selection. Finally, because the previous step can detect many circles, the one that is closest to the center of the image is taken as the pupil and the 2-axis coordinates are computed. The location of the detected pupil is used later as feedback to the first algorithm step in order to compute the ROI on subsequent frames.

In order to fit all these steps in the most efficient way into filters and work along the JLPF library, three composite filters were used. The first one containing the steps 1-4 defined in the previous section, the second one the step 5 and finally, in the third composite, 6 and 7. This composition was chosen after evaluating different execution configurations. Such process is described bellow during the evaluation.

A result comparison of each processing step using the proposed ROI prediction technique against a fixed ROI can be seen in figures 7 and 8. In figure 7 it can be seen that the ROI is smaller, meaning that in a previous frame the pupil was detected. For this reason, the ROI was moved and the image around the eye was cropped to improve confidence on later detections. In figure 8 the default constant ROI is used which means that in previous frames the pupil was not detected yet.

4.3.3 Input/Output

The input to the EyeDroid system is given as video streaming recorded from the user's eye region and is initially converted to a resolution of 640x480px. As output, the resulting 2-axis coordinates from the pupil are sent to a wireless networked client. Even though the current EyeDroid architecture was originally designed to operate along with a USB connected camera, input sources can be given also from a networked source or the cameras installed on the device. Alternative input sources were implemented for future implementations or testing. Since the processing core was decoupled from the input and output implementations, other sources and destinations could be further added. For instance, video streaming could be received from a remote server as input, processed and the result sent to a networked client.

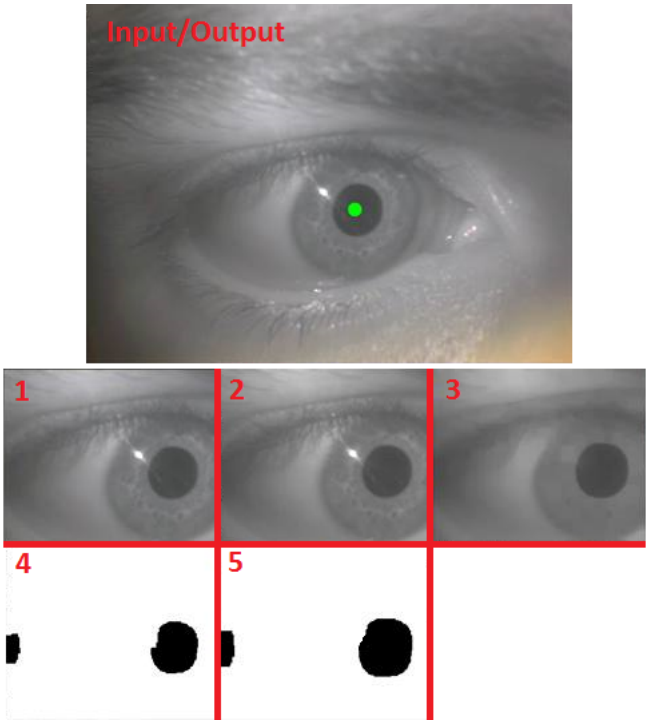

Figure 7. Processing steps using dynamic ROI prediction. The ROI used is smaller than the default one and moves around with the pupil because of a previous pupil detection, making it more confident.

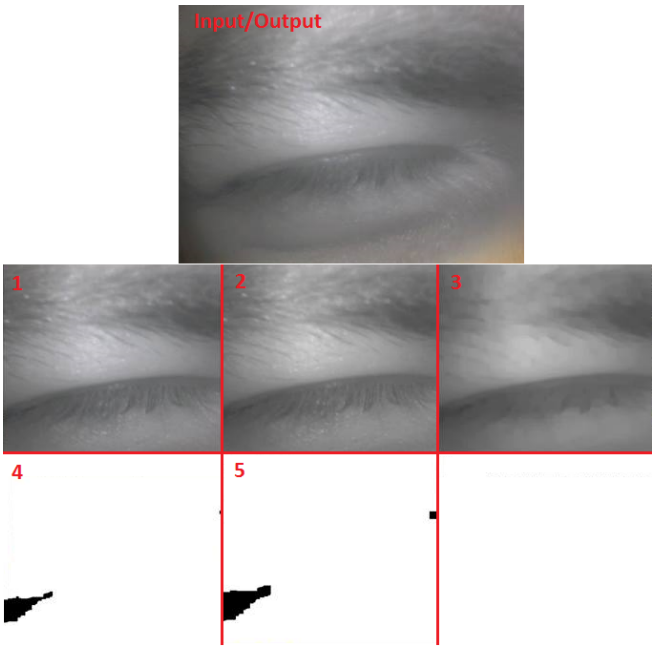

Figure 8. Processing steps using a constant ROI. The ROI used is set using the default position and size because the pupil was not detected in previous frames. This ROI is bigger than the dynamic one.

Step	Execution time (ms)
ROI detection/RGB2Gray	2.359746
Before/Erode-Dilation	22.605388
Threshold	0.761905
After/Erode-Dilation	19.779365
Blob detection	12.087302
Pupil detection	16.969841

Table 1. Algorithm steps averaged execution time per frame.

Composite	Execution time (ms)
Composite 1	25.912467
Composite 2	19.180371
Composite 3	30.477454

Table 2. Compositions averaged execution time per frame. Composite 1: ROI/RGB2Gray, B/Erode-Dilation, Threshold; Composite 2: A/Erode-Dilation; Composite 3: Blob detection, Pupil detection.

4.3.4 Network connectivity

EyeDroid system provides a TCP/IP server-client architecture on which the mobile phone offers server functionality and any other system able to consume the resulting coordinates can connect. Messages from the server to the client are sent in a byte array format containing a message, X-coordinate and Y-coordinate.

5. EVALUATION

EyeDroid was implemented and evaluated on a brand new LG-G2 smartphone with 2 GB RAM, a Quad-core 2.26 GHz Krait 400 processor, an Adreno 330 GPU and running Android 4.4 version. In the next subsections, the evaluation results are presented in therms of speed, energy consumption and accuracy.

5.1 Algorithm execution time

The execution time of the processing algorithm was evaluated under different scheduling policies, including sequential and parallel. At the beginning, the execution time from each algorithm step was measured (table 1), and based on this results, three configurations were evaluated. The first consisted on the sequential execution of all the steps (1 thread), the second consisted on splitting the algorithm into two composites (2 threads) and similarly, a third one using three composites (3 threads). The goal during this experimentation was to balance the workload between threads according to the processing overhead produced by each step. Only two parallel configurations were tested because the execution data collected from each step suggested that these were the best candidates to balance the workload. The results from this evaluation showed that the best configuration was to run three composites in parallel (3 threads). Each individual composite running time is shown in table 2.

Camera	1 composite	2 composites	3 composites
Back	6.75 fps	10.41 fps	14.28 fps
Front	12.19 fps	14.28 fps	16.12 fps
USB	6.17 fps	6.25 fps	6.25 fps

Table 3. EyeDroid processing rate (frames/second). Results from executing the processing algorithm using 1 composition (1 thread), 2 compositions (2 threads), 3 compositions (3 threads) are shown.

Camera	Frame rate
Back	20 fps
Front	20 fps
USB	6.41 fps

Table 4. Frame rate provided by the evaluated cameras. The first two correspond to the phone built-in cameras, and the third one, an external USB camera.

Processing rate results from sequential and parallel executions are shown in table 3. As it can be seen on the results, the processing rate remains around 6.25fps in all its different execution environments. For this reason, further evaluation of the EyeDroid core was done using now the smartphone built-in cameras. These results showed a significant improvement from sequential to parallel executions of the image processing algorithm. One last measurement was performed in order determine the maximum frame rate that could be consumed from the evaluated cameras (table 4) without any image processing performed. Both built-in cameras showed a frame rate of 20fps, meanwhile the external USB camera provided only 6.41fps. As a consequence, it can be concluded that a bottleneck is produced when reading from it and thus limiting the processing capacity of the system. Because third party drivers were used to read from the USB camera, replacing them could improve the overall performance. Finally, an optimistic estimation of 15fps as potential processing capacity is thought, in case the behavioral trend is replicable.

5.2 Energy consumption

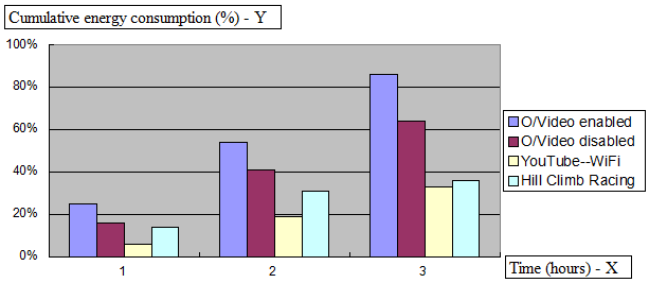

Figure 9. Comparison between EyeDroid and two other popular applications showing cumulative energy consumption (%) per hour

Energy consumption evaluation was done by running EyeDroid for 3 uninterrupted hours and measuring the percentage of energy consumed every hour. Measurements were given by the Android built-in battery level indicator. It is true that this indicator is not as accurate as a measuring using a battery meter but such device was unavailable. In order to compensate the inaccuracy of the default indicator, the device was fully charged before conducting each experiment, any other apps running were closed but default Android services, and the

	Frames
Total	812
Right detections	738
Erroneous detections	74
	Percentage
Accuracy	90.88 %

Table 5. EyeDroid Accuracy. Right detections were counted when either the pupil was present and detected or not present, therefore not detected. Erroneous detections were counted when either the previous conditions were not satisfied or a detection was an outlier.

brightness of the screen minimized. The only user interaction performed was during checking the energy consumption every hour. Each measurement was repeated three times and results were averaged.

Since EyeDroid can optionally show the resulting coordinates drawn in top of the input video streamed on the device display, two different experiments were made. One with the result preview enabled and one disabled. Finally, two other popular applications were measured in the same way in order to provide a context, YouTube video streaming and Hill Climbing racing game. The results suggest that EyeDroid behaves similar to Hill Climbing game but deviating approximately 10% per hour. The maximum battery life time estimation running EyeDroid with result preview disabled is of approximate 4.5 hours. Results are shown in figure 9.

5.3 Accuracy

A sample video was captured from EyeDroid output and measured regarding how many of the total frames detected the pupil correctly, either when the pupil was present or not, and which of those detections were outliers. The video recorded from the eye of the user contained all kind of movements, blinks and complete eye closing. Results are shown in table 5. Most erroneous detections were done when the pupil reached extreme side positions or due to fast eye movements and blinking.

Because the scope of the project was not to develop a reliable eye tracker, but instead to provide a suitable environment for its execution on a mobile device, only simple accuracy evaluation was done over a set of frames.

6. DISCUSSION

6.1 ROI Prediction

As stated above, the ROI prediction was based on the assumption that if the algorithm parallel execution is fast enough, the ROI of the current frame being processed is similar to the most recently computed. For this reason, certain error degree was considered as acceptable. This assumption limits EyeDroid to be run in a reasonably powerful device, otherwise the accuracy of the system could be reduced. However, mobile devices are continuously evolving and becoming more powerful, therefore EyeDroid performance should improve with new generations of smartphones.

6.2 Mobile USB camera

Although a USB connected camera consumes a considerable amount of energy from the mobile device, it allowed to position the camera close to the user's eye and record only the region of interest. This reduced the processing needed compared to stationary systems, and in case calibration was done, it would avoid complex calibration techniques. Moreover, this approach increases accuracy even when no high resolution camera is used.

6.3 Further evaluation

As mentioned before, the default frame size used for evaluation was 640x480. Further evaluation could also be done with a smaller frame size in order to reduce processing overhead. Although an accuracy impact might occur by reducing the frame size, EyeDroid could potentially run faster and consume less resources.

Because evaluation was performed only using one kind of device, results might vary between hardware. It would also be possible that the suggested execution policy is not the best for other devices or platforms, but because of the EyeDroid architectural design, different algorithm configurations could be set to meet such specific hardware requirements.

6.4 Future work

As described in the evaluation section, replacing the current USB camera driver used on EyeDroid could significantly improve its performance. For this reason, an efficient driver implementation that can consume a greater number of frames from an external camera is needed.

In the current implementation there is no filtering performed on the coordinates produced by the algorithm. In order to provide more accurate data and reduce networking overhead, a new filter could be implemented and added at the end of the process to detect outlier coordinates and discard them.

Because calibration was not inside the project scope, it was not performed. As a consequence, the produced coordinates are relative to the recorded frame. In future implementation, a simple calibration technique could be provided in order to produce meaningful coordinates.

Although estimating Z-index distances would be inaccurate because monocular tracking is only supported by EyeDroid, a possible extension would be to transform the algorithm to be able to perform remote eye tracking from the front camera of the mobile device. As a prototype, the modifications needed are two more filters at the beginning of the current filters chain. The first one to perform face detection and the second one to perform eye detection on the detected face, if any. This way, the ROI of the eye could be now passed to the current filters and perform the same processing. Another variation of the current algorithm could be implemented to detect pupil dilation changes.

Due to EyeDroid's extra functionality to stream video from an IP camera, an alternative to using a USB camera could be to use wireless connectivity. Although, the extra networking might slow the algorithm down, this alternative could potentially decrease the energy consumption on the device, and

secondly, increase mobility of the user.

Finally, EyeDroid was implemented as a regular application. In the future it could be implemented as an Android service that runs in the background. This way clients that want to make use of EyeDroid can connect to it at any time and consume coordinates, and not only when the application is active. Additionally, the screen of the device could be turned off in order to reduce the energy consumption.

7. CONCLUSION

This paper proposes EyeDroid, an Android platform mobile eye tracker that supports mobility of the user. To accomplish this, EyeDroid records video from the eye of the user using a head mounted camera and process the frames in a mobile device to determine the pupil coordinates. Additionally, clients can connect via wireless to consume such coordinates. EyeDroid is implemented using pipes and filters architectural pattern, where the image processing algorithm is divided into steps and represented as filters that can be later grouped to create compositions. This way, filters and compositions can be executed in parallel. Because parallel execution was intended but the algorithm's nature is sequential, a technique to estimate regions of interest that allows parallelization by considering acceptable low error was presented. EyeDroid algorithm steps were grouped to form three composites and run in three threads, which resulted to be the best configuration after the evaluation. Even tough optimal performance was not achieved due to a frame rate bottleneck produced by the USB camera third party driver used, results are satisfactory. We conclude that EyeDroid overcomes the mobility issues of current eye tracking systems and for this reason, it could be used along mobile and wearable devices.

ACKNOWLEDGMENTS

We thank IT University of Copenhagen staff who wrote and provided helpful comments on previous versions of this document and the EYEINFO research team who provided the Haytham project source code.

REFERENCES

1. Baldauf, M., Fröhlich, P., and Hutter, S. Kibitzer: a wearable system for eye-gaze-based mobile urban exploration. In *Proceedings of the 1st Augmented Human International Conference*, ACM (2010), 9.
2. Drewes, H., De Luca, A., and Schmidt, A. Eye-gaze interaction for mobile phones. In *Proceedings of the 4th international conference on mobile technology, applications, and systems and the 1st international symposium on Computer human interaction in mobile technology*, ACM (2007), 364–371.
3. Garcia, D., and Sintos, I. Java lightweight processing framework (jlpf). **`https://github.com/centosGit/JLPF`**, 2014.
4. Göetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., and Lea, D. *Java concurrency in practice*. Addison-Wesley, 2006.
5. Jacob, R. J., and Karn, K. S. Eye tracking in human-computer interaction and usability research: Ready to deliver the promises. *Mind 2*, 3 (2003), 4.
6. Lab, K. Usage of usb webcam with customized galaxy nexus (android 4.0.3). **`http://brain.cc.kogakuin.ac.jp/research/usb-e.html`**.
7. Li, D., Babcock, J., and Parkhurst, D. J. openeyes: a low-cost head-mounted eye-tracking solution. In *Proceedings of the 2006 symposium on Eye tracking research & applications*, ACM (2006), 95–100.
8. Mardanbegi, D. Haytham gaze tracker. **`http://eyeinfo.itu.dk/index.php/projects/low-cost-gaze-tracking`**, 2014.
9. Mariappan, M. B., Guo, X., and Prabhakaran, B. Picolife: A computer vision-based gesture recognition and 3d gaming system for android mobile devices. In *Multimedia (ISM), 2011 IEEE International Symposium on*, IEEE (2011), 19–26.
10. Nagamatsu, T., Yamamoto, M., and Sato, H. Mobigaze: Development of a gaze interface for handheld mobile devices. In *CHI'10 Extended Abstracts on Human Factors in Computing Systems*, ACM (2010), 3349–3354.
11. OpenCV. Opencv for android. **`http://opencv.org/platforms/android.html`**.
12. Pulli, K., Baksheev, A., Kornyakov, K., and Eruhimov, V. Real-time computer vision with opencv. *Communications of the ACM 55*, 6 (2012), 61–69.
13. Sewell, W., and Komogortsev, O. Real-time eye gaze tracking with an unmodified commodity webcam employing a neural network. In *CHI'10 Extended Abstracts on Human Factors in Computing Systems*, ACM (2010), 3739–3744.
14. Vazquez-Fernandez, E., Garcia-Pardo, H., Gonzalez-Jimenez, D., and Perez-Freire, L. Built-in face recognition for smart photo sharing in mobile devices. In *Multimedia and Expo (ICME), 2011 IEEE International Conference on*, IEEE (2011), 1–4.