Marc Aurel Kastner

Supervisor Michael Stengel

Referee Prof. Dr. Ing. Marcus Magnor

Robust binocular pupil detection for a mobile computing platform

Project Thesis

August 25, 2015

Computer Graphics Lab, TU Braunschweig

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Braunschweig, 25. August 2015

Marc Aurel Kastner

Zusammenfassung

Applikationen für Augmented Reality und mobiles Virtual Reality (VR) erfordern mobiles Pupil Tracking in Head-mounted Displays (HMD). Das ultimative Ziel ist die Erhöhung von Immersion in VR Anwendungen. Bisher existieren für diese Zwecke jedoch noch keine praktikablen Lösungen. Insbesondere gehen hohe Kosten für den Aufbau, die unterliegende Plattform und den benutzten Kameras, einher.

Diese Projektarbeit präsentiert die Umsetzung eines Pupillenerkennungs-Algorithmus auf einer handelsüblichen mobilen Plattform. In Verbindung mit einer kostengünstigen Kamera stellt das System eine praktikable Lösung für mobiles Eye Tracking dar. Der Algorithmus wurde auf die eingeschränkten Hardwarebedingungen der Zielplattform angepasst. Für die mobile Nutzung wurden Funktionen zum Versand der Ergebnisse über WLAN hinzugefügt. Um die Robustheit des Systems zu optimieren, wurde der Algorithmus stückweise erweitert und durch neuen Herangehensweisen verbessert. Alle Ergebnisse wurden ausführlich getestet und evaluiert. In der abschließenden Diskussionen wird auf offene Probleme und das Potential für zukünftige Arbeiten eingegangen.

Abstract

Applications of Augmented Reality and mobile virtual reality (VR) need mobile pupil tracking within Head-mounted displays (HMD). The ultimate goal is an increase of immersion. However, there are no feasible solutions for this available. In particular, they often come with increased costs for the underlying system, the cameras and its surrounding setup.

This thesis provides an implementation of a pupil detection algorithm ported to a mobile consumer platform. Attached to an affordable camera, the system is able to simulate the functionality of a pupil tracker. Using this is a step to a self-built cheap alternative to expensive HMD setups. The algorithm was ported to limited computing constraints of the target platform. A function to perform remotely over network was introduced, sending results and calibration data via Wi-Fi. To further enhance the robustness of the system, parts of the algorithm were modified and new approaches were tested. All results have been extensively evaluated and compared with the non-optimized algorithm. In the discussion, open problems are analyzed for future work.

Contents

Chapter 1

Introduction

Virtual reality (VR) systems are a comparatively old idea trying to increase immersion and realism for video games, movies and other 3D computer graphics appliances. The first applications using technologies similar to VR date back to the 50s. These approaches however have almost nothing in common with modern trends in terms of size and efficiency. In recent years, research in this field improved and therefore VR applications flourished, making development of the first consumer products using these technologies possible.

Head-mounted displays (HMD), such as the Oculus Rift[Fac15] VR system, are an approach to achieve additional immersion. A well working execution is a big goal for video games, but can also be interesting for interactive movies or other applications. A single step to further vastly increase the immersion of such systems is eye-tracking, or more specifically pupil-tracking. Using advanced techniques, pupil movements can be projected into the applications. While it is possible to implement such a system, it often comes with a high cost for its cameras and surrounding setup. Furthermore, a robust detection uses processing power, which decreases the amount available for other parts of the system.

On the software-side, detecting the pupil center is one of the most challenging problems in the field of computer vision. While a near pixel perfect computation is crucial for VR, algorithms capable of this often are not suitable for mobile purposes due to hardware limitations.

This thesis provides a foundation for a future mobile pupil detection platform. Based on a working approach for pupil detection, a modified version enhanced for mobile computing is developed. Using a low-cost single-board computer and affordable cameras, the system gets evaluated and tested. The ultimate goal is an affordable and self-built solution based on consumer hardware. In future work, the platform can be integrated in head-mounted displays.

A possible use-case for the system is mobile gaming. The mobile platform

can send pupil positions in form of coordinates to a client via network. This client, e.g. a video game running on a smartphone, can directly use the data without the need of further processing needed. This is a major difference to other frameworks. These often need the processing power of a desktop computer connected to the pupil detection device.

The client using pupil information, and the pupil-tracking-device itself are working separately. Therefore, any device which is able to receive network packages would be to use the pupil information without further changes to its system. This includes smartphones, handhelds, tablets and other devices which can run interactive applications.

After this introduction, chapter 2 will outline the environment, this project is built on. First, an overview of the algorithm is presented. Second, a suitable mobile platform fitting the project goals is determined and described. Afterwards, chapter 3 will extensively describe implementation changes to the base algorithm in terms of porting the algorithm to a mobile platform. The algorithm was ported to a new architecture, faced with stricter limitations in processing power. Afterwards, networking communication was included to allow sending results to a remote client. An evaluation regarding the performance, and analysis of platform limitations concludes this chapter. Chapter 4 adds new approaches for making the algorithm more robust. Different techniques and ideas have been tested, implemented and evaluated to see, whether they improve the overall accurateness of the system. Lastly, chapter 5 will conclude the thesis by giving an overview of project results and a prospect to open questions regarding this project.

Chapter 2

Environment

With the goal of developing a mobile pupil detection system, there is a need of a mobile platform, as well as a robust algorithm.

Mobile platforms generally focus on being lightweight and efficient in power consumption, due to restrictions in battery size. Therefore, there are usually heavy limitations on processing power. Hence, a pupil detection algorithm needs to be efficient enough to process its data in such embedded conditions, while also being robust enough for the future use cases of this system.

In Section 2.1, a suitable approach for pupil detection is described. Afterwards, 2.2 explains possible choices of mobile embedded systems for the purpose of building a mobile pupil detection system.

2.1 Pupil detection algorithm

There are multiple ways to approach computer vision and problems surrounding pupil and eye detection. A well-working approach is based on the algorithm developed in a master thesis by Grogorick[Gro15]. Figure 2.1 shows the basic structure of his approach. It uses pictures of an Infrared-exposed eye as shown in Figure 2.2 for its calculations.

In a pre-calculated step, the lens is masked out. This is necessary for the eye only being in the center of calculations. The algorithm often works with histograms over grey-scale data. Therefore, any dark objects not relating to the eye or pupil would influence the robustness of its calculations.

The pupil position is approximated by looking for the darkest area in the picture. This will decrease the area, where future steps of the algorithm need to search for visual features. The result is used in the pupil visibility test, which relies on a simple thresholding. It determines whether the previously approximated area has a chance of having the pupil in it. If it fails, the algorithm can stop, as it won't be able to find a pupil in the next steps.

Afterwards, the occlusion test is an advanced two-step metric, which

decides whether the pupil is clear or occluded. Occlusion can occur in case of overlaying eye lashes or heavy reflections. For the purpose of this project, the functionality in this step is not as crucial. In fact, it will be skipped for performance reasons, as later described in chapter 3.

For a final determination of the pupil, the algorithm distinguishes between two different methods, depending whether the pupil is occluded or clear.

The part for occluded pupil detection tries to reconstruct the pupil, even if major parts of the pupil are not in the visible frame. This is the case for bad angles or if the user is partly blinking. As it is skipped in the modified version of the algorithm, it won't be explained in detail.

The other part of the algorithm designated to clear pupil detection will first filter out reflections and noise. Afterwards, there is a thresholding to further determine the darkest area in the frame. The OpenCV command *findContours* will search for a connected black polygon within the resulting image. Resulting points are then mapped to an ellipse. The center point of the ellipse is reported as the pupil position.

It is important to notice, that the algorithm has some major limitations, when it comes to overly dark eye lashes. This is a problem when people use mascara.

2.2 Platform

When choosing a suitable platform for this project, it is important to evaluate options based on various target goals. In a mobile context with no attached cables, a system which is capable of running by battery power is advisable. Furthermore, the weight of the setup needs to be reasonable.

Due to this, embedded systems or single-board computers are the only suitable candidates. One or two consumer cameras need to be connected via USB or equivalent to input video data. Additionally, the algorithm is based on OpenCV. Therefore, a full operating system with USB drivers and a full development environment is almost necessary. Consumer level hardware and being affordable also excludes self-built hardware solutions. Hence, embedded systems seem not to be a feasible option.

2.2.1 Raspberry Pi 2

For the purpose of porting the algorithm to a mobile system, the Raspberry Pi platform turned out to be a suitable match. In a mobile context, the target is to minimize processing power, power usage and cost of the platform.

The Raspberry Pi platform[Fou15] (Figure 2.3) is a single board computer in the size of a credit card. It got popular for various home-built solutions such as home servers, media players, but is also used also in education. Due to a low cost, abandoning unnecessary extras and being pro-

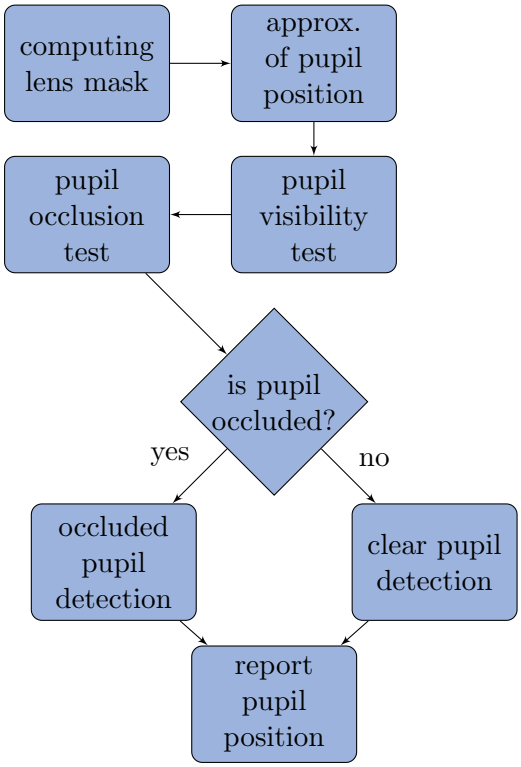


Figure 2.1: Flowchart of the pupil detection algorithm

duced in high numbers, it got a cheap solution for various mobile computing purposes.

The current generation *Pi 2 Board B* features a 900 MHz quad-core ARM Cortex-A7 processor and 1 GB of RAM. It supports 4 USB 2.0 devices, which is enough for two external webcams and a Wi-Fi dongle. An ethernet port can be used for testing. The system is powered by a 5V micro USB port. Therefore, combining it with mobile batteries suited for smartphones or similar is possible. The platform can run any ARMv7 compatible Linux distribution. Beside Raspberrys in-house solution *Raspbian* based on Debian Linux, there are various other options. This project has been developed on top of ArchLinux ARM. It is a rolling distribution for ARM platforms based on the popular ArchLinux. Compared to Raspbian, there is a higher amount of installable precompiled packages, such as a binary version of OpenCV.

For video capturing up to two PlayStation (PS) Eye webcams were used. The PS Eye cameras can capture up to 120 frames per second at a resolution up to 640x480. The raw data has a YUYV format. It is shipped with an IR filter which can be removed by opening its case. It has a reasonable low latency, which makes it a good candidate for affordable eye-tracking applications.

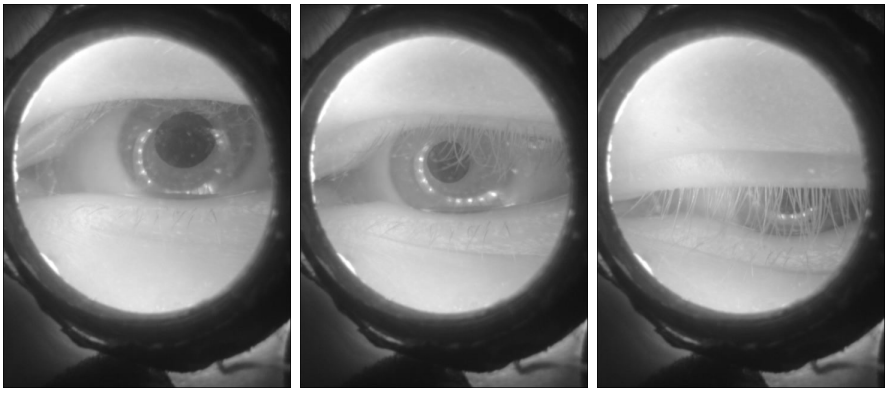


Figure 2.2: Random frame of an Infrared-exposed eye. The input to the algorithm looks similar to these frames. The output are coordinates of the center of the pupil. The left frame is a very clear example. In the middle frame, there are bad lighting conditions and it is partly occluded by lashes. The right shows a frame in very bad conditions, with the pupil almost completely occluded by eye lashes.

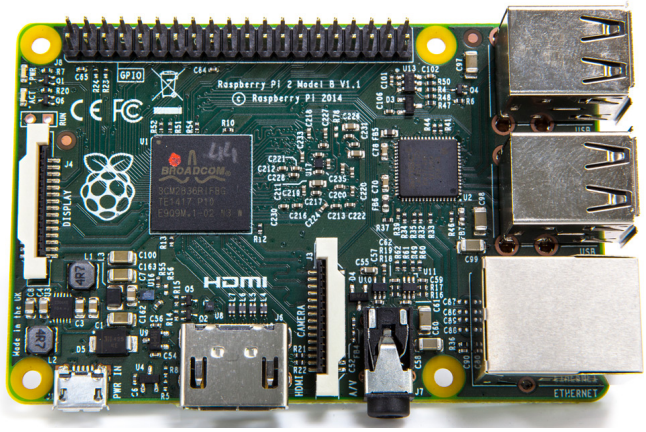


Figure 2.3: The PCB of a Raspberry Pi 2 Model B. On the right, the USB 2.0 and ethernet port are visible. On the top, there is a serial port which is not used in this project. In the bottom area, there is an HDMI slot and the power connector. On the left, there is a MicroSD card providing the operating system.

Chapter 3

Porting to mobile platform

The base algorithm was designed to work on Desktop computers running state-of-the-art Intel x86 hardware and the Microsoft Windows operating system. To support the Raspberry Pi 2 platform, various changes are made to port it to Linux and ARM. On top of it, current desktop processors have a vast difference in terms of performance, compared to mobile platforms. The original algorithm was not designed with strict hardware limitations in mind. Due to this, parts of the algorithms have been rewritten or removed to ensure a calculation in real-time, even on the much slower processor.

Additionally, a network model has been added to communicate to an arbitrary client. This allows transfer of computed data to another platform, such as a video game or VR application.

In section 3.1 all changes made to the implementation are explained exhaustively. Later, section 3.2 evaluates performance gains in modified versions of the algorithm. Finally, section 3.3 will discuss current limitations of the port, which are due to the architecture of the hardware.

3.1 Implementation

The basic architecture is a classical client-server system. The server is the mobile pupil detection system: a Raspberry Pi 2 with up to two connected PlayStation Eye webcams via USB 2.0.

It is connected to an arbitrary client via network. The client could be a game, a game engine or a virtual reality application. For testing, the client is a simple console application receiving the computed results and communications with the test system.

For performance reasons, the server needs a variety of implementation changes in comparison to the base algorithm.

3.1.1 Performance

The platform features an 900 MHz quad-core ARM Cortex-A7 processor with 1 GB of RAM. For the purpose of calculating two eyes in parallel, each frame has two cores available.

The ARM architecture uses the *NEON* instruction set for high performance computations. As the platform has no GPU acceleration, solutions based on CUDA or OpenCL are not available.

To gain the necessary speed to run the algorithm on a restricted platform, several changes to its implementation have been made.

Basic optimizations Various parts of the base code consists of inefficient code, such as loops, memory-inefficient structures and similar. These parts have been rewritten or unrolled, as possible.

Math operations The gcc compiler for ARM tends to use badly optimized variants for common C math library commands. Operations such as *sqrt()*, *pow()* and others were substituted with own implementations using ARM-compatible assembler instructions. This gives a big performance difference compared to non-optimized operations.

The new commands sometimes tend to be less accurate, often calculating with *float* instead of *double* values. This can introduce small jitter. However, it does not change critical parts of the calculations.

OpenCV 3.0 Slight changes to the implementation has been made to achieve a compatibility to OpenCV 3.0. The new release includes hardware acceleration for ARM, the so-called *NEON* instruction set. The Raspberry Pi can not use other forms of hardware accelerations, such as CUDA, OpenCL or any other framework. The new version should improve calculation speed without any changes to the algorithms logic.

Removing parts of calculations As previously described, there was a differentiation between a *clear pupil* and an *occluded pupil* part of the algorithm. This was due to the different visual characteristics in both cases.

However, an analysis with test data showed that for most frames, the *clear pupil* part in fact does have reliable enough results. While there are frames, for which this does not hold true, these frames often can't have good results with either part of the algorithm. Thus, a design decision was made: For frames, where precise robustness can't be satisfied anyhow, performance is more important than possibly rescuing bad results.

The metric for detecting which part of the algorithm should be used is quite slow. It uses filters and other operations, which are not well optimized for ARM. This results in the decision-making step being a major bottle neck of the system.

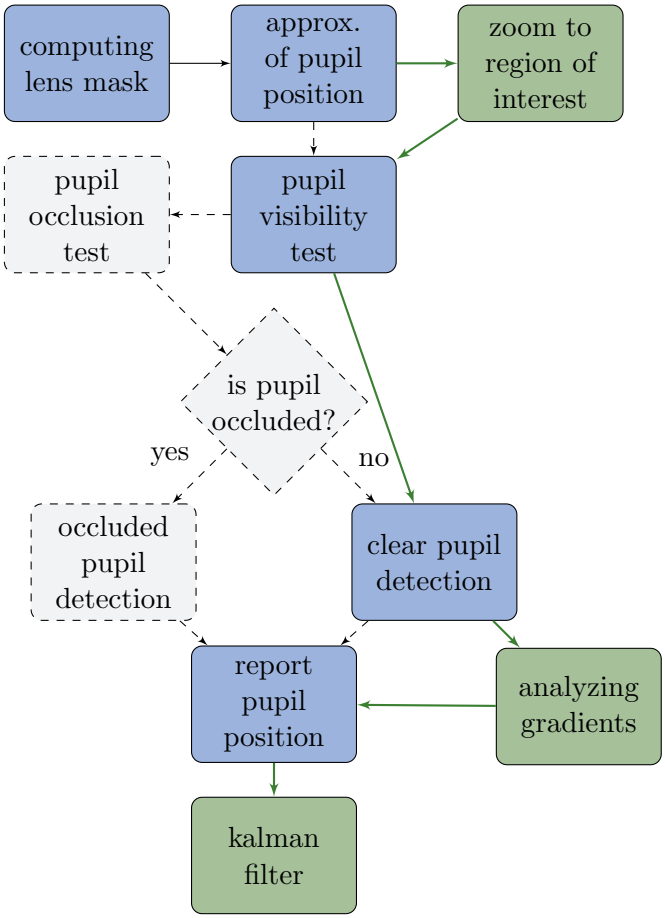


Figure 3.1: Flowchart of the modified algorithm

This problem has been avoided by skipping the decision-making step and calculating every frame with the *clear pupil* part of the algorithm. The flow of the algorithm is described in 3.1. Grey parts of the algorithm are not used anymore. Green boxes are additions by this project. The lower part of the figure includes modifications of chapter 4.

Of course, this change is a trade off between robustness and performance. As the decision metric in particular being a bottle-neck, the change was negligible. This is visualized in figure 3.3.

Region of interest approaches The base version of the algorithm uses masking to decrease the amount of calculations. However, loops are still run through completely, even if only a very small part is masked in. To increase performance vastly, the algorithm was changed to calculate based on resized images rather than masking.

Multithreading The algorithm is implemented in a serial manner, which makes parallelization not feasible without major changes of its logic. It would be possible to pre-calculate filters for next steps of the algorithm. However, additional workload for creating threads and transferring data between threads counter any gains in performance.

As two eyes are processed in parallel, two cores can be used by designating one thread to each eye. Furthermore, the v4l2 driver processing webcams in Linux also uses additional processing power, assigned by the operating system. Thus, around three of four cores are working to their capacity.

Carrying over information from last frame There was an implementation, where the information of the previous frame was used to skip precalculations such as the pupil approximation. However, saccades creates issues with this approach. Furthermore, the increase in performance was not promising enough for further analyses.

In this area, there is possibility for further research. The current algorithm treats every frame as an entirely new calculation, which is necessary for saccades to be detected sufficiently. A more extensive approach could introduce advocating previous results as approximation of following frames.

3.1.2 Networking

To communicate between the mobile platform and a client, which uses the detected pupil data, methods for network communication were implemented. The popular library zeromq[iC15] uses a message-based networking approach. Additionally, a serializing library called MsgPack[Fur15] encodes the content of packages. Both libraries have bindings to several languages, hence making clients in multiple languages possible.

The server opens two ports per eye. On one port, a publishing service will send out detected pupil data after each frame in form of an serialized string. Optionally, the image data can also be send to the listeners. However, this can reduce overall performance in the network as well as the performance of the server, as serializing OpenCV frames takes additional time. On the second port, the server listens to commands, such as changing parameters. This can be used to calibrate the algorithm remotely.

It is be possible to split up both eyes to different Raspberry Pis, and the client connecting to two different addresses.

The client subscribes to the publishing service of the server. It will regularly receive detected pupil positions in form of coordinates in a serialized string. Optionally, the client can open a second port to send commands to the server, such as parameters for calibrating data.

Clients in both C++ and Python were implemented. This shows, that clients and servers with different programming languages are compatible to each other.

3.2 Evaluation

For the purpose of evaluating the speed, two test systems were used.

First, the Raspberry Pi 2 Board B, running Arch Linux ARM in a current version from August 2015. It ships with OpenCV 2.4.11. For testing the speed differences of OpenCV 3.0.0 regarding NEON support, a self-compiled version has been added to the system.

Second, a MacBook Pro with Intel i7 8-core system with 2,5 GHz and 16 GB RAM, running a virtual machine in VMWare Fusion 7.1.2 with Arch Linux 64-bit in a current version from July 2015 and OpenCV 2.4.11.

For checking the capabilities of the platform in conditions similar to a real HMD setup, two PlayStation Eye USB 2.0 cameras were added. Unfortunately, there is currently no custom HMD setup using IR available. This renders frames captured by the webcams not processable. Hence, pre-captured data were used for other parts of the algorithm analyses.

3.2.1 Algorithm performance

In figure 3.2, there are results of performance analyses. It shows the average calculated frames per second (fps) over a time of approx. 1300 frames. As the input data is pre-recorded, it is possible to gain more frames than a common 60 fps camera. In this case, it would result in the webcams being the bottleneck rather than the algorithm itself. However, a higher fps value can result in lower latency and thus a better perception for the user of the VR system.

Furthermore, figure 3.3 shows a profiled overview of the algorithm. This indicates, which parts of the algorithm are a bottleneck, when it comes to processing time.

Before vs After The modified version shows an increase to up to 24 times as fast processing speed. Even though parts of the algorithm are missing, the precision of the results doesn't lose as much. A comprehensive robustness analysis is given in chapter 4.

In raw data, the Raspberry Pi 2 is capable of rendering 96 frames per second (fps). This also means it is sufficient for real-time processing video data. Furthermore, this tops fps of regular video capture devices. The average latency for an input frame to be processed is down to 10 ms.

Two eyes A test using two parallel computations, simulating two separate eyes, has been evaluated internally. There is no decrease in performance for a single eye, which is due to the multi-threaded approach.

OpenCV 2 vs OpenCV 3 OpenCV 3 promised hardware-acceleration for all basic functions for NEON. The difference between ARM compiled

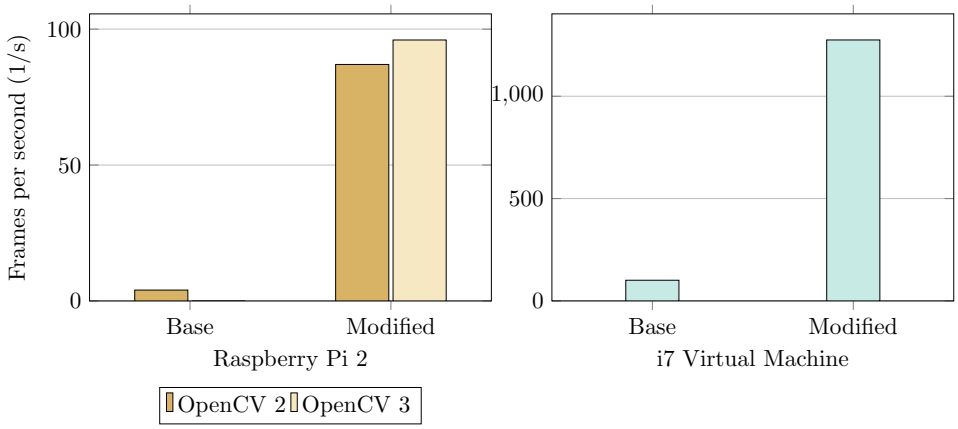


Figure 3.2: Total performance evaluation. This indicates the average amount of frames which are processed per second (fps). A real-time performance is reached at 60 fps. A higher fps than the webcam can reduce latency, enhancing the users perception. OpenCV 3.0 has not been tested on Intel x86 platforms, as the changes are only relevant for ARM-based systems.

OpenCV 2.4.11 and 3.0 is around 9%.

This new version is not yet available in binary form for most linux distributions (as of August 2015). The changes to the source code to support OpenCV 3.0 were marginal. Therefore, an upgrade to 3.0 for ARM-based platforms seems worth it.

3.2.2 Camera performance

The current version uses one CPU core per eye. This leaves two cores open for the operating system. This is advisable, as the v4l2 driver for the PS Eye webcams also uses a high amount of computing power to process requests.

As mentioned, the tests in subsection were evaluated with pre-recorded data. This was necessary, because there is currently no IR setup available. However, there are no major differences when it comes to processing other video sources. Therefore, the system performs similarly when completely working on webcams.

Unfortunately, as it turned out, there are bandwidth bottlenecks regarding the available USB host interface bandwidth on the Raspberry (Section 3.3). Due to this, the operating system can't deliver the expected frame rates. This is not due to the algorithm, which could process frames faster, but hardware limitations (Sec. 3.3).

	Base	Modified
pre-processing	2.4 ms	2.4 ms
extract pupil region	95.3 ms	7.2 ms
visibility test	0.6 ms	0.6 ms
occlusion test	94.0 ms	-
detect (clear / occluded)	11.0 ms / 73.6 ms	3.4 ms
total (clear / occluded)	203.3 ms / 265.9 ms	13.6 ms

Figure 3.3: The average computation times of steps in the algorithm, based on a small selection of frames. This shows possible bottlenecks when it comes to processing speed. The time is measured on the Raspberry Pi 2. These numbers are higher than figure 3.2 suggests. Debug output adds an almost static impact of about 3 ms, which results in a performance decrease of up to 2% to the base algorithm and up to 32% to the modified algorithm.

3.3 Hardware limitations

In the current setup, the Raspberry Pi is not able to fully process two external cameras at full speed and a Wi-Fi dongle at the same time. This seems either due to USB 2.0 stack limitations of the platform, or processing power, as the v4l2 driver almost uses a full CPU core for processing its data.

By USB 2.0 specification, the Raspberry Pi 2 has a theoretical bandwidth of approx. 60 MB/s. However, its operating system running on MicroSD card as well as other devices such as Ethernet share a single bus. This results in a reduced net performance for additional devices.

The PS Eye camera transfers all frames in YUYV data, which results in 2 B per pixel of raw data. In the maximal resolution of 640x480, this results in 35 MB/s of raw data per connected camera. Unfortunately, the PS Eye camera does not seem to support other modes than YUYV, as the Linux v4l2 driver emulates every other option in software. There are currently no speed-optimized drivers available for Linux, as compared to the Code Laboratories Eye driver for Windows[Lab15].

A single camera can process around 50 fps without any frame drops. However, connecting a second camera decreases the available bandwidth and thus performance of each camera. Due to this, the driver options need to be scaled down to 25-30 fps per eye to process the frames correctly without frame drops.

A solution to this might be a switch to another single-board computer with better USB throughput or general performance. Other options would be Intel Atom-based MinnowBoard[Com15] or Korean ODROID[Har15], which both feature USB 3.0 ports as well as an updated CPU. However, these platforms often double or triple the price of a Raspberry Pi 2, making the setup less affordable.

Chapter 4

Enhancing robustness

Computer vision applications like this can be vulnerable for slight variations in between frames. This is due to simple miscalculations regarding the algorithm, but can also happen in cases of noise, bad lightning and other obstacles.

In one use-case of virtual reality and video games, eye movements can directly control what the user will see. This changes the perception of the game. Virtual reality is prone to nausea and seasickness as analyzed by LaViola[LaV00]. A high jitter and jumping in between frames will presumably increase these effects, if they do not correctly respond to the real movements of the users eyes. Therefore, robustness of the computations is an important constraint when keeping in mind future applications of the system.

Additionally, the base algorithm has severe problems with dark eyelashes. This restriction is especially noticeable in case the user wears cosmetics like mascara, severely reducing the amount of correctly detected frames.

This chapter will start with a discussion of tested approaches in Section 4.1. These approaches have been implemented and compared with the base algorithm in terms of speed and robustness. In Section 4.2, an evaluation will show a comparison to the base algorithm. The chapter is concluded with Section 4.3 showing a glance at other promising approaches, which have not been tested yet due to time limitations.

4.1 Various approaches

After optimizing the base algorithm and ensure it working efficient on the Raspberry Pi 2 platform, which was the main goal of the project, several approaches have been tested to further enhance the robustness of detected pupil positions.

This however often comes in hand with additional computations and less approximations for certain parts of the algorithm. Therefore, it results in a

trade off between performance and robustness. This is especially critical for a mobile platform.

Due to this, this section is a mixture of possible features, ideas for future work and already implemented ideas.

4.1.1 Gradient changes to optimize pupil position

In the implementation of the base algorithm, small changes in noise can often result in pixel derivations in between multiple frames, making the calculated pupil center jitter or move, even in time of no gaze movements.

Similarly other common approaches (Timm et al.[TB11], Wisniewska et al.[WRK14], Zhu et al.[ZMRW99]), gradient changes of the pupil in contrast to the iris and sclera were analyzed. Their approaches however commonly use upper-body pictures of people or pictures of heads, having a much lower resolution of the eye. This vastly decreases the amount of pixels of the visible eye, and thus the need for accuracy for near-pixel perfect results. It also reduces problems regarding eyelashes and small reflection in the eyes, which only have a minor impact on the visual data in such a resolution.

In well lighted frames, an approach like these work remarkably well. The corner of the pupil can be found by finding gradient changes in all directions from an arbitrary point inside the pupil. When finding multiple corner points of the pupil, a new ellipse can be calculated which is more robust than the base algorithm proposed in [Gro15].

However, in badly lighted frames the detected positions can be quite off. Often, gradient changes along the pupil, iris and even sclera are very small, making them hard to detect reliably. Due to this, the contrast needs to be enhanced vastly for the approach to work efficient enough. Noise and reflections can furthermore change visual features in-homogeneously, resulting in deformed ellipses.

A downside of a simple high contrast conversion of frames is the loss of information, most drastically seen in badly lit frames. An example is shown in Figure 4.1. The upper line of pictures show the base algorithm approach in green and the gradient approach in blue. The left and middle frame show good and acceptable results, while the right one shows a frame, where both approaches fail. The gradient can not be determined correctly due to bad lightning conditions in the right half of the iris.

To solve these problems, popular approaches for local histogram equalization, like CLAHE as proposed by Pizer et al.[PAA$^+$87] have been tested. Unfortunately, results were not as promising. The main issue for non-optimally lit frames persists. In average it results in similar or only slightly better conditions than simpler contrast changes. However, CLAHE and local histogram equalizations are slower slower.

On top of that, an approach using downscaled versions of the frame has been tested. It improves the results regarding noise and small gradient

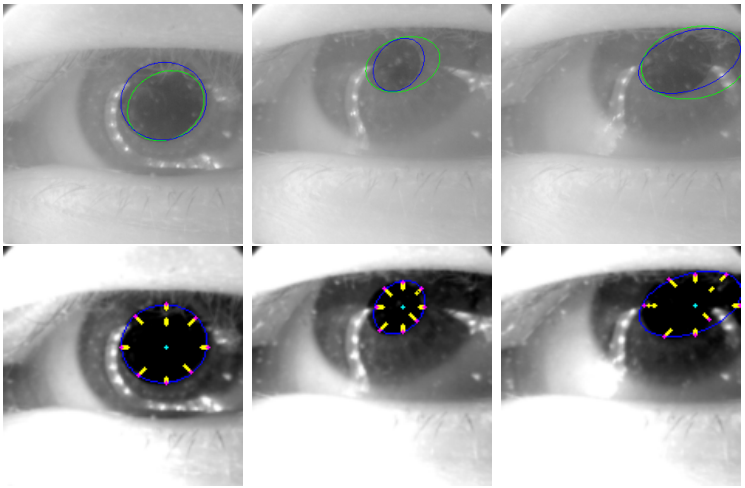


Figure 4.1: Gradient approach. Top: The green ellipse is the base algorithm, the blue one shows the gradient approach. Buttom: Yellow points show possible candidates for the corner of pupil, the pink point is the chosen corner point.

changes and generally enhanced robustness of the approach. Naturally, this technique come in hand with reducing the pixel accurateness of the results.

4.1.2 Kalman filter

To counter-measure pixel derivations and large miscalculations due to spectral light or other disturbances in between close frames, a Kalman filter[Kal60] has been added.

The Kalman filter is especially designed to remove noise and short-term errors in series of events, such as a video of frames. The implementation is straight-forward using 2D input values for each frame. The filter is a post-procession step before communicating the results to listening clients over the network. It estimates a new, most likely position of the pupil center, trying to sort out wrongly detected frames.

This can largely reduce jitter. However, it introduces some latency when it comes to fast movements like saccades.

4.1.3 Compatibility with dark eye lashes

As shown in figure 4.2, there are major differences in eyes using mascara compared to those using none. This can be a hindrance for the algorithm. The dark parts of the eye-lashes are a noise to histograms and gradients. This leads to bad results and the algorithm often not being able to process frames.

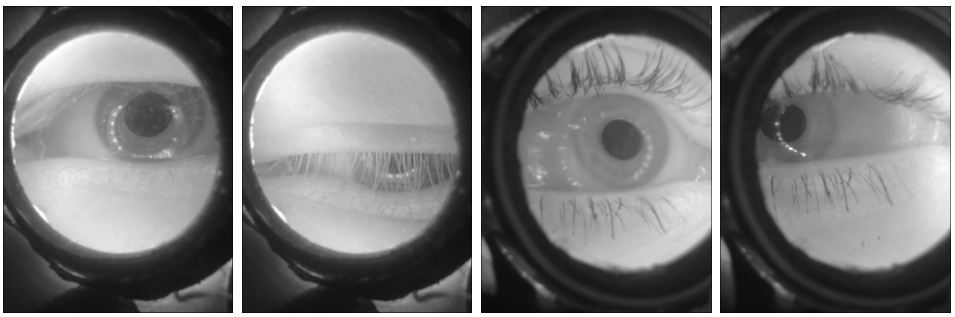


Figure 4.2: Test scenes. Left, scene 1: frames of male eye using no make-up in good/bad conditions. Right, scene 2: frames of female eye using mascara in good/bad conditions.

Dark eyelashes increase the possibility for the pupil approximation to fail. This is especially true, if the lens detection did not work well and there is a fuzzy dark corner. A static region of interest clipping in advance can prevent such behavior. This is no problem, as the IR lens is static and can't move.

To solve these issues, parts of the lens detection have been made restructured. A simpler thresholding replaced the more error-prone lens detection.

4.2 Evaluation

For evaluation of robustness, two different scenes are used. The first scene is a male eye without make-up. The second scene features a female eye with mascara. Figure 4.2 shows a sample frame of each scene. Both scenes were pre-recorded using an IR HMD setup.

The calculations are measured on a Raspberry Pi 2 Board B, running Arch Linux ARM in a current version from August 2015 with OpenCV 3.0.0.

In the top of figure 4.3, it shows error histograms of the base algorithm compared to the most robust modified algorithm. In the bottom, it summarises raw data captured from different versions of the algorithm. *M1* is a version only containing speed changes from 3, including removal of occluded eye detections, region of interests and simpler lens detection. *M2* adds a Kalman filter. *M3* also adds the gradient approach from subsection 4.1.1.

All computed data is compared to ground truth data, measuring the average mistakes. The sum of distance indicates the total movement of the eye. This is an important measure. A low total eye movement indicates less jitter and a calmer movement. This can be a good indication for the users perception, even if the average error is comparatively worse.

The average fps is measured to see whether changes in this chapter have a major impact on the performance, running on a Raspberry Pi 2.

4.2.1 Test scene 1: Eyes without make-up

The modified algorithm is more robust than the base algorithm. The average mistake in pixels stays competitive while being up to 24 times faster. After removing the non-occluded parts of the algorithm, the average mistake actually vastly decreases. This can be a result of false-positives when it comes to the decision metric. However, the *M1* version introduces a high amount of jitter, due to being less accurate.

The addition of a Kalman filter could increase users perception by removing unnecessary jitter. Beside a small reduction of robustness, the total eye movement decreases tremendously.

The gradient approach in particular reduces the amount of false negatively detected frames. This is due to the gradient approach acting as a fallback method.

4.2.2 Test scene 2: Eyes using mascara

In the second scene, the eyes with mascara are harder for the algorithm to process. Furthermore, a very high amount of frames are non-optimally lit. There are reflections of the size of half the pupil, which distract around a fourth of the frames.

The Kalman filter can reduce a little bit of jitter. However, in scene 2, there is a very high number of saccades. Saccades are too fast for the Kalman filter to instantly recognize. Therefore, the average error increases, as the Kalman filter introduces a lag.

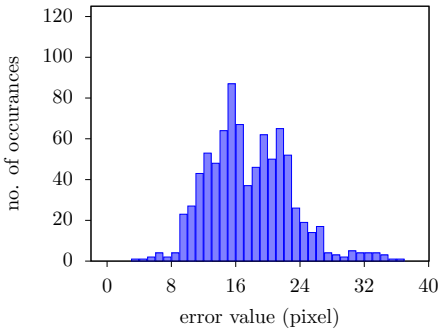
Due to the high amount of reflections, the gradient approach does not work as well as in scene 1. The most right frame in figure 4.2 shows reflections results in such behavior.

4.3 Other approaches for future work

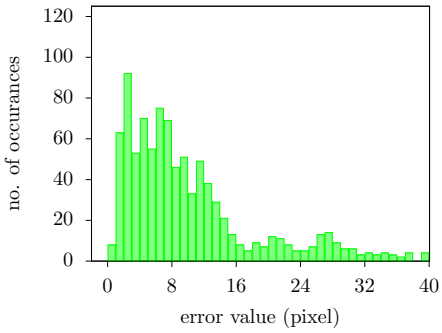
There are other approaches for detecting exact pupil positions in eye tracking, which have not been tested yet. For the purpose of analyzing other methods for future work, here is a short overview.

Ponz et al.[PVS+11] present an approach using Topography-based detection with multiple different resolution pictures.

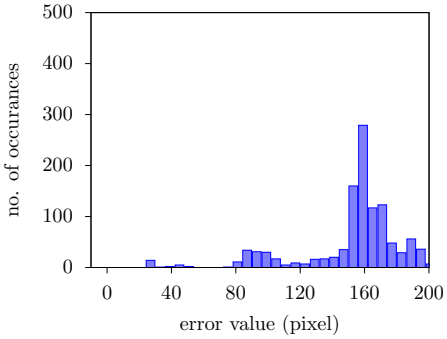
Another interesting work by Markuš et al.[MFP+14] proposes an approach using randomized regression trees to evaluate the position of a pupil. It is reported to be efficient for real-time usage on mobile devices. However, it uses an external dataset for training data.



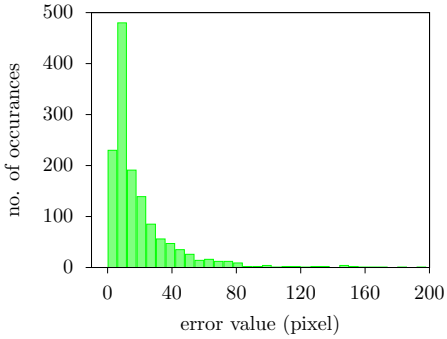
(a) Test scene 1: Base algorithm



(b) Test scene 1: Modified algorithm



(c) Test scene 2: Base algorithm



(d) Test scene 2: Modified algorithm

Scene 1 (1239 frames)	Base	M1	M2	M3
eye movement	20 958	79 422	1854	2441
# false pos.	49	4	5	15
# false neg.	174	182	228	91
mean	17.49	7.38	9.09	10.26
std deviation	7.73	6.32	8.99	9.00
fps	4	96	72	45

Scene 2 (1949 frames)	Base*	M1	M2	M3
eye movement	232 217	58 124	10 220	14 422
# false pos.	254	19	16	26
# false neg.	247	168	192	98
average	258.15	10.57	16.48	24.79
std deviation	1798.03	26.25	32.05	52.77
fps	5	99	64	44

Figure 4.3: Comparison of computed data with ground truth data. In the top, it shows error histograms. Below, there are tables with raw data. All values, except frames per second, are measured in pixels. *Base* is the unmodified version of the algorithm. *M1* consists of speed enhancements discussed in chapter 3. *M2* adds the Kalman filter. *M3* adds all robustness changes in this chapter. *Eye movement* is the total movement of pupil center between subsequent frames. *In scene 2, the base algorithm produces a segmentation fault after 1800 frames, which does not occur on modified versions.

Chapter 5

Conclusion

In summary, the speed evaluation in chapter 3 showed the great capabilities of the algorithm on mobile platforms. It turns out to be fast enough to operate solely on the device in real time. There are some minor limitations when it comes to the host platform regarding USB bandwidth. This can easily be solved by using two Raspberry Pi platforms, one for each eye separately. Alternatively, other host-platforms can be tested.

As a result, the goal of being able to perform in real time has been met. This means, the system is capable of calculating more than 60 frames per second (fps). This speed surpasses usual webcams. It would be possible to further enhance the speed of the algorithm by applying additional temporal metrics, skipping parts of the algorithm for subsequent frames. This is not necessary for real time performance, but could decrease latency. As the goal was met, the focus of the project shifted to robustness.

Chapter 4 could successfully show an improved robustness of results in-between different builds of the algorithm. Unfortunately, the gradient approach has some major problems with reflections, and decreases the performance to a level below 60 fps. Other implementations could vastly decrease the jitter and miscalculations while keeping the average mistake to a minimum. Replacing the lens detection with a static region of interest approach could solve most major issues for eyes with mascara.

In future work, there can be attempts to further increase robustness in the gradient approach of Sec. 4.1 for badly lit frames, especially for reflections. Furthermore, the current system can be integrated with an HMD and tested with real applications, such as mobile games. For this, the Raspberry Pi and the cameras need a custom case, a sufficient battery, as well as an IR light source.

Bibliography

[Com15] Minnowboard Community. Minnowboard. `http://www.minnowboard.org`, 2015. [Online; accessed 12-August-2015].

[Fac15] Facebook. Oculus rift vr. `https://www.oculus.com/en-us/`, 2015. [Online; accessed 12-August-2015].

[Fou15] Raspberry Pi Foundation. Raspberry pi webpage. `https://www.raspberrypi.org`, 2015. [Online; accessed 12-August-2015].

[Fur15] Sadayuki Furuhashi. Messagepack. `http://msgpack.org`, 2015. [Online; accessed 12-August-2015].

[Gro15] Steve Grogorick. Binokulares eye-tracking fuer linsenbasierte head-mounted displays. Master's thesis, TU Braunschweig, Germany, March 2015.

[Har15] Hardkernel. Odroid. `http://www.hardkernel.com/main/`, 2015. [Online; accessed 24-August-2015].

[iC15] iMatix Corporation. zeromq webpage. `http://zeromq.org`, 2015. [Online; accessed 12-August-2015].

[Kal60] R. E. Kalman. A new approach to linear filtering and prediction problems. *ASME Journal of Basic Engineering*, 1960.

[Lab15] Code Laboratories. Cl-eye platform. `https://codelaboratories.com/products/eye/`, 2015. [Online; accessed 12-August-2015].

[LaV00] Joseph J. LaViola, Jr. A discussion of cybersickness in virtual environments. *SIGCHI Bull.*, 32(1):47–56, January 2000.

[MFP+14] Nenad Marku, Miroslav Frljak, Igor S. Pandi, Jrgen Ahlberg, and Robert Forchheimer. Eye pupil localization with an ensemble of randomized trees. *Pattern Recognition*, 47(2):578 – 587, 2014.

[PAA+87] Stephen M. Pizer, E. Philip Amburn, John D. Austin, Robert Cromartie, Ari Geselowitz, Trey Greer, Bart Ter Haar Romeny, and John B. Zimmerman. Adaptive histogram equalization and its variations. *Comput. Vision Graph. Image Process.*, 39(3):355–368, September 1987.

[PVS+11] V. Ponz, A. Villanueva, L. Sesma, M. Ariz, and R. Cabeza. Topography-based detection of the iris centre using multiple-resolution images. In *Machine Vision and Image Processing Conference (IMVIP), 2011 Irish*, pages 32–37, Sept 2011.

[TB11] Fabian Timm and Erhardt Barth. Accurate eye centre localisation by means of gradients. In *Proceedings of the Int. Conference on Computer Theory and Applications (VISAPP)*, volume 1, pages 125–130, Algarve, Portugal, 2011. INSTICC.

[WRK14] Joanna Winiewska, Mahdi Rezaei, and Reinhard Klette. Robust eye gaze estimation. In LeszekJ. Chmielewski, Ryszard Kozera, Bok-Suk Shin, and Konrad Wojciechowski, editors, *Computer Vision and Graphics*, volume 8671 of *Lecture Notes in Computer Science*, pages 636–644. Springer International Publishing, 2014.

[ZMRW99] D. Zhu, S. T. Moore, T. Raphan, and C. C. Wall. Robust pupil center detection using a curvature algorithm. *Comput Methods Programs Biomed*, 59(3):145–157, Jun 1999.