Osnabrück University

Institute of Cognitive Science

Biologically oriented Computer Vision

Master's Thesis

Gaze Tracking Using Common Webcams

Sebastian Höffner

February 1, 2018

Supervisors:

Prof. Dr. Gunther Heidemann

Julius Schöning, M. Sc.

Abstract

Eye and gaze tracking have long been methods to study visual attention. Many devices for gaze tracking are expensive and require specific setup and calibration procedures. For many gaze tracking setups, it is even mandatory to use multiple computers, for showing stimuli and for tracking gaze, respectively.

Today, modern laptops are equipped with enough processing power to process a video stream live. Additionally, many laptops come with a built-in webcam for teleconferencing and video chats. In this thesis, the possibility of performing gaze tracking using a calibration free, feature-based approach on laptops using built-in webcams is explored. To try the model, the free and open source software library Gaze is implemented and evaluated.

It is shown that Gaze reaches very good eye tracking capabilities and manages to be easily usable and extendable. Its gaze tracking abilities are still to be improved, but because of its modular structure existing solutions like pre-trained neural networks can be integrated to leverage their strengths.

Acknowledgements

I would like to thank Prof. Heidemann and Julius Schöning for supervising this thesis, for giving me useful remarks about the topic, and encouraging me to follow my ideas. Many thanks to Andrea Suckro, for her invaluable support consisting of hours of debugging, proofreading, general life counseling, and all other issues which came up during the thesis. I am also very grateful to Carolin Gaß, who helped me a lot with her insightful comments and remarks on the text. To all my friends and family, I would also like to say thanks for their continuous aid.

Sebastian Höffner

Contents

List of Tables

List of Figures

List of Code Listings

List of Abbreviations and Acronyms

API Application Programming Interface.

AR Augmented Reality.

BLAS Basic Linear Algebra Subprograms.

CC0 Creative Commons Zero.

CNN Convolutional Neural Network.

ECG Electrocardiography.

EEG Elecroencephalography.

EOG Electrooculography.

EPnP Efficient Perspective-n-Point.

FOSS Free and open-source software.

FPS Frames Per Second.

GPU Graphics Processing Unit.

GUI Graphical User Interface.

HoG Histogram of oriented Gradients.

MMOD Max-Margin Object Detection.

MRI Magnet resonance imaging.

PnP Perspective-n-Point.

ROS Robot Operating System.

SaaS Software as a Service.

SD card Secure Digital card.

SLA Service-level agreement.

URL Unique Resource Locator.

VCS Version Control System.

VR Virtual Reality.

YAML YAML Ain't Markup Language.

Chapter 1

Introduction

1.1 A brief history of eye and gaze tracking research

In the late 19th century, Helmholtz published his "Handbuch der physiologischen Optik" (Helmholtz 1866), which is an exhaustive summary and critical review of about 150 years of research about the eye, especially its inner workings and functions, and, more importantly, about visual sensation and perception. While a great part of the handbook focuses on eyeball movements and lens accommodation, attention is frequently brought up.

> Erst indem wir unsere Sinnesorgane nach eigenem Willen in verschiedene Beziehungen zu den Objecten bringen, lernen wir sicher urtheilen über die Ursachen unserer Sinnesempfindungen[.][1]
>
> — Helmholtz (1866). Handbuch der physiologischen Optik, p. 452.

With these words Helmholtz makes a very important observation: Humans are only able to reason about what they perceive with their eyes because they can willfully direct them towards objects. In other words, humans can only reason about their visual perception because they can control their visual attention.

Helmholtz distinguishes between "facial sensations" (German: "Gesichtsempfindungen") and "facial perception" (German: "Gesichtswahrnehmungen"). Facial sensations are sensations of the retina and the optic nerve, while facial perception denotes the semantics humans attribute to the combination of those sensations. To form semantics of an object, to perceive it, humans have to move their eyes to the right position to get the needed sensory information. Thus they need to attend an object with their eyes. Turning this conclusion around and thinking ahead, one can make the assumption that what humans attend, that is what they look at, is what they are interested in because they have to actively direct their eyes towards that object.

But where are humans looking at? Huey (1908) tried to answer this question for reading tasks. He built a device as depicted in Figure 1.1 which is attached to the locally sedated cornea using

[1] Translated quotation (Helmholtz 1866): We learn to reason about our sensory impressions only because we can establish different relations between our sensory organs and objects at our own discretion.

a little cup made of plaster cast. The other side of the device points at smoked paper onto which it draws the movements of the attached eye. Using the device, Huey was able to report accurate measurements of fixations and saccades, stops and rapid eye movements, which he used to analyze reading behavior.

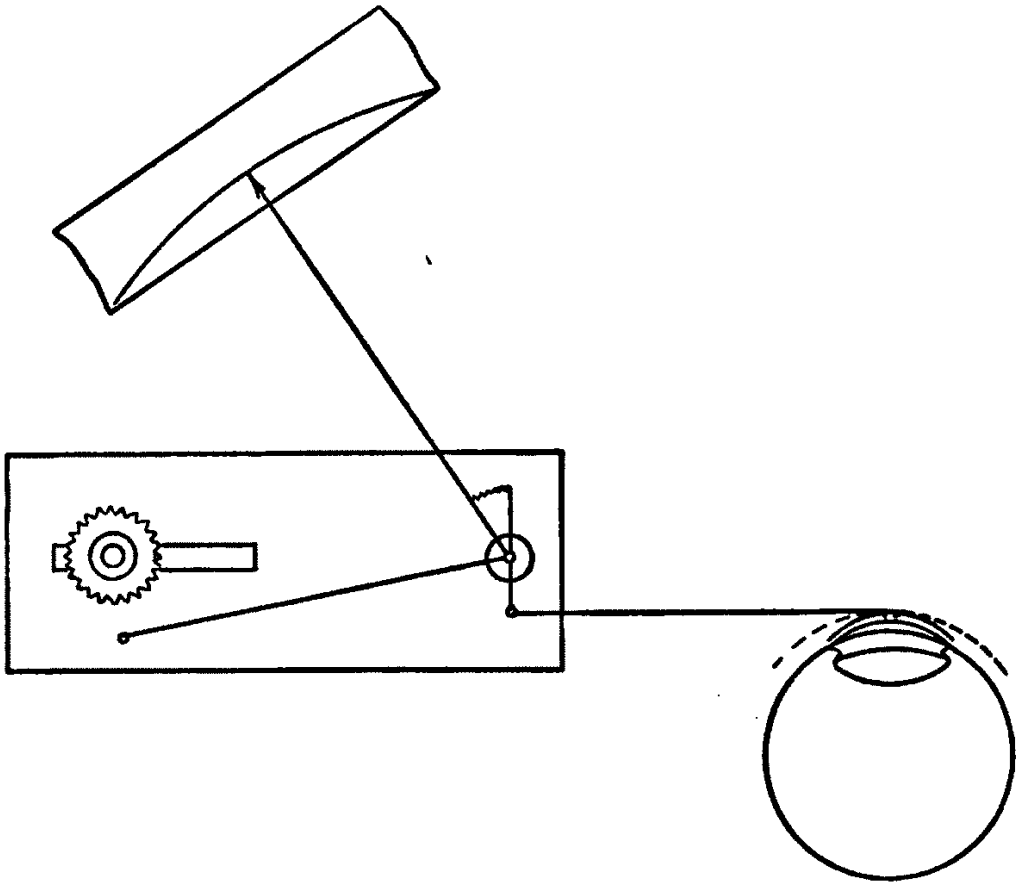

Figure 1.1: Schema of Huey's eye tracking device. It works similar to a seismograph. Image from Google's scan of the New York Public Library's 1968 reprint of Huey (1908), p. 26.

Following Huey's example, a surge of new eye tracking devices hit the research world. Yarbus (1967) became famous for his research on how a given task influences eye movements in comparison to free exploration of a scene. The device he used is similar to Huey's, but it is attached to the eyes without plaster cast, but with small rubber cups. These early methods by Huey and Yarbus are intrusive leading other researchers to the investigation of non-intrusive ones.

Today, many different eye and gaze tracking tools exist. Chennamma and Yuan (2013) categorize them into four kinds: Electrooculography (EOG), scleral search coils, infrared oculography, and video-oculography. EOG determines the rotation of eyes by measuring the electrical fields around them. It is used in electroencephalography (EEG) research to remove blink artifacts, although other techniques using more sophisticated eye tracking have been proposed for artifact rejection (Noureddin et al. 2012). Another exemplary use case for EOG and EEG is sleep research, where

they can be used to determine sleep stages, or as an indicator for lucid dreams (Appel et al. 2017). Scleral search coils are search coils attached to or embedded into special contact lenses. By moving the search coils inside a magnetic field, their exact poses can be measured with great accuracy and resolutions (Shelhamer and Roberts 2010). In infrared oculography, an eye is illuminated by infrared light and the intensity of the reflected light is measured. It is very coarse but can be used during magnet resonance imaging (MRI) (Chennamma and Yuan 2013). The most important kind is video-oculography, in which eyes and gaze are tracked by analyzing video streams. It comes in many varieties: employing one or many cameras, using visible or infrared light, constraining subjects by means of chin rests or similar tools, with cameras attached to special glasses, computer screens or standing in front or around the participants, and much more. As many types of video-oculography there are, as many different use cases do they have. Eye and gaze tracking is used in research on attention and perception in neuroscience as well as psychology (Duc et al. 2008; Duchowski 2002), in marketing (Wedel and Pieters 2008), process planning and optimization (Duchowski 2002), but also in usability research and human computer interaction (Duchowski 2007). Eye and gaze tracking research have long been limited to laboratory settings, but with better and smaller hardware experiments are more often performed in real-world scenarios, for example, in front of any computer (Papoutsaki et al. 2016), on mobile phones and tablets in various locations (Krafka et al. 2016) or with head mounted tracking devices in retail shops (Vilks 2017). With the advent of augmented reality (AR) and virtual reality (VR) even more applications arise, for example foveated rendering can be used to improve rendering performance in VRs (Patney et al. 2016).

Strictly speaking, it is important to distinguish between eye tracking and gaze tracking. In this thesis, eye tracking refers to the process of detecting the positions of the eyes or eye centers – the pupil centers. Gaze tracking, on the other hand, refers to the process of estimating the gaze point on a computer screen. In the literature the two terms are used interchangeably, though mostly only the term eye tracking is used while gaze tracking is not used at all. Especially the commercial hardware and software solutions presented in Section 2.2 are often marketed as eye tracking devices, while in fact they also perform gaze tracking. This thesis will try to use the two terms distinctively, but since they are closely interwoven it might not always be possible.

1.2 About this thesis

The goal of this thesis is to implement the gaze tracking library Gaze which employs a calibration-free geometric feature-based model and is easy to use, modify, and extend. The library focuses only on tracking of screen directed gaze, that is it is designed only to track subjects gazing at a computer screen and to estimate where on the screen the subjects gaze at. It requires only a common webcam to keep its deployment cost at a minimum. One important aspect is that the library should be Free and open-source software (FOSS) so that everyone interested in the project can reuse and modify parts of it for any purpose.

In the first part, an overview of the current state of the art in eye and gaze tracking hardware and software will be established, and different open source eye and gaze tracking solutions will be briefly evaluated. Then the methods and models will be presented, starting with the geometric model used in and the architecture of Gaze. The eye tracking model is chosen because of its success in eyeLike (Hume 2012), an implementation of Timm and Barth (2011). As an alternative to the geometric model, a pre-trained neural network by Krafka et al. (2016) will be introduced and employed. After the introduction of the methods and models, the Gaze library will be described

in more detail to explain how the geometric model is implemented and how the library can be used or extended. One example for such an extension is the incorporation of the pre-trained convolutional neural network (CNN) of Krafka et al. (2016). Following the library description, the results of the different methods and models will be discussed and the two approaches, geometric model and CNN, will be compared. It will be shown that the eye tracking approach works as good as in its original paper, even though the rest of the geometric model is not as good as was hoped for. Finally, a brief lookout will be given to discuss future steps beyond this thesis.

Thus the scope of this thesis has four main points: First reimplementing a model for eye tracking, second use a simplified model to perform gaze tracking on a screen surface, third compare the model with a pre-trained CNN, and fourth realize all these points in a free, open, and reusable software library which only needs a webcam video.

1.2.1 Note about online references and other sources

Online references

Since this thesis focuses on the development of a computer software library, many resources are not available as classical journal or conference papers, or books. Instead, blogs and websites, documentation pages and other formats used as references are published exclusively on the internet. While such sources are cited with a Unique Resource Locator (URL) and, if they are listed in the references, a retrieval date, it is possible that over time internet contents change. If some content is no longer available online or changed by the time you read this thesis, please try to access the content through the Internet Archive's Wayback Machine[2], which hopefully is still around. Often links to software products will be provided directly as part of the text. To avoid URLs in the middle of sentences, they will be placed inside footnotes.

Photographs from Pexels

During the development of Gaze, several photographs of faces differing in backgrounds, poses, lightings, and other conditions, are used. Unless otherwise noted, all photographs in this thesis are either taken by the author or downloaded from the website Pexels[3]. Pexels releases all images into the public domain, using the Creative Commons Zero (CC0) license. This allows the usage of the photographs without the need of any attribution (Creative Commons 2018).

File names and source code availability

In many situations, there will be source code or similar code listings. Some are annotated with a file path, denoting in which files they can be found. These file paths are usually relative to the library's source code's root directory or the thesis' source code's `assets/examples` directory. It should become clear from the context, whichever is correct. In case you did not receive this thesis in a print format with an attached Secure Digital card (SD card) containing the data, you can find the source code for the Gaze library and the thesis online. A list of URLs can be found in Table 1.1. The two code repositories contain the tagged commits `thesis.shoeffner` which reference the state at the time of submission.

[2] https://archive.org/web

[3] https://pexels.com

Table 1.1: Links to the thesis' resources and source code.

Description	Link
Gaze Source Code	https://github.com/shoeffner/gaze
Gaze Documentation	https://shoeffner.github.io/gaze
Thesis PDF & Materials	https://shoeffner.github.io/mthesis
Thesis Source Code	https://github.com/shoeffner/mthesis
Continuous Integration	https://semaphoreci.com/hoeffner/gaze

Exchange rates between USD and EUR

In Section 2.2 all prices of hardware and software are given in EUR. Some prices were only available in USD on the manufacturers' websites, so to keep the currency comparable all USD prices were converted and rounded to EUR. The exchange rate used was USD 1 = EUR 0.804 804, as reported by XE.com Inc.[4] on January 28, 2018 at 19:48 UTC.

[4] https://xe.com

Chapter 2

State of the art

This chapter will outline the current state of the art in eye and gaze tracking approaches which focus on video-oculography, as those are most relevant for this thesis. First, a few of the models for gaze tracking are presented, covering pupil localization, head pose estimation, and gaze point estimation. Then, commercial gaze tracking solutions will be presented, covering examples for remote, mobile and VR gaze trackers, but also some webcam gaze trackers and some analysis software. It is important to note that although they are coined *gaze* tracking solutions here, which they are, most are marketed simply as *eye* tracking solutions. After a brief coverage of the commercial solutions, a number of FOSS projects will be compared. Almost all of them focus on webcam based gaze tracking, which is what this thesis aims to achieve as well.

2.1 Models for eye and gaze tracking

Video-oculography tracking essentially requires two kinds of models, one model to detect pupil centers and one model to estimate the gaze point. Since in this thesis subjects are supposed to be able to move their head freely, a third model to estimate the head pose is needed. In this section, a few models for these purposes will be briefly reviewed. For an in-depth review of some of these and many other models, please refer to Hansen and Ji (2010), who compiled a detailed review of eye and gaze models.

For pupil detection, it is in general possible to distinguish between shape- and appearance-based models (Hansen and Ji 2010), most of them work on image patches containing the eyes and a small area around them. Shape-based models assume the iris as a circular object and optionally add ellipses around it to model the sclera or eyelids. Some of those methods use a circular Hough transform to find the pupil centers (George and Routray 2016; Soltany et al. 2011). Other models perform local fits of circles or ellipses into the image using expectation maximization or random sample consensus to detect the eyes or pupils (Hansen and Pece 2005; Li et al. 2005). Appearance-based models search for other image features, such as color to discard the white sclera in different color spaces (Periketi 2011). Others discard eye center candidate pixels if they have high entropy, stating that the sclera's blood vessels and illumination differences lead to it (Fini et al. 2011). Machine learning models like support vector machines are another method to detect facial features like eyes (Park et al. 2002). This thesis will build on a feature-based

approach using the gradients pointing from a dark iris towards a bright sclera as indicators for eye centers (Timm and Barth 2011).

To estimate the gaze point many models use infrared light and track the first Purkinje image, which is the reflection of the light source on the cornea (Ohno and Mukawa 2004). Others use multiple cameras to estimate the 3D head pose and approximate the eyeball centers to cast rays from the eyeballs through detected pupils (Newman et al. 2000). Many methods for gaze estimation require some kind of calibration (Hansen and Ji 2010), but some research is also done to use calibration-free methods, for example using Gaussian mixture models (Xiong et al. 2014) or geometric calculations (Nagamatsu et al. 2009). However, the calibration free methods apply some constraints. Xiong et al. (2014) only distinguished between left and right, and Nagamatsu et al. (2009) used a chinrest to avoid head movements and to keep the head in focus of four cameras.

Face detection is an old problem in computer vision with a popular solution of applying Haar features to detect faces (Viola and Jones 2001). The "300 Faces In-The-Wild Challenge" (Sagonas et al. 2016, 2013) sparked broad interest in improving the state of the art. For the challenge, which took place twice, Sagonas et al. (2013) proposed a unified landmark scheme to compare detection results. As a side effect, all methods relying on those landmarks provide simple means to extract image patches containing the eyes, which can then be used for further processing with one of the many methods for pupil detections mentioned above. This removes the need to detect the eyes specifically, as is for example done by Sirohey and Rosenfeld (2001). The most successful contributions to the "300 Faces In-The-Wild Challenge" in both installments rely on CNNs (Fan and Zhou 2016; Zhou et al. 2013).

2.2 Commercial gaze tracking solutions

Commercial solutions for gaze tracking come in a great variety. There are hardware systems with one or multiple cameras, coming with specialized computer hardware or without, and some having their own software solutions to visualize and analyze the data recorded with them, while others rely on other software. Few manufacturers focus on webcam solutions, most built highly specialized hardware instead. The prices range from about EUR 100 to prices beyond EUR 20 000 (Biggs 2016; Mahler 2017). Most commercial gaze trackers state their accuracy in degrees of visual angle. With an accuracy of 1°, a gaze tracker has an error of about 1 cm at a viewing distance of 57.3 cm. In this section, only a segment of available solutions is listed.

In the low-end price range of remote gaze trackers, there are the cheaper Tobii EyeX and Tobii Eye Tracker 4X[1] models, which are not for research but gaming. With prices of EUR 159, they currently are the cheapest gaze tracking hardware available. Tobii hardware comes with a developer kit for Windows computers and can be purchased either as standalone gaze trackers or built-in modern gaming laptops. Their only competitor within the low price class, The Eye Tribe[2] which sold trackers for EUR 99 to EUR 160, was acquired by Oculus in late 2016 (Constine 2016) and is no longer selling its products on their website. Still below EUR 1000 are the GP3[3] if bought without any software, which raises the price up to EUR 2800. It has an accuracy of 0.5° to 1° and a frame rate of 60 Hz. In the higher price segment for remote gaze tracking, Tobii offers a frame rate of up to 600 Hz with an accuracy of 0.4°. Another remote gaze tracker with

[1] https://tobiigaming.com
[2] http://theeyetribe.com
[3] https://gazept.com

up to 1000 Hz and a similar accuracy is the EyeLink 1000 by SR Research[4]. Smart eye[5] offer multi-camera setups for setups with multiple monitors using up to eight cameras. They have an accuracy of 0.5° and between 60 Hz and 120 Hz. Another multi-camera setup comes from LC Technologies[6].

For mobile eye tracking pupil labs[7] offers a unique solution: All their hardware as well as their software is open source and can potentially be built manually. They offer their 200 Hz gaze tracking glasses from EUR 1000 and also have add-ons for the Microsoft HoloLens, an AR kit, and the HTC Vive, a VR kit. Another choice for VR is the Fove[8] at EUR 482, a gaze tracking solution developed specifically for VR: They specialize in foveated rendering (Patney et al. 2016) to improve the performance of graphics renderings in VR. But Fove and pupil labs are not the only contenders in the gaze tracking for VR and AR market. Ergoneers[9] sell a hardware kit which can be adapted to VR but is also designed to be integrated into helmets. SensoMotoric Instruments[10] offered a wide range of remote, mobile, AR, and VR solutions in the higher price ranges, until they were bought by Apple in 2017 (Rossignol 2017).

Although many hardware manufacturers also ship their own analysis software, there are independent software companies offering analysis software for gaze tracking. One such company, interactive minds[11], works closely together with the hardware manufacturer LC Technologies. Other software companies include iMotions[12], which offers software for gaze tracking but also for EEG, Electrocardiography (ECG), and other biometrics. Companies like the Institut für Wahrnehmungsforschung[13] offer to conduct gaze tracking studies, they specialized in marketing and advertisement.

A modern software as a service (SaaS) approach is done by Eyezag[14], which offers a service to perform gaze tracking for websites, only using user webcams. Similarly the platforms EyesDecide[15] and the related xLabs[16] offer browser integrations to record, replay and analyze user gazing behavior on websites by employing webcams.

2.3 Free and open-source webcam gaze tracking software

Several FOSS projects attempt to perform gaze tracking. They all have different requirements and use cases: They require webcams, modified webcams, or depth cameras and they need either images of the full face or of individual eyes. Most projects only track eyes. A comparing overview over the software presented in this section can be found in Table 2.1.

The eye tracking implementation eyeLike[17] by Tristan Hume is the most important work for this thesis. The implementation uses gradients to detect eye centers (Timm and Barth 2011). It is not suited to be integrated into other software and can be seen as a reference implementation of the eye center detection algorithm. Gaze implements the same algorithms but provides a more flexible interface.

[4] http://sr-research.com
[5] http://smarteye.se
[6] http://eyegaze.com
[7] https://pupil-labs.com
[8] https://getfove.com
[9] http://ergoneers.com
[10] https://smivision.com
[11] https://interactive-minds.com
[12] https://imotions.com
[13] http://institut-fw.de
[14] https://eyezag.com
[15] https://eyesdecide.com
[16] https://xlabsgaze.com
[17] https://github.com/trishume/eyeLike

Table 2.1: Comparison of open source gaze tracking software. Unfortunately, not all could be tested, thus the results rely on the respective author's reports.

Software and Author	Input	Tracks	License
deepgaze (Patacchiola and Cangelosi 2017)	not available	head	MIT
eyeLike (Hume 2012)	Face	eyes	MIT
gazr (Lemaignan et al. 2016)	Face	head	Apache 2.0
iTracker (Krafka et al. 2016)	Face, eyes, mask	gaze	Research-only
OpenGazer (Ferhat 2012)	Face	gaze	GPL 2.0
webcam-eyetracker (Dalmaijer 2015)	Infrared eye	eyes	GPL 3.0
Webgazer.js (Papoutsaki et al. 2016)	Face	gaze	GPL 3.0
Gaze	Face	eyes	MIT

Opengazer[18] is a software originally developed by Piotr Zieliński which tracks gaze after a few calibration steps using a normal webcam. It was published in 2010 and last updated in 2013, but a fork[19] of the project was created by Onur Ferhat in 2013 and maintained until 2016. The fork achieved errors of about 1.5°, improving the original project by about 17.5 %, and being about 1° short of Tobii X1 Light Eye Trackers (Ferhat 2012).

One of the more prominent examples is PyGaze[20], a software primarily used to perform eye tracking and gaze tracking analysis (Dalmaijer et al. 2014). It is written and Python and maintained by Edwin Dalmaijer and Sebastiaan Mathôt. Since 2015 it features a webcam-based pupil tracker named webcam-eyetracker (Dalmaijer 2015). It tracks the pupil of one eye and uses infrared light. Because of that, it is not suitable for all webcams, as most have a built-in infrared filter – though for some webcams it is possible to remove it. It also needs to have specific lighting conditions; Dalmaijer reports he had to turn off all regular lights when using it.

The package gazr[21] by Séverin Lemaignan integrates with the Robot Operating System (ROS)[22]. Gazr is designed for human-robot interaction and currently work is done to extend it with gaze tracking capabilities. So far it performs head pose estimation using webcams or RGB-D cameras (Lemaignan et al. 2016).

In 2016, Kyle Krafka and his colleagues published the dataset GazeCapture alongside the pretrained CNN iTracker (Krafka et al. 2016). It performs gaze tracking using iPhone and iPad cameras. Gaze employs the iTracker CNN to compare the geometric model against it, the results are in Section 5.3.

Alexandra Papoutsaki enabled online gaze tracking in web browsers using WebGazer.js[23] (Papoutsaki et al. 2016). They offer a few example programs and make it easy to integrate their tool into websites. WebGazer.js uses the webcam and calibrates the camera by having the subject clicking a number of times at different screen positions they gaze at.

The Python package deepgaze[24] by Massimiliano Patacchiola uses CNNs to perform various human-computer interaction tasks. It claims one of its features is to be able to do gaze tracking,

[18] http://inference.org.uk/opengazer
[19] https://github.com/tiendan/OpenGazer
[20] http://pygaze.org
[21] https://github.com/severin-lemaignan/gazr
[22] https://ros.org
[23] http://webgazer.cs.brown.edu
[24] https://github.com/mpatacchiola/deepgaze

but the current version does not yet offer this functionality. However, it does perform head pose estimation (Patacchiola and Cangelosi 2017). Since its publication about the head pose estimation is from 2017, it is possible that gaze estimation will be added in the future.

Chapter 3

Methods and models

Estimating the gaze points is done in Gaze using a geometric model realized in a flexible pipelined architecture. Most pipeline steps consist of smaller models which solve parts of the gaze estimation problem, others serve for input and output of the data. This chapter details the models and gives an overview of Gaze's architecture. Finally, there will be a short introduction of iTracker, the CNN trained by Krafka et al. (2016), an alternative deep learning model for gaze tracking.

3.1 Geometric model

The goal of the geometric model is to find the gaze points on the screen using ray casts. A ray cast is performed from an eyeball center through its pupil. The intersections of the screen plane and the ray casts are the gaze points. A line–plane intersection (Wikipedia contributors 2017a) can be expressed using three points of the screen plane, an eyeball center and a pupil in the same 3D coordinate system. Given an eyeball center $c \in \mathbb{R}^3$ and a pupil center $p \in \mathbb{R}^3$, the points on the line from the eyeball center through the pupil can be described as

$$c + (p - c)t, \tag{3.1}$$

with $t \in \mathbb{R}$ shifting points on the line. The points on the screen plane can be described by three screen corners $tl, tr, br \in \mathbb{R}^3$, one functioning as the reference point and two as the directions into which the screen plane extends:

$$tl + (tr - tl)u + (br - tl)v, \tag{3.2}$$

with $u, v \in \mathbb{R}$ shifting points on the plane. The intersection between Equation (3.1) and Equation (3.2) is thus

$$c + (p - c)t = tl + (tr - tl)u + (br - tl)v \tag{3.3}$$
$$c + (p - c)t - tl = (tr - tl)u + (br - tl)v \tag{3.4}$$
$$c - tl = -(p - c)t + (tr - tl)u + (br - tl)v \tag{3.5}$$
$$c - tl = (c - p)t + (tr - tl)u + (br - tl)v. \tag{3.6}$$

Or, in a more concise matrix and vector form, Equation (3.6) can be expressed as

$$\begin{pmatrix} c_x - tl_x \\ c_y - tl_y \\ c_z - tl_z \end{pmatrix} = \begin{pmatrix} c_x - p_x & tr_x - tl_x & br_x - tl_x \\ c_y - p_y & tr_y - tl_y & br_y - tl_y \\ c_z - p_z & tr_z - tl_z & br_z - tl_z \end{pmatrix} \begin{pmatrix} t \\ u \\ v \end{pmatrix}. \tag{3.7}$$

To find the intersection, this matrix equation needs to be solved for t so that the intersection can be calculated by inserting t into Equation (3.1), for both eyes independently. This can be done by inverting the 3×3 matrix, which works as long as the line is not parallel to the plane, in other words as long as the line vector stays linear independent to the direction vectors. In practice this should never be a problem, as the methods to detect the face and pupils would fail earlier, resulting in these calculations never to be executed.

The difficulty is to find all variables such that Equation (3.7) can be solved. The following sections will show how the eyeball centers are determined using a generic 3D head model, followed by how Gaze detects faces and eyes in an image. The detected face landmarks are used to find the pupil centers and project them into the 3D model, as well as to estimate the head pose in relation to the camera. Once the relation to the camera is established, the distance is estimated to calculate the screen position in the model coordinate system. The eyeball centers, pupil locations in model coordinates, and screen corners in model coordinates can be inserted into Equation (3.7) to find t and calculate the gaze points. Eventually, the gaze points are converted into target coordinates, for example the pixels of the screen.

3.1.1 3D head model and eyeball centers

The eyeball centers can be modeled as part of a 3D head model which will be needed to estimate the head pose in relation to the camera. A simplified 3D head model using six landmarks is proposed by Mallick (2016). The landmarks involved are the tip of the nose (pronasal), the point at the bottom front of the chin (gnathion), the two outer corners of the eyes (exocanthions), and the two mouth corners (cheilions). The 3D model does not use the metric system but coordinates within "some arbitrary reference frame" (Mallick 2016). The model uses the tip of the nose as the origin and spans the coordinate system parallel to the standard anatomical planes. The x axis is parallel to the coronal and transverse planes, with the positive values to the left from the head's perspective. Mallick (2016) uses left and right from the viewer's perspective. Gaze uses left and right from the head's perspective which is more in line with Swennen (2006). Thus when looking at the model, the x axis stretches to the right. The z axis is parallel to the transverse plane and lies inside the mid-sagittal plane, pointing away from the face. The y axis is orthogonal to the x and z axes and points upwards in relation to the head. Table 3.1 summarizes the model points and Figure 3.1 visualizes the locations and the coordinate system. The model is converted to the metric system to be useful for Gaze using data from Weinberg and Marazita (2009), in particular the mean outer canthal width. This is the width of the left and right eyes' outer corners, from ex_r to ex_l. Weinberg and Marazita (2009) report a mean outer canthal width of 86.71 mm, measured using 3D stereophotogrammetry and within 1845 European Caucasian individuals above the age of 18. In the 3D model, the outer canthal width is 450. Since the idealized head model assumes symmetry along the mid-saggital plane it follows that $225 = 43.355\,\text{mm}$, or $1 \approx 0.192\,69\,\text{mm}$. This relation can be used to calculate the metric model, as done in Table 3.1.

Table 3.1: 3D head model by Mallick (2016). The first columns describe the landmark and name its abbreviation (Swennen 2006), followed by the "300 Faces In-The-Wild Challenge" index (Sagonas et al. 2016). Then, the 3D model coordinates are described as used by Gaze and in the original model. The eyeball centers are an exception as they are only important for the ray cast and do not have conventional soft tissue landmark abbreviations nor model points in the original model.

Landmark	Abbr.	Index	Gaze [mm]	Mallick (2016)
Pronasal	*prn*	31	$(0, 0, 0)$	$(0, 0, 0)$
Gnathion	*gn*	9	$(0, -63.6, -12.5)$	$(0, -330, -65)$
Exocanthion right	ex_r	37	$(-43.3, 32.7, -26)$	$(-225, 170, -135)$
Exocanthion left	ex_l	46	$(43.3, 32.7, -26)$	$(225, 170, -135)$
Cheilion right	ch_r	49	$(-28.9, -28.9, -24.1)$	$(-150, -150, -125)$
Cheilion left	ch_l	55	$(28.9, -28.9, -24.1)$	$(150, -150, -125)$
Eye ball center right	(c_r)		$(-29.05, 32.7, -39.5)$	
Eye ball center left	(c_l)		$(29.05, 32.7, -39.5)$	

An initial idea to model the eyeball center was to place it at the center of the palpebral fissure – the line segment between the inner and outer eye corners, the endocanthion and exocanthion – and move it inside the head until the *ex* and *en* are on the eyeball surface. The problem with this idea is that the mean palpebral fissure length is 28.19 mm (Weinberg and Marazita 2009) but the mean eyeball diameter is much less than that: It is reported to be about 24 mm (Davson 2017), or 22.0 mm to 24.8 mm (Bekerman et al. 2014). Instead of solving the equations with a greater diameter or by accounting for the distance between the eyeball surface and the *ex* and *en*, the midpoint between *ex* and *en* is just moved back along the z axis by 13.5 mm, which is the furthest distance between the cornea and the eyeball center (Gross et al. 2008). The endocanthions are assumed to be on the same line parallel to the x axis and with a distance of the palpebral fissure length, 28.19 mm apart from their respective exocanthions. The eyeball centers are then at $(-29.05, 32.7, -39.5)$ mm and $(29.05, 32.7, -39.5)$ mm.

3.1.2 Detecting faces and eyes

Before finding the pupil centers to project them into the 3D model the face of the subject needs to be found. There are various methods available, Frischholz (2018) lists 15 FOSS libraries providing some sort of face detection and additionally lists several websites and commercial software able to do the same. One method is to use OpenCV's pre-trained classifiers, which perform a variant of Haar feature detection using AdaBoost (Viola and Jones 2001). But this method, albeit popular, only finds face and eye boundaries, while Dlib also finds useful face landmarks. King (2014) released a face detector in Dlib which uses five histograms of oriented gradients (HoGs) and Max-Margin Object Detection (MMOD) (King 2015). Gaze uses Dlib's classifier because it offers an advantage over OpenCV's classifier: It detects the 68 landmarks used for the "300 Faces In-The-Wild Challenge" shown in Figure 3.2. These landmarks include the landmarks listed in Table 3.1, so no additional processing and detection steps are needed after the face is detected. Section 4.2.3 explains one downside of using Dlib's model, which is licensing. A possible solution is to use the five landmark model which Dlib offers as an alternative, but it does not detect all landmarks included in the 3D head model. Also using a 3D head model containing the five

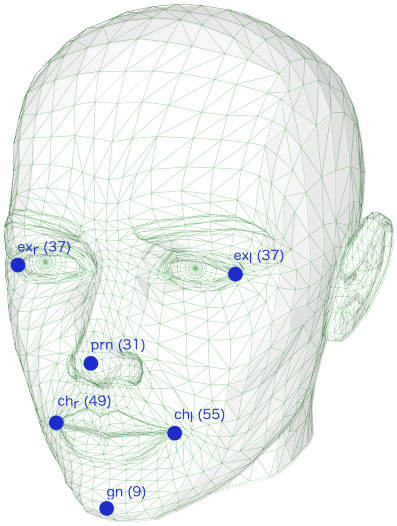

Figure 3.1: Annotated visualization of the 3D head model. See Table 3.1 for the values of the marked landmarks. The head model is adapted from pixabay.com and under the CC0 license.

instead of the six out of 68 landmarks is not stable enough for the head pose estimation, which is discussed in more detail in Section 3.1.4.

Dlib describes detected faces with a bounding box around the detected landmarks as well as a list of landmark coordinates, ordered as labeled in the "300 Faces In-The-Wild Challenge" (Sagonas et al. 2016). The landmarks 37 and 46 in Figure 3.2 correspond to the ex_r and ex_l, respectively, similarly landmarks 40 and 43 denote en_r and en_l. The eyes are extracted by placing a rectangle with ex and en being opposite corners and detecting its center. The eye is cropped to a square with a side-length of 1.5 times the distance between the eye corners and centered around the rectangle's center. To visualize this, examples of processed eyes can be found in the appendix in Figure C.2.

3.1.3 Pupil localization

Finding the pupil centers in the eyes is done following Timm and Barth (2011), inspired by the success of Hume (2012). The algorithm they propose assumes that the iris, the colored part of the eye, is a dark circle on a bright background, the sclera, which implies that there is a strong gradient at its boundary. The gradient direction at the boundary points from darker to brighter areas, that is towards the sclera. To find the center, all points within the image of the cropped eye are assigned a value by a target function and the point with the maximum value is chosen as the pupil center.

The pupil location $p \in \mathbb{N}^2$ in pixels is found by solving (adapted from Hume 2012; Timm and Barth 2011):

$$p = \underset{\hat{p}}{\operatorname{argmax}} \left\{ \frac{1}{N} \sum_{i=1}^{N} w_i \left(\max \left\{ \left(\frac{x_i - \hat{p}}{\|x_i - \hat{p}\|_2} \right)^\top \varphi\left(g_i, \vartheta\right), 0 \right\} \right)^2 \right\}, \tag{3.8}$$

Figure 3.2: The 68 landmarks as detected by Dlib on the left, on the right their original description by Sagonas et al. (2013). In Dlib, the indexes start with 0. Landmarks schema used with kind permission by Stefanos Zafeiriou.

where $\hat{p} \in \mathbb{N}^2$ are the potential pupil locations, $x_i \in \mathbb{N}^2$ are all N pixel locations of the image crop, $w_i \in \mathbb{R}$ are weights for those pixel locations, $g_i \in \mathbb{R}^2$ are the gradients at each pixel location, respectively, and $\|\cdot\|_2$ is the Euclidean norm. A very important function is φ, which is defined as:

$$\varphi(x, \vartheta) = \begin{cases} \frac{x}{\|x\|_2} & \text{if } \|x\|_2 \geq \vartheta \\ 0 & \text{else} \end{cases} \tag{3.9}$$

with $\vartheta \in \mathbb{R}$ as the dynamic threshold depending on all gradient magnitudes

$$\vartheta = \mu_{\text{mag}} + \theta\sigma_{\text{mag}}, \tag{3.10}$$

employing the mean and standard deviation $\mu_{\text{mag}}, \sigma_{\text{mag}}$ over the gradient magnitudes $\text{mag}_i = \|g_i\|_2$, and the model parameter θ, which describes the number of standard deviations. As explained in Section 4.5.1, in Gaze θ can be configured and is set to 0.3 by default, following Hume (2012).

So to detect a pupil center first the gradient image of the eye has to be calculated, for which Gaze uses the standard Sobel filter. Using the gradient magnitudes and the model parameter θ, a dynamic threshold ϑ can be calculated to discard all low gradients and normalize those which are not discarded, as defined in Equation (3.9). Then, for each possible pupil center location $\hat{p}$, each other pixel location x_i is used to evaluate the target function: If the direction from x_i to $\hat{p}$ is similar to the gradient direction g_i, the value will be squared and added to $\hat{p}$'s target value. The similarity measurement is the scalar product: If the two normalized vectors point in the same direction, it evaluates to 1, if they point in the opposite directions it evaluates to -1. Hume (2012) noted that in the paper all these values were taken into account, but vectors pointing inwards should not be considered at all. Thus Equation (3.8) discards negative scalar products instead of squaring them by applying the $\max\{\cdot, 0\}$ function. The possible pupil center location which on average has the most directions to locations which have a gradient pointing into the same or similar directions will be the winning center point.

Sometimes the gradient method leads to problems where the eye crop has some wrinkles, shadows, low eyelids, reflections on glasses, or other illumination changes. Timm and Barth (2011) use an inverted Gaussian filtered image to calculate weights which should give the real pupil higher chances to be selected. The dark parts of the image – low gray values – will thus get higher weights. In `Gaze`, where Dlib uses 8 bit image values, a dark pixel with a smoothed value of 15 would for example get a weight of $255 - 15 = 240$. The extension of discarding vectors facing inwards by Hume (2012) also leads to an improvement, especially because eyebrows and eyelids no longer hint towards arbitrary points inside the sclera.

Once the pupil centers are found, they need to be aligned with the 3D model. To project the pupil centers from image coordinates into model coordinates, the detected landmarks which correspond to the model points are extended to be 3D coordinates, with their z coordinates being set to 0. The affine transformation from the landmarks to the model is estimated using OpenCV's `estimateAffine3D` function. Applying the result transformation to the pupils effectively moves them into the head model. Figure 3.3 shows the 3D head model with pupils and eyeball centers after the transformation.

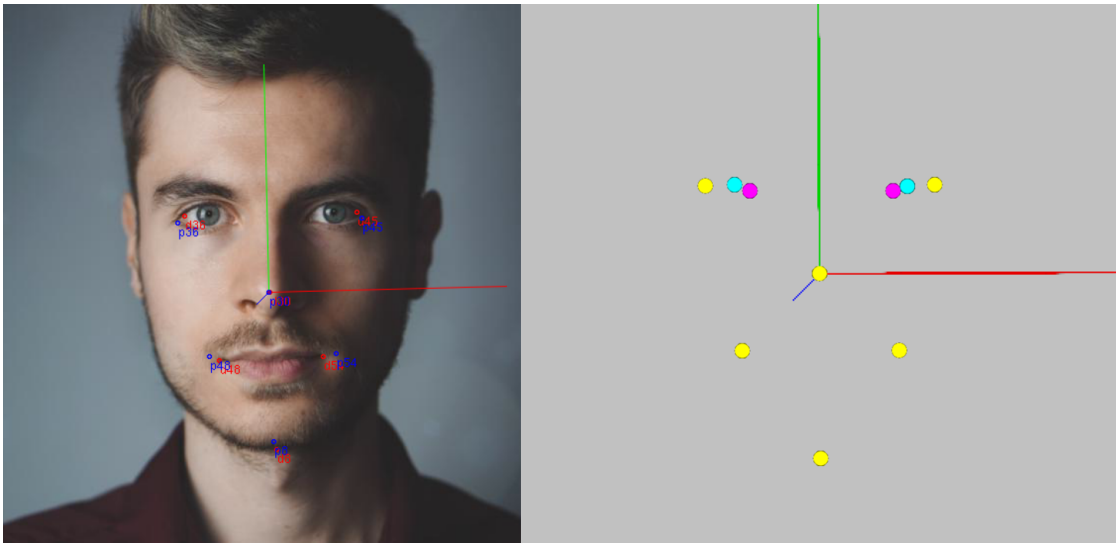

Figure 3.3: Left: Head pose estimation, the red markers are detected by Dlib and the blue markers are a projection of the model to visualize the differences. Right: The cyan pupils and magenta eyeball centers inside the yellow 3D model. The image was visually enhanced by increasing the dots and brightening the background.

3.1.4 Head pose estimation

To properly estimate the screen–head relation the head pose needs to be known. A pose consists of a position or location $(x, y, z) \in \mathbb{R}^3$ and an orientation (α, β, γ), with $\alpha, \gamma \in \{x \in \mathbb{R} | -\pi < x \leq \pi\}$, and $\beta \in \{x \in \mathbb{R} | 0 \leq x \leq \pi\}$. Of course, in its own coordinate system defined above, the head pose is always at $(0, 0, 0)$ with an orientation of $(0, 0, 0)$. But by estimating the head pose in terms of the camera coordinate system, it is possible to derive the camera location, which in turn is fix in relation to the screen. So by knowing the camera location, the screen corners can be found and used to detect gaze points on the screen.

In Gaze head pose estimation is performed closely following the approach outlined by Mallick (2016). The pose only needs to be estimated indirectly by describing an affine transformation from the model coordinates to camera coordinates such that a projection into the image coordinates becomes possible. This affine transformation can be described using a rotation $R \in \mathbb{R}^{3\times3}$ and a translation $T \in \mathbb{R}^3$. Adapted from OpenCV's documentation[1], the model

$$\begin{pmatrix} p'_x \\ p'_y \\ 1 \end{pmatrix} = C\ P\ \begin{pmatrix} R \mid T \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} \tag{3.11}$$

$$= \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} R_{11} & R_{12} & R_{13} & T_x \\ R_{21} & R_{22} & R_{23} & T_y \\ R_{31} & R_{32} & R_{33} & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} \tag{3.12}$$

describes the projection from the 3D model point $p \in \mathbb{R}^3$ to the 2D image point $p' \in \mathbb{R}^2$, using homogenous coordinates. It uses the camera matrix $C \in \mathbb{R}^{3\times3}$, which can be found via calibration or approximated by the image size, as presented in Section 4.5.1. It also uses a projection matrix $P \in \mathbb{R}^{3\times4}$ which reduces the dimensions from three to two, and the affine transformation from the model coordinate system into the camera coordinate system. Finding the values for R and T is called the Perspective-n-Point (PnP) problem. There are multiple algorithms to solve PnP problems. Suitable for six points, as it is the case in Gaze, are Efficient Perspective-n-Point (EPnP) (Lepetit et al. 2009) and an iterative approach which uses a Levenberg–Marquardt optimization (Levenberg 1944; Marquardt 1963; Wikipedia contributors 2018). The latter estimates a hidden parameter set β as an approximation $\hat{\beta}$ such that

$$\hat{\beta} = \underset{\beta}{\operatorname{argmin}} \left\{ \sum_{i=1}^{N} \|p'_i - f(p_i, \beta)\|_2^2 \right\}, \tag{3.13}$$

where β are the values of R and T in Equation (3.12), and $f(p_i, \beta)$ is the estimated projection of a model point p_i. Once found, the parameters can be used together with the distance estimation from Section 3.1.5 to estimate the screen corners as described in Section 3.1.6. In Gaze the Levenberg–Marquardt optimization is used because it subjectively performs slightly better when subjects face the camera more directly, while the EPnP becomes better when subjects turn their heads. A comparison is shown in Figure C.3. Since for gaze tracking subjects look roughly towards the camera, it is more important to estimate frontal images more accurately.

3.1.5 Distance estimation

In general, to estimate the distance between an object and a camera from an image, it is important to know the real size of the object, the sensor size, the resolution, and the focal length. By using the intercept theorem and the pinhole camera model, the distance to an object can be determined. Let $o \in \mathbb{R}$ be the width of the object, $d \in \mathbb{R}$ be the distance between the image plane and the object, $f \in \mathbb{R}$ the focal length, $i \in \mathbb{R}$ the object's size on the image, and $p \in \mathbb{R}$ the pixel width. Then, the distance can be expressed as

$$d = \frac{fo}{i}. \tag{3.14}$$

[1] https://docs.opencv.org/3.4.0/d9/d0c/group___calib3d.html#ga549c2075fac14829ff4a58bc931c033d

Assume the sensor size to be 0.006 35 m and its aspect ratio as 16:9 (Luepke 2005, for an Apple iSight). In that case, the sensor width $s \in \mathbb{R}$ is 0.0055 m. The pixel width p can be found through division of the sensor width by the horizontal resolution, $w \in \mathbb{N}$, so $p = \frac{s}{w}$. In Gaze, the outer canthal width is used to determine the distance. It's model size o is

$$\|ex_r - ex_l\|_2 = \left\| \begin{pmatrix} -0.0433 \\ 0.0327 \\ -0.026 \end{pmatrix} - \begin{pmatrix} 0.0433 \\ 0.0327 \\ -0.026 \end{pmatrix} \right\|_2 = 0.0866\,\text{m}. \tag{3.15}$$

The outer canthal width in the image i is the distance between the landmarks 37 and 46, which is detected by Dlib. It has to be multiplied by p to get its width in m. Thus, to determine d, the only missing value is f. An approximation for f can be measured, for a MacBook Pro with the assumption of the above mentioned sensor size it is about 0.01 m, which is found using the procedure in Appendix A.2. Substituting all variables into Equation (3.14) leads to

$$d = \frac{fo}{i} = \frac{0.01\,\text{m} \cdot 0.0866\,\text{m}}{i\text{pixels} \cdot 0.0055\,\text{m/pixels}} = \frac{0.1575}{i}\text{m}. \tag{3.16}$$

This can be used as an approximate distance measure, it is however only accurate if the head is parallel to the camera. For the purpose of this thesis, this should be sufficient.

3.1.6 Calculation of screen corners

The camera marks the origin of the camera coordinate system. A transformation between the model coordinate system and the camera coordinate system is found by solving the PnP problem in Section 3.1.4. To express the camera in model coordinates it is moved back into the translation direction, facing the model origin. Since the transformation gained from solving the PnP problem is not taking the distance into account, the translation needs to be adjusted by normalizing it and then multiplying it by the estimated distance. Thus, the camera position $cam \in \mathbb{R}^3$ in model coordinates is

$$cam = \frac{-R^\top T}{\|-R^\top T\|_2} d. \tag{3.17}$$

The screen corners can be calculated by first defining them inside the model coordinate system and then performing almost the same transformation as was done for the camera. To get the first screen corner, the offset between the camera and the top left corner of the screen needs to be measured manually. For example, for a MacBook Pro with a 15-inch screen the top left corner is 17.25 cm left of and 0.7 cm below the camera, so it can be defined as $tl_{\text{model}} = (-0.1725, 0.007, 0)$. From there each other screen corner can be found by adding screen width and height where appropriate, for example, the bottom right corner would be $br_{\text{model}} = (-0.1725, 0.007, 0) + (0.335, 0.207, 0) = (0.175, 0.214, 0)$. The screen corners need to be rotated about the model origin, again using the transposed rotation. After that, they have to be translated by the camera coordinates, since they were defined relative to the camera. So for screen corner tl the transformation is: $tl = R^\top tl_{\text{model}} + cam$.

3.2 Implementation as a software library

To implement the model and make it usable, it is realized in the software library Gaze, which sets the goals to be

- easy to integrate into other projects,
- easy to extend,
- free and open source,
- well documented,
- and available on multiple platforms.

Gaze is written in C++ and compiles into a static library to be easily integrable into other projects. It uses CMake which is used in many C++ projects and thus it should be widely known. C++ was specifically chosen because it can often be integrated into other programming languages since many languages already provide mechanisms to call for example system libraries, which are often written in C or C++. Another reason is that OpenCV and Dlib are natively written in C++, and while both have Python bindings, in general their C++ documentation is much more comprehensible. One important aspect of making Gaze integrable is the Application Programming Interface (API) design. By first recreating an eye tracking experiment (Judd et al. 2009) and finding out what the needs for such an experiment are, Gaze is developed around a very simple API. Extendability is given by a modular design. Gaze builds around a multi-purpose data processing pipeline in which each step performs a small task. It is taken great care to allow for simple extensions using custom steps, for which instructions are provided in Section 4.5.2. One step towards easy extension is also making the source code available for free and as open source software. This way everyone can inspect it, reproduce the results of this thesis and extend Gaze, criticize it, modify it, or build upon it. These are the reasons why publishing the source code and software alongside scientific contributions is very crucial in science (Barnes 2010). To make it easy to do anything of the above with Gaze, it tries to follow many good practices and provides a thorough documentation.

3.2.1 Library architecture

Gaze has three threads which loosely interact with each other if needed through an event system. The first thread is the calling program's main thread. If a program integrates Gaze, it needs to create a `GazeTracker` object and interacts with it. The `GazeTracker` object starts two additional threads. This is done so that the calls from the main program to Gaze are fast and do not interfere with the main program's execution. The second thread is the GUI event thread. This thread is only used if Gaze's debug window is started and uses Dlib's GUI capabilities. The third thread is the most important part of Gaze: The pipeline. The pipeline thread is always started and processes the data in the background by creating a new data object and passing it from one pipeline step to the next. Whenever an object passed the pipeline, an event is emitted to notify the GUI and the main thread. The main thread can then store the latest tracking results to provide seamless access inside feedback loops and the GUI can retrieve the latest data to update its visualizations.

Each pipeline step follows the same interface and has to implement two methods: `void process(util::Data&)` and `void visualize(util::Data&)`. The process method mutates the data object by performing some calculations and storing the result back. By convention, each step should not overwrite the results of other steps, but if two steps perform the same task, this becomes almost inevitable. Each pipeline step's execution time is measured and stored inside the data object. To visualize the data, each pipeline step initializes a GUI widget into which data can be written. The GUI calls the visualize methods only for one step at a time, as it only visualizes one step at a time.

3.3 An alternative approach: iTracker

As an alternative approach to the geometric model presented above, a pre-trained CNN (Krafka et al. 2016) is used for a comparison. It is called iTracker and trained on their dataset GazeCapture, which contains data of 1450 subjects, or about 2.5 million frames. Krafka et al. (2016) make their models publicly available[2] on GitHub. Their network consists of four parts, one per detected eye, one for the face, and a face grid, which is a binary representation of where the face of a subject is located in respect to the original image. All the information except for the face grid is already used inside the geometric model, so it only needs to be adjusted to fit the input layers of the network. ITracker is implemented in Caffe[3] (Jia et al. 2014), which can be integrated into C++ programs. One disadvantage of iTracker is, that it is only trained on iPhones and iPads.

3.4 Datasets

Some datasets were needed during the development and tests for Gaze. While for most ad hoc tests the webcam live stream is enough, it is not enough to allow for reproducibility of the results.

The first dataset, Pexels, is a custom dataset with 120 images from Pexels[4]. The images are rescaled so that all are 640 pixels wide. After resizing, the smallest image measures 640 pixels × 332 pixels, the biggest 640 pixels × 1137 pixels. Most images are portrait photographs using different backgrounds, poses, facial expressions, lighting conditions, and more. The majority of images are color images and contain a single person's face, with few exceptions to this rule, like are partial faces, full body photographs, multiple people, or cats. The people in the images are of different sexes, ages, and colors, but with about 71 the vast majority are young white females. Another 20 people are young white males. Only a handful of people appear to be older than 50, and only about 10 people are of other ethnicities. Less than five people appear in more than one picture. Note that all numbers are only approximate counts to get an idea of the dataset, as the age is always difficult to guess and even sex and skin color can become difficult depending on pose, lighting, or accessories. A few example faces can be seen in Figure C.1. Because the dataset was downloaded for this thesis and is used to evaluate the results of the eye center detection, they are annotated by hand. The annotations and scaled images can be found as supplementary material at the thesis' GitHub repository[5]. A script is provided inside the thesis' code repository to download the original images and perform the annotations.

To compare the pupil detection with the original implementations referenced in Section 5.2.2, the BioID dataset[6] (Jesorsky et al. 2001) is used. It contains 1521 gray images with a fixed resolution of 384 pixels × 286 pixels. The BioID dataset features only 23 different people with multiple images of each. Thirty arbitrary example photos can be found in Figure C.5. The dataset is often used to compare face and pupil detection algorithms (Timm and Barth 2011) and Gaze's implementation is compared to the original implementation by Timm and Barth (2011).

As described in Section 3.3, the iTracker is trained using the GazeCapture dataset (Krafka et al. 2016). It includes photos of 1450 subjects and almost 2.5 million frames. Unfortunately, the download of the raw data is hidden behind a registration, but Krafka et al. (2016) describe the

[2] https://github.com/CSAILVision/GazeCapture
[3] http://caffe.berkeleyvision.org
[4] https://pexels.com
[5] https://github.com/shoeffner/mthesis/releases/download/thesis.shoeffner/PexelsDataset.zip
[6] https://www.bioid.com/facedb

dataset thoroughly and give some example images. The data was recorded using iPhones and iPads and contains images featuring various backgrounds, head poses, accessories, and lighting conditions.

The remaining datasets are those Dlib's shape detectors are based on. One is the "300 Faces In-The-Wild Challenge" dataset, which consists of 600 images with 68 landmark annotations, which were produced semi-automatically. The other dataset is the *dlib 5-point face landmark dataset*, containing 7198 faces. It was labeled using only five landmarks by King. Both datasets are only used indirectly in Gaze, as only the models are used to track facial landmarks and determine the head pose.

Chapter 4

Gaze – A gaze tracking library

Gaze is the software library developed during this thesis. It is supposed to perform gaze tracking in relaxed laboratory conditions. *Relaxed* laboratory conditions means that, unlike for other eye tracking solutions, no special setup or calibration is needed. Gaze works with normal lighting conditions and just a laptop with an attached webcam in front of the participant. It was programmed with many best practices for FOSS in mind and tries to provide a transparent API to track user gaze using a common webcam.

This chapter gives an overview of the library and its features and provides detailed instructions on how to build and use it. The level of detail for the instructions is very high to also give people without a strong background in software development enough information to compile and run it. Because Gaze is FOSS, this should enable many interested researchers to reproduce the results presented in Chapter 5. Or they can try out other methods by using their own configurations and implementations, leveraging Gaze's modular architecture and extendability.

4.1 Features of Gaze

Gaze provides a flexible data pipeline consisting of multiple pre-defined steps. It is possible to add custom steps by altering Gaze's source code in very few places, as is explained in Section 4.5.2. Additionally Gaze features a debug Graphical User Interface (GUI), which visualizes each pipeline step individually. Once implemented, the gaze tracker, the pipeline and the pipeline steps can be configured using a YAML Ain't Markup Language (YAML) file, `gaze.yaml`. The library works best with the camera sensor being in the same plane as the screen surface, thus built-in webcams are recommended. The configuration for the camera position assumes only offsets along and across the screen, the orientation and depth cannot be changed. Gaze can process live webcam streams, video files, and images. The gaze tracker reliably tracks a subject's face and eyes, detects pupils, estimates the head orientation, and also estimates the distance between camera and subject. From this measured and estimated information, Gaze calculates an approximate gaze point. Gaze has been developed using macOS High Sierra, but also builds on Ubuntu 14.04 LTS.

4.2 Free and open-source software

Gaze is FOSS, its source code can be found on GitHub[1]. That means it is publicly available and the source code and software can be modified and redistributed without any limitations. It is released under the MIT License, which is open and permissive: It allows commercial and private use, redistribution, and modification of the source code without any conditions other than keeping the license with the files (Open Source Initiative 2018).

4.2.1 Why free and open-source software?

The decision to release Gaze as a FOSS under the MIT License (Open Source Initiative 2018) was done because the author strongly believes that Open Source and Open Access are important for research. By allowing everyone to use and modify the software and to access the accompanying documentation and thesis, other researchers can verify and reproduce the results. Other individuals can get all available information about the project without having to pay for a license. In case they are interested in the project, they can improve it and contribute their modifications back, or build their own software out of it. Especially due to the nature of thesis projects, it is unlikely that the author will keep working on the project, but potential other authors do not have to start from scratch and can use the code.

Positive side effects of releasing Gaze's source code publicly are that the author tries to follow stricter coding guidelines and provides a detailed source code documentation[2].

4.2.2 Development process

A good practice is to manage code and other projects with a version control system (VCS). Version control allows to roll back changes if needed, retains a change history and, since it can usually be synchronized between multiple devices, provides a simple way to create backups. The VCS used for Gaze is Git[3], which is very popular among software developers: Troy (2017) finds in the stackoverflow Developer Survey 2017[4], that about 70 % of 30,730 responses claim to be using at least Git for version control, followed by Subversion[5] with about 10 %. Of course this data has to be taken into account carefully, as most respondents are in some way users of Stack Overflow[6], a programming related questions and answers website. But the results mean that many people are already familiar with Git and can easily join the project and collaborate without having to overcome high entrance barriers like learning a new VCS.

There are many different ways to structure a Git workflow. One is the GitFlow branching model (Driessen 2010), which largely influenced Gaze's workflow in the beginning. Gaze does not use a specific develop and release branch, instead finished features get pushed to the master branch directly, which makes the process look more like a traditional trunk-based workflow, where all features are developed and pushed onto a common branch, the so-called trunk or master.

Gaze is published on the source code hosting service GitHub[7]. When a new commit is pushed, that means uploaded, to the GitHub servers, a web request is sent to the continuous integration

[1] https://github.com/shoeffner/gaze
[2] https://shoeffner.github.io/gaze
[3] https://git-scm.com
[4] https://insights.stackoverflow.com/survey/2017
[5] https://subversion.apache.org/
[6] https://stackoverflow.com
[7] https://github.com

service Semaphore CI[8]. Semaphore will compile the published version and run the unit tests, which test a few methods for integrity. On success, the commit is considered valid and the changes can be merged into the master branch. To prevent common mistakes and ensure certain code quality guidelines, cpplint[9] is used before pushing a commit to GitHub. It checks if the code conforms to the Google C++ Style Guide[10].

If a commit updates the master branch, Semaphore performs an additional step. It builds the documentation for Gaze and pushes it to a specially named branch, the `gh-pages` branch. This branch is orphaned, which means it has no direct relation to the other source code. GitHub uses this special branch for one of its features: static page hosting. All contents on the `gh-pages` branch are published at a specific URL. This way, the source code documentation is always available online and contains the latest changes. "Always" is a slight simplification, as failures can always happen: GitHub has a service-level agreement (SLA) uptime of 99.95 % for its business customers. Since Gaze is only hosted as a free repository, this SLA does not apply directly, but it is reasonable to assume that the services are available most of the time for free users as well.

4.2.3 License issues

While Gaze is licensed under the MIT License, it cannot be used for commercial applications at the time of writing this thesis, although the license gives this impression. This is because the license of the training data (Sagonas et al. 2016) for the 68 face landmarks' model `shape_predictor_68_face_landmarks.dat` (King 2009), which is used for the face and landmark detection, does not allow commercial uses. On the website accompanying the dataset, it is explicitly stated that "the annotations are provided for research purposes ONLY (NO commercial products)"[11], and King emphasizes in the `README.md`[12] accompanying the model, that the derived model falls under the same license.

A similar notice accompanies Gaze. To avoid problems and allow commercial applications, initially the five landmarks model was tried to be incorporated into Gaze. But the five landmarks selected by King do not perform well to estimate the head pose in 3D as is explained in Section 5.2.3. Thus the 68 landmarks model was chosen, resulting in this license disagreement.

The same licensing problem arises when using the iTracker extension. The pre-trained model is released under a custom license which also does not permit non-research usage, as Khosla states in the repositories "License agreement for use of GazeCapture database and iTracker models"[13]. This usage restriction is surprising: The paper presenting the dataset and the models is called "Eye Tracking for Everyone" (Krafka et al. 2016), a bold claim which does not seem to hold.

4.3 Setting up Gaze

Reproducing the results achieved with Gaze is possible by following a couple of setup steps. The source code should be downloaded, built, and tested using one of the provided example

[8] https://semaphoreci.com
[9] https://github.com/cpplint/cpplint
[10] https://google.github.io/styleguide/cppguide.html
[11] https://ibug.doc.ic.ac.uk/resources/facial-point-annotations
[12] https://github.com/davisking/dlib-models/blob/ae50fe33583de33c60276611d37915e93d11566b/README.md
[13] https://github.com/CSAILVision/GazeCapture/blob/03e687b039a822e7d5bc70673f101def0cba7255/LICENSE.md

programs, or a custom test program. Gaze builds on other free software and thus has some pre-requisites which need to be fulfilled before compiling it. A C++17 compiler is inevitable, as Gaze uses `std::shared_mutex` from the C++ standard library, which was introduced in the C++17 standard[14]. For example on macOS High Sierra this can be Clang 9, which compiles Gaze properly, for Ubuntu 14.04 g++7 works well, as was tested on Semaphore CI's build system. Additional requirements are OpenCV 3.3.1, CMake 3.10.0, boost 1.55.0, and Dlib version 19.8. Usually, different versions of the software dependencies should work fine, but for Dlib version 19.8 is a hard minimal requirement, as it contains a bug fix[15] for a bug discovered during the Gaze's development.

4.4 Building, installing, and using Gaze

Building Gaze just requires two steps from the `gaze` directory as is shown in Code Listing 4.1. The `configure.sh` script in line 1 executes CMake to create the required build files. Afterwards `make` compiles the project. To install Gaze, `make install` can be issued as a followup command; it installs all header files and the library to the proper system locations.

Code Listing 4.1: Building Gaze.

```
1 ./configure.sh
2 cd build
3 make
```

Gaze is a library, so it can not be used alone but only by incorporating it into other programs. A minimal program to integrate Gaze is demonstrated in Code Listing 4.2. The program `gaze_simple` will start the gaze tracker in line 8 and track twenty webcam frames during the `for` loop in lines 9–12. The only additional prerequisite is to have the `shape_predictor_68_face_landmarks.dat` in the same directory as the executable `gaze_simple`. To compile the program, only a short CMake configuration file as shown in Code Listing 4.3 is needed. It configures the compiler to link the executable to Dlib, OpenCV and Gaze by searching the library locations in lines 5–7 and linking against them in lines 10–11. By calling `cmake .. && make` from a `build` directory next to the source file, the program will be built.

4.4.1 Demo programs

The Gaze repository contains two example programs: `simple_tracking` and `where_people_look`. To compile and use them, it is necessary to run the configure step in Code Listing 4.1 again, this time appending the `--examples` option. After building the examples using `make`, two executables can be found in the `build` directory.

The `simple_tracking` program demos the debug GUI shown in Figure 4.1. The debug view can be used to visualize Gaze's various pipeline steps and inspect the computation times each step needs. The tab bar at the top of the window can be used to switch between the different pipeline

[14] http://en.cppreference.com/w/cpp/thread/shared_mutex

[15] https://github.com/davisking/dlib/pull/957

Code Listing 4.2: gaze_simple/gaze_simple.cpp

```cpp
#include <iostream>
#include <memory>
#include <utility>

#include "gaze/gaze.h"

int main(const int, const char** const) {
  std::unique_ptr<gaze::GazeTracker> tracker(new gaze::GazeTracker("1"));
  for (int i = 0; i < 20; ++i) {
    std::pair<int, int> point = tracker->get_current_gaze_point();
    std::cout << point.first << ", " << point.second << std::endl;
  }
}
```

Code Listing 4.3: gaze_simple/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)
project(gaze_simple VERSION 1 LANGUAGES CXX)
set(CMAKE_CXX_STANDARD 17)

find_package(dlib REQUIRED)
find_package(gaze REQUIRED)
find_package(OpenCV REQUIRED)

add_executable(${PROJECT_NAME} ${PROJECT_NAME}.cpp)
target_link_libraries(${PROJECT_NAME}
    dlib::dlib gaze::gaze ${OpenCV_LIBRARIES})
```

step visualizations. The debug GUI uses a tabbed bar to only visualize one step at a time, as drawing many GUI widgets at once is a computationally expensive process. Additionally to the debug GUI, a small window showing the current measured gaze point using a green cross on a black background is shown when using `simple_tracking`. It also implements the main loop of the program: As soon as it is closed, Gaze will close its debug GUI as well and stop tracking.

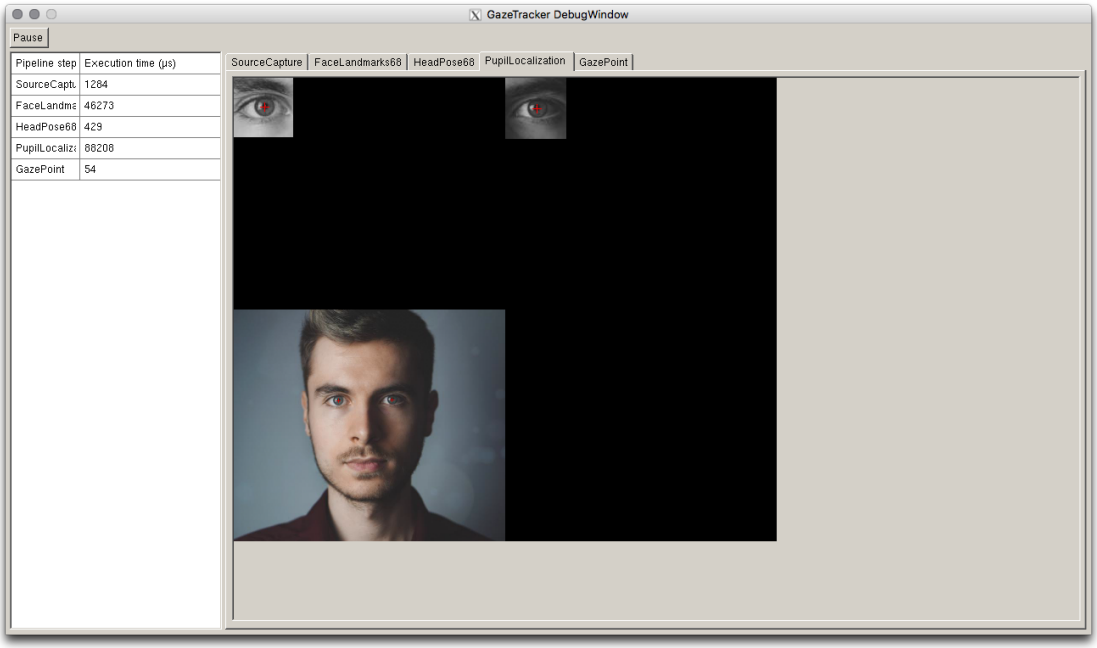

Figure 4.1: Gaze's debug GUI. On the left, the pipeline steps are listed along with their computation times in µs. The tabs can be used to switch between pipeline step visualizations, here the pupil localization is shown.

In their paper, Judd et al. (2009) investigate salient regions of images. The program `where_people_look` is a re-implementation of Judd et al. (2009)'s experiment using Gaze. In the experiment, subjects are presented with 1000 randomly chosen images. Their task is just to freely view each image for three seconds. Each of the images is followed by a gray screen for one second, in which subjects are asked to fixate the center of the screen. In the original task, the experiment was split into two blocks of 500 images each. The re-implementation does not perform this split, as it is mostly used as an example of how Gaze can be integrated into typical experiments. To run the experiment, first the subject identifier must be entered and the directory containing the stimuli needs to be selected. The stimuli are downloaded during the build process. After selecting the stimuli, a gray screen opens up and once the subject is ready, the experiment can be started by pressing the space key.

4.4.2 Gaze's application programming interface

The example program `where_people_look` from the last section was developed before the API for Gaze was settled. This helped to find out which functionality is needed for an eye or gaze tracking

experiment and allowed to design the API in a behavior-driven way. The tasks in Table 4.1 were observed during the implementation of `where_people_look` and implemented in Gaze.

Table 4.1: Functions of the Gaze gaze tracker and their corresponding API methods.

Function	API methods
Initialization	`void init(std::string, bool)` `GazeTracker(std::string, bool)`
Calibration	`void calibrate()`
Trial annotation	`void start_trial(std::string)` `void stop_trial()`
Result storage	None
Latest gaze location	`std::pair<int, int> get_current_gaze_point()`

The initialization of the `GazeTracker` is done by either calling the default constructor followed by its `init()` function, or by using the two argument constructor directly. In Code Listing 4.2, the two argument constructor is called with the subject identifier `"1"` and the default value `false` for the debug flag. The subject identifier will be used to store results on the hard drive. The debug flag decides whether the debug GUI which is shown in Figure 4.1 should be opened by supplying `true`, or not. This is mostly of interest during the development process and not during actual experiments, which is why the default is `false`. If the debug GUI should be shown when starting the program, but not during the experiment, it can be simply be opened by setting the flag to `true`. When the window is no longer needed, it can be closed, without interfering any other windows needed for the experiment and without interfering with Gaze's functionality.

The calibration method is introduced in case Gaze would use some calibration later on, but it is not used at the time of writing this thesis. Because the eye tracking functionality was prioritized during development, the trial annotations have also no effect other than printing a notice to the terminal. They are supposed to write the identifier passed via `start_trial(std::string)` to the result set, to identify which trial was active during the stored measurements. As soon as a writer is implemented, this functionality will work. The result storage is not used as it is supposed handled by a pipeline step. If a pipeline step stores data, it will write the output as soon as it receives the data, thus removing the need for a specific function to trigger a save action.

This leaves Gaze with just one other function than the initialization ones, `get_current_gaze_point()`. This function is especially useful for feedback loops and debugging, as it provides a direct access to the latest result calculated by the Gaze pipeline.

4.5 Configuring and extending Gaze

There are two ways to customize Gaze. The first is to configure it using the `gaze.yaml`, the second is to write a custom pipeline step. In the following section, both options will be briefly

explored to make it easy to integrate with Gaze and to extend it. This should make it easier to adjust Gaze to one's own needs and to contribute pipeline steps to the project.

4.5.1 Configuring Gaze

The `gaze.yaml` can be used to configure Gaze. By creating a `gaze.yaml` file inside the directory from which the program integrating Gaze is executed, the default configuration values can be overwritten. This works because Gaze first loads the default values and then replaces them by any changes made in a potential `gaze.yaml`. This is done to allow customization of parameters without recompiling the source code, thus making it easy to install Gaze once and use it in several different projects with different needs and settings.

The `gaze.yaml` consists of two major parts: The meta configuration block to configure the camera and screen parameters and the pipeline configuration. To get an idea of the default values and to get started writing a custom `gaze.yaml`, the project contains a `gaze.default.yaml`, which exhibits the default values used by Gaze and contains a lot of information about how each parameter affects the program and how it should be chosen. To complement the documentation inside the file, the following two sections give some hints about how to decide the values for which parameters.

Camera and screen parameters

Inside the meta configuration block which is seen in Code Listing 4.4 reside the setup related parameters, that is the camera and screen settings. The screen settings consist of a resolution in pixels and measurements in meters. For example, the laptop used for the development process had a resolution of 2880 pixels × 1800 pixels, and its screen width and height were 33.5 cm and 20.7 cm. Since screen sizes are usually provided in inches across the diagonal, the screen width and height have to be measured manually. Additionally to the screen width and height, the camera position has to be measured. At the time of writing, Gaze only supports cameras which are built into the laptop screen or in the same plane as the screen, and only cameras directed orthogonally away from the screen towards the subject. Thus, the camera position has to be provided using two values, the horizontal offset x from the upper left screen corner, and the vertical offset y from the same corner, here $x = 17.25\,\text{cm}$, and $y = -0.7\,\text{cm}$. Figure 4.2 visualizes which measurements have to be taken. The target parameters allow specifying an area of interest: Instead of mapping the measured gaze coordinates to the screen coordinates, they will be mapped into a regular grid with the dimensions mentioned inside the target parameters.

The camera's resolution can also be configured alongside the target Frames Per Second (FPS). Many webcams are limited in their FPS capabilities, so even by providing high values it is possible that the camera does not reach more than about 30 FPS. For online fixation tracking, this does not matter much, as Gaze is slower than 30 FPS on a common MacBook Pro which is detailed in Section 5.4. Still it means that Gaze will not be able to properly track saccades in online settings because its sampling rate is simply too slow. In offline settings, when analyzing pre-recorded video data, Gaze does not have these problems, as the frame rate is defined by the video material. The computation speed is additionally highly dependent on the image sizes, so smaller resolutions lead to faster computation times. Better hardware, for example, a dedicated GPU to improve Dlib's or Caffe's speeds, might also result in better computation times.

Code Listing 4.4: The default meta configuration block for Gaze. (`gaze.default.yaml`)

```yaml
meta:
  target:
    width: 72
    height: 45
  camera:
    position:
      x: 0.1725
      y: -0.007
    resolution:
      width: 640
      height: 360
    fps: 30
    sensor_size: 0.00635
    sensor_aspect_ratio: 16:9
    focal_length: 0.01
    camera_matrix: !!opencv-matrix
       rows: 3
       cols: 3
       dt: d
       data: [640, 0, 320, 0, 640, 180, 0, 0, 1]
    distortion_coefficients: !!opencv-matrix
       rows: 4
       cols: 1
       dt: d
       data: [0, 0, 0, 0]
  screen:
    resolution:
      width: 2880
      height: 1800
    measurements:
      width: 0.335
      height: 0.207
```

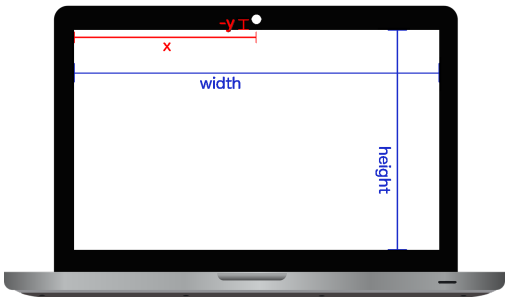

Figure 4.2: To configure Gaze, the display's width and height need to be known. The camera offset to the top left screen corner has to be measured as well, note the positive y values extend downwards, like image coordinates. The notebook is adapted from pixabay.com and under the CC0 license.

Crucial settings for the camera to estimate distances properly are the sensor's size, its aspect ratio, and the focal length. These values are difficult to measure and many device vendors do not report them (Crisp 2013). Apple uses the iSight for their MacBook Pro. Its older versions use a 6.35 mm sensor (Luepke 2005) with an aspect ratio of 4:3. The 15 inches MacBook Pro from mid-2015 used for Gaze's development has a default webcam resolution of 1280 pixels × 720 pixels, which leads to an aspect ratio of 16:9. Since the sensor size is unknown, the best available approximation is to use the old known value, 6.35 mm. It follows that the sensor size is 5.5 mm × 3.1 mm, although it is likely that they use a different size in reality, since the aspect ratio changed. The focal length can be measured manually if it is not provided by the manufacturer; this is shown in Appendix A.2.

Additionally to setting the sensor parameters, the webcam can be calibrated for the use with OpenCV. Calibration, in this case, means to estimate the camera matrix and distortion coefficients of a camera, which can be used to undistort the images. Gaze does not directly undistort the images to process them further, but algorithms like `cv::solvePnP`, which is used by Gaze, benefit from exact values. OpenCV provides a calibration tool which outputs the needed matrices. Detailed instructions on how to use it are inside the appendix in Appendix A.1. Parts of the resulting output file in Code Listing D.1 need to be merged into the `gaze.yaml`, namely the sections `camera_matrix` and `distortion_coefficients`. They need to be placed into the section `camera` inside the `meta` part. An example is already given inside the `gaze.default.yml` file in Code Listing 4.4.

Note that the calibration is not necessary for testing and development purposes, as it is possible to use an estimated camera matrix C without any distortions. According to Mallick (2016), a good approximation can be made using the image width w and its height h. For the example configuration of a 16:9 image with dimensions 640 pixels × 360 pixels, a possible estimated camera matrix using the approximation would be

$$C = \begin{pmatrix} w & 0 & \frac{w}{2} \\ 0 & w & \frac{h}{2} \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 640 & 0 & 320 \\ 0 & 640 & 180 \\ 0 & 0 & 1 \end{pmatrix}. \tag{4.1}$$

Pipeline steps

The pipeline step order, as well as each individual pipeline step, can be configured using various options. This is useful as for example the different implementations of Timm and Barth (2011) can be exchanged without recompiling Gaze by just changing the configuration file. The default pipeline configuration can be found inside the `gaze.default.yaml` as well, it is shown in Code Listing 4.5.

Code Listing 4.5: The default pipeline configuration block for Gaze. (`gaze.default.yaml`)

```yaml
pipeline:
  - type: SourceCapture
    source: 0
  - type: FaceLandmarks
    name: FaceLandmarks68
    model: shape_predictor_68_face_landmarks.dat
  - type: HeadPoseEstimation
    name: HeadPose68
    landmark_indices: [30, 8, 36, 45, 48, 54]
    model: [
      [0.0, 0.0, 0.0],  # Nosetip (Pronasal prn)
      [0.0, -0.0636, -0.0125],  # Chin tip (Gnathion/Menton gn)
      [-0.0433, 0.0327, -0.026],  # Outside right eye (right exocanthion ex_r)
      [0.0433, 0.0327, -0.026],  # Outside left eye (left exocanthion ex_l)
      [-0.0289, -0.0289, -0.0241],  # Right mouth corner (Cheilion ch_r)
      [0.0289, -0.0289, -0.0241]  # Left mouth corner (Cheilion ch_l)
    ]
    model_scale: 1
  - type: PupilLocalization
    relative_threshold: 0.3
  - type: GazePointCalculation
    eye_ball_centers: [
      [-0.029205, 0.0327, -0.0395],  # Right eye ball center
      [0.029205, 0.0327, -0.0395]  # Left eye ball center
    ]
    landmark_indices: [30, 8, 36, 45, 48, 54]
    model: [
      [0.0, 0.0, 0.0],
      [0.0, -0.0636, -0.0125],
      [-0.0433, 0.0327, -0.026],
      [0.0433, 0.0327, -0.026],
      [-0.0289, -0.0289, -0.0241],
      [0.0289, -0.0289, -0.0241]
    ]
```

All pipeline steps are identified by a type, which by convention maps to their C++ class implementation name. The available types at the time of writing are `SourceCapture`, `FaceLandmarks`, `HeadPoseEstimation`, `PupilLocalization`, `EyeLike`, and `GazePointCalculation`.

`SourceCapture` is a step to record input data. The data can be specified using the `source` attribute. If an integer is provided, the corresponding webcam device is used as a live stream. Usually the first camera with device ID `0` is needed, thus this is the default. The setting can also be set to an image or video path, allowing to analyze static images and video files as needed.

The `FaceLandmarks` step employs Dlib's Histogram of oriented gradients (HoG) classifier with a pre-trained model as explained in Section 3.1.2. The model path can be adjusted using the `model` attribute. By default, it is `shape_predictor_68_face_landmarks.dat`, which is the model file downloaded during Gaze's build process. The path has to be specified relative to the model directory.

To estimate the head pose the `HeadPoseEstimation` step is configured with an abstract 3D model of the head. The model is defined using the three parameters `landmark_indices`, `model`, and `model_scale`. The landmark indices are a list of integers corresponding to the landmarks of Sagonas et al. (2016), but with an offset of -1 since Sagonas et al. (2016) use 1-based indexing, while Dlib uses 0-based indexing. The default model uses six landmarks as outlined in Section 3.1.2, but in Section 5.2.3 some other models have been tried.

The steps `EyeLike` and `PupilLocalization` are fully exchangeable since both are implementations of Timm and Barth (2011). `EyeLike` is a copy of Hume's code eyeLike (Hume 2012), hence the name; It was adjusted to fit into Gaze. The two steps have some slight implementation differences. `EyeLike` scales the image patches containing the eyes to a specific size of $50\,\text{pixels} \times 50\,\text{pixels}$, while `PupilLocalization` avoids this. Usually, this means that if a subject sits closer to the camera and thus the eyes are larger, `EyeLike` will perform faster, while `PupilLocalization` is much faster for subjects which are further away from the camera. The precision for both implementations is similar, but depending on whether the scale effects precision or not, either one can outperform the other in certain circumstances. Another implementation difference is that `PupilLocalization` uses a pre-calculated lookup table for some constant values because it was hoped that a lookup table might speed up the process at the cost of some memory. Eventually, the speed did not change much, the image size is a much more important factor as it dominates the algorithm's complexity. A third implementation detail is that `EyeLike` is implemented using OpenCV, while `PupilLocalization` uses Dlib. A very interesting difference is the choice of gradient functions. `EyeLike` uses a gradient function inspired by Matlab (Hume 2012), `PupilLocalization` uses the standard Sobel edge detector as Dlib implements it. For both implementations, the `relative_threshold` can be set. In both steps it is used to discard possible eye center locations if the gradient magnitude at the tested location is below $\mu_{\text{mag}} + \theta\sigma_{\text{mag}}$ with the `relative_threshold` θ, which is detailed in Section 3.1.3. By default the `PupilLocalization` with a relative threshold of 0.3 is used.

The final step of the default pipeline, `GazePointCalculation`, uses a simple model to map the detected pupils onto the 3D head model and performs a ray cast towards the screen as explained in Section 3.1. For this to work, the position of the screen in relation to the head is determined first. As explained in Section 3.1.5, this is done by estimating the direction and distance between head and screen.

4.5.2 Writing a custom pipeline step

To add a new pipeline step to Gaze, a few changes have to be made. The best start is to add some configuration to the `gaze.default.yaml`, it only needs to contain the type: `- type: NewStep`. When Gaze is used with such a "faulty" configuration, the `FallbackStep` will be used. It explains

which changes have to be made to implement a custom pipeline step. The documentation for `PipelineStep` covers the procedure additionally. First, a new header file needs to be created. A template like Code Listing 4.6 is provided inside the documentation. For the new pipeline step, the names have to be adjusted in lines 1, 2, 13, 18, and 29 to represent the new type chosen above. In line 15 the visualization type can be changed, allowed choices are `Label`, `Image`, and `Perspective` visualizations. The choice determines which widget is used for the pipeline step inside the debug GUI. The last change inside the header file is to add documentation for the `process` and `visualize` methods. After that, a new implementation file, here `new_step.cpp`, has to be added and referenced inside the `src/gaze/CMakeLists.txt` as an additional source file. Similarly, the header file must be included in `include/gaze/pipeline_steps.h`. The last step is to add a case to the `init_pipeline()` function inside `src/gaze/gaze_tracker.cpp` which is seen in Code Listing D.2. Now the new step is readily available.

Code Listing 4.6: The template header for a new pipeline step. The names are modified to match the file name, so `MY_STEP` becomes `NEW_STEP` in the example. (`pipeline_steps/new_step.h`)

```
1  #ifndef INCLUDE_GAZE_PIPELINE_STEPS_NEW_STEP_H_
2  #define INCLUDE_GAZE_PIPELINE_STEPS_NEW_STEP_H_
3
4  #include "gaze/gui/visualizeable.h"
5  #include "gaze/pipeline_step.h"
6  #include "gaze/util/data.h"
7
8
9  namespace gaze {
10
11 namespace pipeline {
12
13 class NewStep final
14     : public PipelineStep,
15       public gui::LabelVisualizeable {
16
17  public:
18   NewStep();
19
20   void process(util::Data& data) override;
21
22   void visualize(util::Data& data) override;
23 };
24
25 }  // namespace pipeline
26
27 }  // namespace gaze
28
29 #endif  // INCLUDE_GAZE_PIPELINE_STEPS_NEW_STEP_H_
```

4.5.3 GazeCapture

`GazeCapture` is one custom pipeline step which is more complicated. Instead of the few changes in Section 4.5.2, it requires additional compilation steps and depends on Caffe 1.0. Since it is also not completely straightforward to compile it and still crashes from time to time due to some memory access errors, it is disabled by default. It can be enabled by configuring `Gaze` using CMake by setting the flag `-DWITH_CAFFE=ON`. The build process will then fetch Caffe and compile it so that `Gaze` can link against it and it can be employed inside the pipeline. When `GazeCapture` is used inside the pipeline, it requires only the `SourceCapture` and `FaceLandmarks` pipeline steps to come before it. It performs predictions using the iTracker CNN outlined in Section 3.3.

Chapter 5

Results, evaluations and comparisons

As shown in Chapter 4 the Gaze library achieves its goals. It is able to process data in a timely manner and solves eye center tracking extremely well. One of its shortcomings is the model to transform the detected pupils from the 2D image into the 3D model. In the following sections, the shortcomings and successes of Gaze are quantified and qualified. Near the end of the chapter a comparison with iTracker was planned, but since the gaze points cannot be calculated properly using the geometric model, the comparison is limited. Instead, a brief qualitative review of iTracker will be done.

5.1 Library implementation

Gaze's goals to be easily integrable, extendable, to be FOSS, well documented, and cross-platform are mostly fulfilled. While it is sufficiently easy to integrate Gaze into software as shown in Section 4.4, it is not possible to extend Gaze with a custom pipeline step without recompiling it. It would have been a bigger success if Gaze truly followed a multi-purpose plugin architecture to quickly prototype custom pipeline steps and test new models. But because Gaze achieves its goal to be FOSS, it is possible to change that in the future – and even gather feedback if that is really a useful addition. The documentation is available and automatically built using Semaphore and thus always readily available. The cross-platform usage is not thoroughly tested, but Gaze builds successfully on macOS and Ubuntu 14.04 and its tests pass on both those platforms as well. The software fulfills many of its goals and is ready to use for eye tracking in feedback loops. For gaze tracking in research settings, however, some shortcomings detailed below need to be resolved first.

5.2 Evaluation of the geometric model

The geometrical model proposed to track gaze is not working as expected. Each individual part will be evaluated in the next sections to explain which ones are working and which ones are not.

5.2.1 Face detection

The face detection works reliable on webcam images: Dlib's face detector finds the faces in all images of the BioID dataset. The Pexels dataset is more difficult: Dlib only detects faces in 80 out of 120 images. Figure C.7 shows the discarded 40 images. More than half of those 40 images contain faces which are either visible from the side or which have an occluded half. Four faces feature uncommon sunglasses, two photos feature only eyes, and two others only cats. Only a handful of faces are properly visible and many of those are not upright. So the limitations of Dlib's face detector are mostly occluded faces and faces in uncommon orientations, which is okay for the purpose of eye and gaze tracking.

While missing some faces, Dlib has a much lower false positive rate than OpenCV (King 2014), which is another option for face detection. Processing a 640 pixels × 360 pixels image took about 130 ms to 140 ms using OpenCV and about 160 ms to 170 ms using Dlib. But factoring in the advantage of also getting the 68 landmarks needed for the head pose estimation during Dlib's processing time strengthens the choice of Dlib over OpenCV.

5.2.2 Pupil localization evaluation

One very successful part of Gaze is the pupil center localization (Timm and Barth 2011). Using the BioID dataset (Jesorsky et al. 2001) and the *relative error* introduced by Jesorsky et al. (2001) the accuracy of Gaze's `PupilLocalization` can be benchmarked. The relative error is defined as

$$e_{\text{op}} = \frac{\text{op}\left(d_l, d_r\right)}{d_p}, \tag{5.1}$$

with d_l, d_r the Euclidean distance between the left eye center and its estimation, and the right eye center and its estimation, respectively. d_p denotes the Euclidean distance between the real eye centers. The operator op is either min, max, or $\text{mean}(x, y) = \frac{x+y}{2}$. Depending on the operation, different results are measured. If min is used, the accuracies only take the result of the better eye into account, likewise the worse eye is used for max. The average error of both eyes is calculated using the mean for the error measurement. In Figure 5.1 `EyeLike`, Timm and Barth (2011), and `PupilLocalization` are compared using the max relative error, which makes a lot of sense for accurate eye tracking. To measure the accuracy, the error e_{max} is calculated and the percentage of faces for which the error is below or equal the thresholds is reported. A comparison of all three different errors, min, max, and mean, can be found in Table B.2.

The accuracy of `PupilLocalization` is very good and reaches the same accuracy as Timm and Barth (2011). Inspecting the times in Table B.4 it becomes clear that `PupilLocalization` performs faster than the implementation of `EyeLike` on the BioID dataset with a median computation time of 1.810 ms, compared to `EyeLike`'s median computation time of 25.893 ms across the whole dataset. Even the maximum computation time is still on par. Still, since the eye size is a quadratic factor in the implementation, the computation time of `PupilLocalization` changes dramatically with bigger images, while `EyeLike`'s computation times are relatively stable. This can be seen in Table B.3 for the Pexels dataset, which contains bigger images: The median computation time is still better for the whole dataset, but while inspecting only results where both eye patches have a side length of more than 50 pixels, the size to which `EyeLike` scales the eye patches to, `EyeLike` outperforms `PupilLocalization` by a factor of about 4.

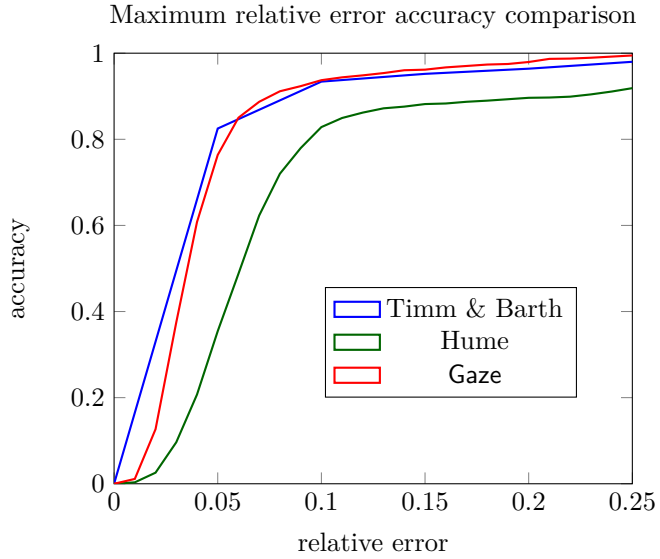

Figure 5.1: Comparison of pupil detection accuracy between Timm and Barth (2011), Hume (2012) and Gaze's `PupilLocalization` on the BioID dataset. Only the maximum relative error is shown, that is the worse eyes are taken into account. Refer to Table B.2 for a tabular version of all relative errors.

It can be concluded that for real-time scenarios with today's image resolutions `EyeLike` is the better choice, as it is faster with only a small loss in performance. For accurate processing when time is of no critical importance, for example during offline analysis of recorded video data, Gaze's `PupilLocalization` is usually the better option. In general, it should be noted, that with `PupilLocalization` and eyeLike (Hume 2012) there are two successful replications of the eye center detection approach by Timm and Barth (2011). Comparing the different accuracies per dataset which are listed in Table B.1 and Table B.2, it becomes clear that the algorithm works better with the images from the BioID dataset, but still performs reasonably well on the Pexels dataset.

5.2.3 Head pose estimation

The head pose estimation works well in most cases. In the extreme case when the nose points directly towards the camera and the coronal plane is parallel to the screen plane, that is if a subject is looking straight into the camera, there are two equally likely possible solutions: the z axis pointing outwards or inwards. In most cases, this is not a problem, as it is unlikely that this happens. But sometimes this causes different results from frame to frame as the z axis might change its direction in between. Since there is no annotated ground truth containing head poses for the datasets used it is not possible to give a quantitative analysis of the success of the method, but as was lined out in Section 3.1.4 a qualitative analysis can be performed. In Section 3.1.4 it is already established that the EPnP version of the `solvePnP` algorithm does not work as well as the iterative version, thus here the difference between the five and the 68 landmarks models will be shown.

The problem with the 68 landmarks model is, as stated in Section 4.2.3, that it is only allowed for research usage. As an alternative, the five landmarks model by Dlib was tried using a custom 3D head model. Unfortunately, the 3D head model used in Gaze (Mallick 2016) relies on the six landmarks pronasal, gnathion, exocanthions left and right, and the cheilions left and right, which are not all present in the five landmarks model. It uses the two exocanthions, the subnasal – the point below the nose – and the endocanthions. To be able to estimate a head pose, a different 3D model is used, Model A in Code Listing 5.1. It is difficult to find accurate relational measurements for these landmarks. The endocanthions are offset towards the center by the length of the palpebral fissure, similar to the eyeball centers. The subnasal is located using an educated guess backed by the data about the nasal height and the philtrum length (Weinberg and Marazita 2009).

Code Listing 5.1: The two 3D head models used for the five landmarks comparison.

```
# Model A
landmark_indices: [4, 3, 0, 2, 1]
model: [
  [0.0, -0.0055, -0.0125],  # subnasal
  [-0.0433, 0.0327, -0.026],  # exocanthion right
  [0.0433, 0.0327, -0.026],  # exocanthion left
  [-0.01511, 0.0327, -0.026],  # endocanthion right
  [0.01511, 0.0327, -0.026],  # endocanthion left
]

# Model B
landmark_indices: [4, 3, 0, 2, 1]
model: [
  [0.0, -0.0055, -0.0125],  # subnasal
  [-0.0433, 0.0327, -0.026],  # exocanthion right
  [0.0433, 0.0327, -0.026],  # exocanthion left
  [-0.01511, 0.0327, -0.016],  # endocanthion right
  [0.01511, 0.0327, -0.016],  # endocanthion left
]
```

As can be seen in Figure C.4, this model is not sufficient. The reason is that four of its five points are located on the same line. Thus the vectors can no longer span three linearly independent vectors. A few adjustments are tried to resolve this issue, for example using Model B in Code Listing 5.1 which just moves the endocanthions about a centimeter outwards of the face. For some faces like 0031 in Figure C.4 this works slightly better, for others like 0044 worse. It remains unclear whether the five landmarks can be used with a proper model to estimate head poses accurately, or if the points really do not contain enough information to properly span three dimensions between them.

5.2.4 Gaze point estimation

The gaze point estimation consists of multiple parts. The first part is the distance estimation, followed by the inverted projection of the detected pupil centers into the model coordinates. The final step is the ray cast to determine the gaze point locations.

The distance estimation is only using the outer canthal width, thus this part is very likely to be highly inaccurate. No precise tests were performed during the evaluation, a short measurement using a folding rule was conducted instead. To test the distance, only very strict face poses rotated towards the camera were considered. Variations of about 5 cm to 10 cm are expected. Using the focal length of 10 mm established in Appendix A.2 this cannot be achieved. Gaze estimates the distance about twice further than it is. One possible explanation is that the focal length in Appendix A.2 is wrong. Given the fact that the MacBook Pro's display is only about half a centimeter deep, it seems to be a better idea to use 5 mm instead. This way the overestimation by a factor of two is canceled and the estimates fall into the right range. Another possible explanation is the video resolution. In Gaze's experiments, a resolution of only 640 pixels × 360 pixels is used, which is only half of the native webcam resolution. A third explanation might be that the sensor size assumption established in Section 4.5.1 is invalid, and the newer MacBook's indeed use a different camera. Testing which of the problems accounts for the faulty distance estimation is left open for the future.

The reprojection of the pupil centers turns out to be Gaze's biggest issue. While the pupil centers appear to end up in roughly the correct region as can be guessed from Figure 3.3, their location is not accurate enough. Presumably the issue is that due to the transformation from a flat 2D projection into a 3D model the information which is lost during the original 3D-2D projection is not approximated well enough. This is likely a problem of the model. A model worth testing instead might be to first perform an orthogonal projection into the direction of the screen, fit the pupils into the projection and then perform a ray cast from the pupils onto modeled eyeballs. This would resolve an error which likely occurs in the current model: By not projecting the pupil centers properly onto the eyeball, they are likely displaced in relation to their true position. The effect is visualized in Figure 5.2 using a 2D simplification: It can easily be seen that even slight variations of a few millimeters from the correct p_0 can lead to huge errors on the screen.

Because of these reconstruction problems, the ray casting also produces wrong results – although for the given inputs the results are correct. An alternative to using a different model might be to introduce a calibration method, which Gaze tries to avoid.

To test how well Gaze performs even with a faulty model it can be considered to see how well it can distinguish between people looking to the left from people looking to the right. Unfortunately this turns out to be difficult to test using the datasets already in use: Neither of the datasets contains annotations whether the person is looking to the left or right. When considering annotating the small Pexels dataset it quickly turns out that due to the nature of portrait photos, most people gaze directly into the camera. In fact, only 18 images feature obvious gazes to either side. The same holds for the BioID dataset, in most pictures the people also gaze into the camera. Due to time constraints, no further selection of images featuring prominent gazes to either side has been done.

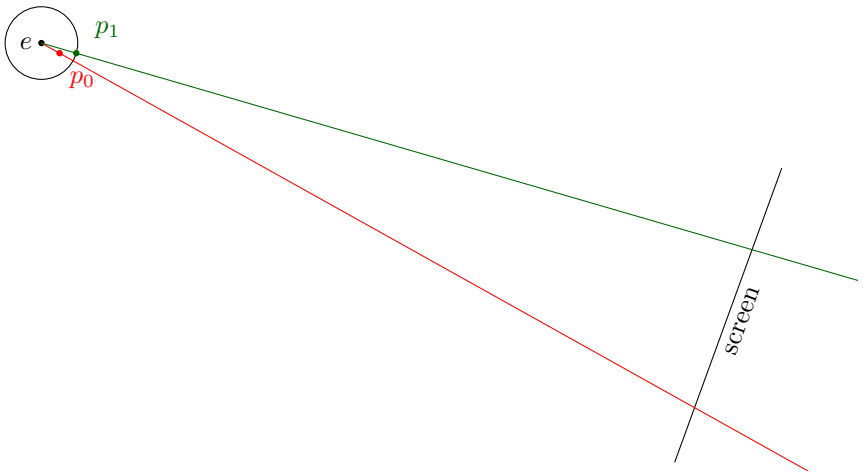

Figure 5.2: A small pupil restoration error of using p_0 instead of p_1 for the ray cast from e to the screen can lead to big errors on the screen surface.

5.3 Comparison with and brief review of iTracker

As outlined in the beginning of this chapter and Section 3.3, originally a comparison between the geometric model and the CNN was planned. Now that the geometric model is only capable of doing eye tracking, the comparison is simple: iTracker works well for gaze tracking, but not for eye tracking, while the geometric model works well for eye tracking, but not for gaze tracking. Hence, the following is a small review of iTracker's qualitative performance and its usability.

The custom pipeline step `GazeCapture`, designed to replace most parts of the default pipeline and employing the CNN iTracker (Krafka et al. 2016), performs well up to some limitations.

Integrating iTracker into Gaze is not an easy task as its dependency Caffe and Gaze's dependency Dlib both depend on Basic Linear Algebra Subprograms (BLAS), and there are multiple different implementations of BLAS available. While most libraries offer bindings to multiple variants, it is only possible to use one version of BLAS in a final program. Thus Caffe and Dlib need to be compiled using the proper bindings first before they can be used together in iTracker. To build a custom program using Gaze with the `GazeTracker` pipeline step requires writing a CMakeLists file which first finds iTracker before searching for Gaze. When using iTracker, Gaze occasionally crashes due to some memory errors. It is not clear whether it is the fault of Gaze or if the problem is a resource management within Caffe. But because this crash is not resolved, `GazeCapture` is currently only an optional component.

Taking the technical issues aside, iTracker is able to estimate gaze inside a limited scope. Because it is trained on data measured exclusively on iPhones and iPads (Krafka et al. 2016), it has a strong bias towards calculating gaze points within the boundaries of those screens. This bias is visualized in Figure 5.3, which is a log log visualization of the estimated gaze points on the Pexels and the BioID dataset. Of course, as discussed in Section 5.2.3, a high proportion of people are looking directly into the camera, which of course leads to a natural aggregation around the top middle area of the image, directly below the camera. Still only very few points can be observed outside the boundaries of an iPhone or iPad screen in relation to the camera.

Figure 5.3: Double-logarithmic visualization of the estimated gaze points on the screen using `GazeCapture`'s iTracker for the BioID dataset on the left and the Pexels dataset on the right. Brighter means more gaze points.

Overall iTracker is a convincing implementation of a CNN for gaze tracking, but a qualitatively broader training dataset might help improving it even further to make it feasible in tracking gaze in other settings than on mobile screens. While its computation times are slightly slower than the default pipeline used in Gaze as is seen below, this is likely due to the fact that the test laptop does not have a dedicated Graphics Processing Unit (GPU) to use Caffe's full potential. This is something to test in the future.

5.4 Computation times

For real-time gaze point estimations, fast computation times are a very critical metric. Gaze performs relatively fast for small eyes. The most time-consuming parts are the face detection and the eye center localization as can be seen in Table 5.1. The other pipeline steps, especially the head pose estimation and the final ray cast are very fast. The computation times of the `SourceCapture` step were not measured as it was not used during the benchmarks because it is not capable of loading a series of images unless they are frames of a video file.

Table 5.1: The different computation times per pipeline step, measured on the pexels dataset. All values are in µs. Note that all pipeline steps except for the source capture are included, even though the default pipeline only uses the first four.

Pipeline step	Median	Max	Mean	Min
`FaceLandmarks`	39 694	92 488	49 031	29 581
`HeadPoseEstimation`	359	1677	391	206
`PupilLocalization`	18 028	1 992 450	136 196	133
`GazePointCalculation`	30	74	33	28
`EyeLike`	27 708	36 537	27 934	19 044
`GazeCapture`	39 972	1 767 480	61 912	38 398

Without taking the `SourceCapture` into account, the default pipeline is usually the fastest pipeline. In cases where it needs to resize the lookup table for the pupil detection step, it slows down a little bit as observed in Section 5.2.2. But even the slowest pipeline using `GazeCapture` is still similarly fast, as can be seen in Table 5.2. It might be faster if a speedup is gained by using a GPU as theorized above. But even with the slightly slower speed iTracker's CNN is the best model currently implemented in Gaze.

Table 5.2: Accumulated computation times per pipeline configuration, measured on the pexels dataset. The `SourceCapture` step is excluded. The other steps are `FaceLandmarks` (F), `HeadPoseEstimation` (H), `PupilLocalization` (P), `EyeLike` (E), `GazePointCalculation` (G), and `GazeCapture` (C). All values are in µs.

Pipeline	Median	Max	Mean	Min
Default (FHPG)	63 846	2 078 141	185 651	35 219
EyeLike (FHEG)	70 142	121 546	77 389	55 375
iTracker (FC)	79 875	1 820 336	110 943	69 975

The total times for a pipeline in Table 5.2 are about 80 ms, which means that the pipeline can process at most 12 frames in one second. The real number will be lower, as grabbing the image from the webcam is not included in the measurements. This makes Gaze a good choice for non-time critical applications and offline data processing. During online settings, it will only detect fixations lasting at least 160 ms properly – saccades or even microsaccades are too short for Gaze to process. In offline analysis, this can of course still work if the source material's FPS were high enough to detect the relevant features of gaze.

5.5 Conclusion

Gaze tracking using common webcams is definitely no solved problem. While it earned more attention in the literature over the last few years, few solutions exist that really work without many constraints. In this thesis, two possible methods are evaluated: The geometric model, realized in the Gaze library, and the CNN iTracker, a pre-trained model which can be used instead of the geometric model.

Gaze is a good library to perform eye tracking. It can easily be extended and integrated into projects. Its gaze tracking capabilities are not working properly, but thanks to its extendability it is possible to use iTracker to detect gaze points. Many other issues in webcam gaze tracking cannot be resolved using software alone: most webcams are limited to small frame rates and it is extremely difficult to find the real specifications like sensor sizes and focal lengths for many webcams.

While there are some other solutions to webcam eye and gaze tracking, many of the projects are abandoned, need sophisticated calibration procedures or reach low precision. The CNN iTracker and similar approaches seem to be promising models to make gaze tracking available for everyone and for all devices, but there is still some work to do, like increasing the area of interest and reducing computation times. But concerning the possibility of gaze tracking without additional equipment and considerable cost it warrants further investigation.

Chapter 6

Future work

After concluding that Gaze is a useful software for eye tracking but not so much for gaze tracking in the last chapter, this chapter presents a non-exhaustive list of possible work to explore in the future. It will also very briefly discuss the pre-trained models used in the thesis and how to improve those in the future, and provide an outlook of what to expect next.

6.1 Possible fixes and extensions for Gaze

Gaze has some issues which need attention. Mostly this is related to the geometric model used to calculate the gaze points. The affine transformation used to reverse the 2D projection to the 3D model and to transform the pupils should be re-evaluated and possibly replaced. One way could be to model the eyeballs as spheres or ellipsoids and perform more sophisticated projections onto their surfaces. Something not taken into account at all in Gaze's models is the refraction of light in the eye. Some of the models mentioned in Hansen and Ji (2010) could be of use. Another possibility is to introduce a calibration method and see if the model works using a setup calibrated to the subject. This might be difficult because of the pipeline architecture, but custom steps or a general extension to the pipeline are possibilities to solve this.

One additional possible shortcoming of Gaze is that it relies on both, Dlib and OpenCV heavily. Because of that, oftentimes conversions between the data formats are needed, potentially slowing down the computations. While first the algorithms themselves should be improved before thinking about such details, it is something to keep in mind if working on Gaze.

The way the configuration works is also worth improving. In the current implementation, the 3D model has to be supplied multiple times inside the configuration because the pipeline steps do not share their information. It might be a good idea to aggregate shared information in one block or define a different configuration format in general, to avoid redundancies, as those can easily lead to errors.

Apart from these issues, there are also many possible extensions to make for Gaze. One is to implement result writes and readers. Writers give the ability to store data and analyze it, readers allow to continue processing a video file or to reevaluate data using a different method. When implementing writers, it might also prove useful to determine what other information needs to be

stored inside the data object; for example, the frame number is currently not stored but might be very useful for video analysis. Although there are numerous different data formats for eye tracking data (Schöning et al. 2016), Gaze's modular approach allows for easy implementation of various formats.

Because the current implementation only allows for cameras located on the screen plane, a second interesting extension to Gaze would be to make this more flexible. This is no easy task as the conversions then need to account for four additional dimensions – not only offsets in two directions but offsets in three directions and a three-dimensional orientation. But this would allow having setups with external webcams much easier than with the current approach.

In terms of implementation, Gaze has a couple of things to improve on. One example is to provide bindings to other languages – to use Gaze in for example Python, but also to write pipeline steps in other languages than C++ and use them within Gaze. The former could be realized using frameworks like Boost.Python[1] or pybind11[2], the latter by building the pipeline around a plugin system. Additionally many parts of the code do not have proper tests yet. While there are some tests for the `PupilLocalization` pipeline step, most other parts of the code are just tested manually by visual inspection using the GUI. This should be improved, as that also improves the effectiveness of the continuous integration, as it is more likely to warn of broken builds.

6.2 Thoughts on the data and pre-trained models

To avoid the license disagreements between the different models used in Gaze, a good idea is to search for other models with more permissive licenses or train custom models which do not have those limitations. One of such alternative models could be the five landmarks model provided by Dlib, and since it is still unclear if those points are enough to determine the head pose, it is an option to explore. To train custom models the biggest challenge is to come by properly annotated data, but also to keep models small and efficient.

As was seen in Section 5.2.1, face detection is clearly not a completely solved task yet. Even models trained on the "300 Faces In-The-Wild challenge" do not recognize occluded faces, especially if occluded by shades or if only half of the face is visible. Of course, it is a little bit of a definition issue and depends on the question if one wants to detect occluded faces or not, but in general, this is still not resolved. And even if only non-occluded faces should be detected, tilted faces also still pose challenges. Thus improving on face detections is still an interesting task. Of course, for eye tracking in lab conditions the current detectors work well enough, even exceeding their needs.

6.3 Closing remarks

Eye and gaze tracking are an exciting field of research. Over the last century, significant progress has been made, providing many sophisticated solutions to track eyes and gaze in various ways. Many of those solutions require difficult setups, are expensive, require specific hardware, or their implementations are not available as open source solutions. Gaze tries to fit into this landscape by providing an FOSS solution which does not require any difficult setup and no specialized

[1] http://www.boost.org/doc/libs/1_66_0/libs/python/doc/html/index.html

[2] https://github.com/pybind/pybind11

hardware. While it does not reach the same gaze tracking performance like other solutions, its eye tracking capabilities are very promising. It will be interesting to see if geometric models like the one in Gaze, or machine learning model like iTracker will provide tools to bring gaze tracking to a broad community in the future – and if webcams will be a useful tool to do gaze tracking in general; inside academia, and outside. For studies which do not need saccade information it might prove useful, as well as for computer control and user experience research, and many other areas, some of which are unthinkable of today.

Appendices

Appendix A

Additional information

This chapter contains information to help reproduce the results of Gaze but are not important enough to appear in the main text.

A.1 Calibrating OpenCV

OpenCV offers a tool to store camera calibration settings which can be used for certain functions to improve their results. Gaze benefits from such functions, among others it uses `solvePnP` which is used to estimate the head pose in Section 3.1.4. This section explains briefly how to use the calibration tool. Please note that there is also a tutorial[1] available which provides some videos and a more thorough explanation of the mathematics.

The calibration tool can be found in OpenCV's samples, samples/cpp/calibration.cpp[2]. To be able to use it, OpenCV needs to be compiled manually by providing the CMake flag `-DBUILD_EXAMPLES=ON` to build the `cpp-example-calibration` executable. To calibrate the camera, the calibration tool needs to take a couple of pictures of a benchmark image: a checkerboard pattern as in Figure C.6 is used in the following. Calibration works best if the image is on a hard surface, like cardboard. An example call to the calibration tool is denoted in Code Listing A.1. The parameters `-h=6` and `-w=9` describe the layout of the checkerboard pattern. It means that the checkerboard is seven squares down and ten squares across since the parameters expect the numbers of corners between four squares. `-n=10` is the number of images to be taken, `-d=1000` is the delay between two images. A higher delay allows that during calibration the image can be moved to more divers poses without triggering another image, resulting in a higher variety of points which in turn leads to a more exact estimation of the camera parameters. At least three pictures should be taken, but more images provide better results. The output file to which the calibration values are written is stored in the file passed with `-o`. The last parameter, `-s=0.0015` is the size of one checkerboard square in meters. This value should be measured on the printout of the checkerboard, as slight variations can occur depending on page orientation, zoom levels, margins, printer settings, and other factors. In the example, the printed version's squares' side

[1] https://docs.opencv.org/3.0-beta/doc/tutorials/calib3d/camera_calibration/camera_calibration.html

[2] https://github.com/opencv/opencv/blob/fc9e031454fd456d09e15944c99a419e73d80661/samples/cpp/calibration.cpp

lengths were 1.5 mm. After a successful calibration, `camera_calib.yml` will be written into the directory. It can be used to configure Gaze, as explained in Section 4.5.1.

Code Listing A.1: Using the OpenCV calibration tool to calibrate the camera.

```
./cpp-example-calibration -h=6 -w=9 -n=10 -d=1000 -s=0.0015 -o=camera_calib.yml
```

A.2 Determining the focal length

To find the effective focal length of a camera the angle of view needs to be measured, and the sensor size has to be known. For Gaze's examples the sensor width is assumed to be 0.0055 m. After measuring the angle of view, it can be used to solve (Wikipedia contributors 2017b)

$$\alpha = 2 \arctan \frac{d}{2f} \tag{A.1}$$

$$f = \frac{d}{2 \tan \frac{\alpha}{2}}, \tag{A.2}$$

with d being the sensor size, either 0.0055 m for horizontal calculations, or 0.0031 m for vertical one. These values are taken from Section 4.5.1. α is the angle of view and f is the focal length. To find α the camera is placed parallel to a wall, facing it. Then the distance w between the left most and the right most points which are still visible on the camera image, and the distance between the camera and the wall v are measured for the horizontal calculations. For applying this to the vertical case, the top and bottom most points need to be used. Using trigonometry the angle of view can be calculated by substituting the values into

$$\alpha = \arctan \frac{w}{2v}. \tag{A.3}$$

For the examples in Gaze the focal length used is 0.01 m, which is the approximate mean of the measured values for the horizontal and vertical focal lengths (f_h, f_v), measured using a folding rule at a distance of 1.04 m for a MacBook Pro:

$$f_h = \frac{0.0055\,\mathrm{m}}{2 \tan\left(\frac{1}{2} \arctan\left(\frac{w_h}{2v}\right)\right)} = \frac{0.0055\,\mathrm{m}}{2 \tan\left(\frac{1}{2} \arctan\left(\frac{1.13\,\mathrm{m}}{2 \cdot 1.04\,\mathrm{m}}\right)\right)} \approx 0.011\,\mathrm{m} \tag{A.4}$$

$$f_v = \frac{0.0031\,\mathrm{m}}{2 \tan\left(\frac{1}{2} \arctan\left(\frac{w_v}{2v}\right)\right)} = \frac{0.0031\,\mathrm{m}}{2 \tan\left(\frac{1}{2} \arctan\left(\frac{0.66\,\mathrm{m}}{2 \cdot 1.04\,\mathrm{m}}\right)\right)} \approx 0.01\,\mathrm{m} \tag{A.5}$$

$$f = \frac{f_h + f_v}{2} = \frac{0.011\,\mathrm{m} + 0.01\,\mathrm{m}}{2} = 0.0105\,\mathrm{m} \approx 0.01\,\mathrm{m}. \tag{A.6}$$

Appendix B

Tables

Table B.1: Different accuracies per relative error thresholds on the Pexels dataset.

Error type	Method	0.05	0.10	0.15	0.20	0.25
max	`EyeLike`	0.350	0.650	0.800	0.875	0.912
	`PupilLocalization`	0.713	0.900	0.925	0.963	0.963
mean	`EyeLike`	0.438	0.750	0.850	0.938	0.950
	`PupilLocalization`	0.800	0.925	0.975	0.988	0.988
min	`EyeLike`	0.588	0.850	0.912	0.938	0.963
	`PupilLocalization`	0.950	0.975	0.988	0.988	0.988

Table B.2: Different accuracies per relative error thresholds on the BioID dataset. While Timm and Barth (2011) provide a graph of their results, only their max values are reported.

Error type	Method	0.05	0.10	0.15	0.20	0.25
max	`EyeLike`	0.354	0.828	0.882	0.896	0.919
	`PupilLocalization`	0.764	0.937	0.962	0.980	0.995
	(Timm and Barth 2011)	0.825	0.934	0.952	0.964	0.980
mean	`EyeLike`	0.524	0.863	0.914	0.968	0.982
	`PupilLocalization`	0.870	0.963	0.992	1.000	1.000
min	`EyeLike`	0.702	0.960	0.974	0.977	0.987
	`PupilLocalization`	0.949	0.991	0.997	1.000	1.000

Table B.3: Comparison of computation times between `EyeLike` and `PupilLocalization`. Data measured on the Pexels dataset.

Computation times [µs]	Eye size	`PupilLocalization`	`EyeLike`
Median	≤ 50	2178	29 562
	> 50	99 510	26 347
	all	18 028	27 708
Min	≤ 50	133	19 928
	> 50	14 018	19 044
	all	133	19 044
Mean	≤ 50	3827	29 529
	> 50	273 999	26 223
	all	136 196	27 934
Max	≤ 50	14 465	34 534
	> 50	1 992 450	30 834
	all	1 992 450	36 537

Table B.4: Comparison of computation times between `EyeLike` and `PupilLocalization`. Data measured on the BioID dataset.

Computation times [µs]	Eye size	`PupilLocalization`	`EyeLike`
Median	≤ 50	1706	25 838
	> 50	20 769	27 091
	all	1810	25 893
Min	≤ 50	123	17 488
	> 50	14 462	21 711
	all	123	17 488
Mean	≤ 50	2532	26 074
	> 50	21 597	27 232
	all	3405	26 134
Max	≤ 50	20 596	37 822
	> 50	40 784	35 687
	all	40 784	37 822

Appendix C

Figures

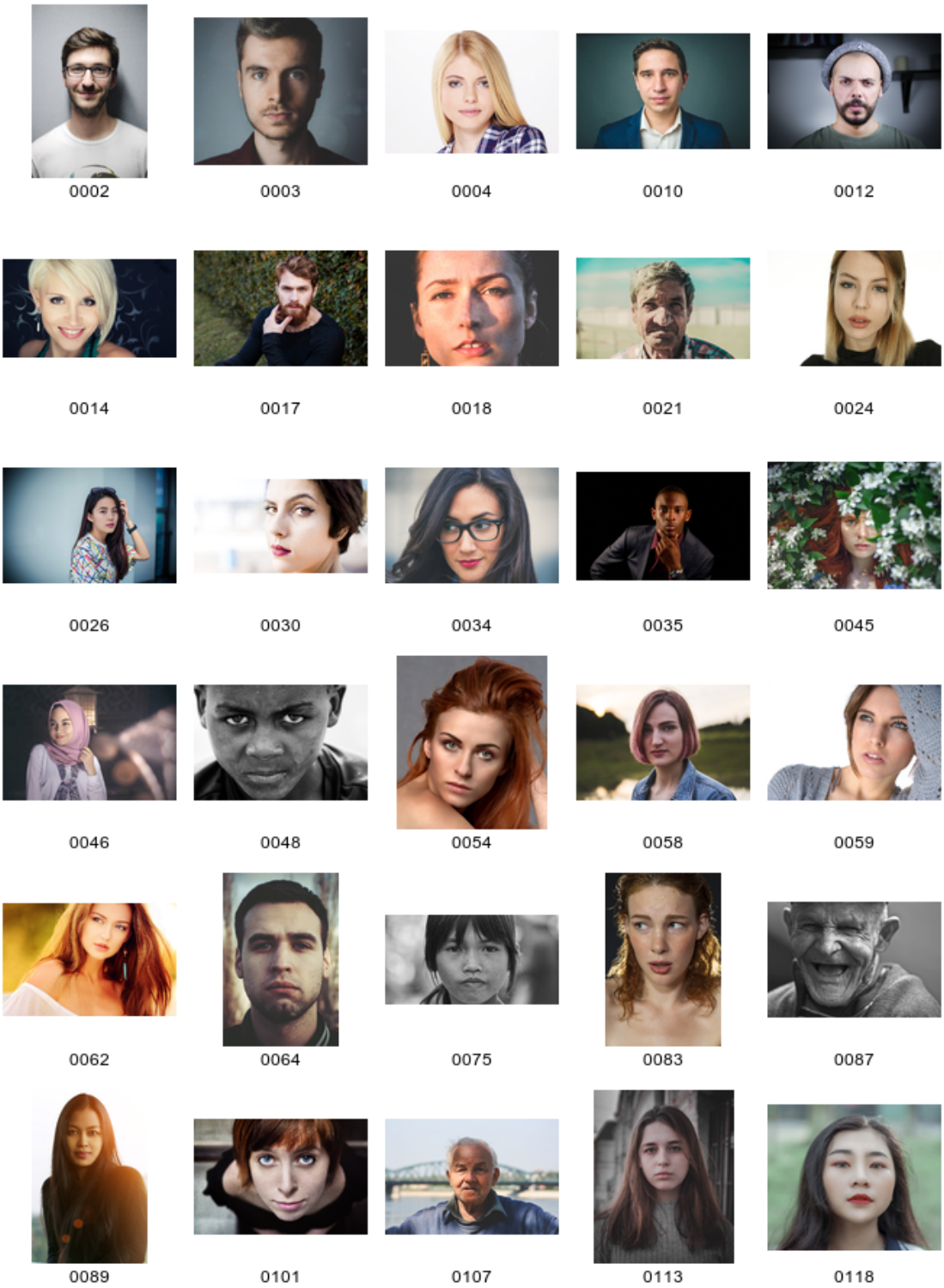

Figure C.1: Some example faces from the Pexels dataset used to visualize or compare different pipeline steps. The numbers refer to the image names.

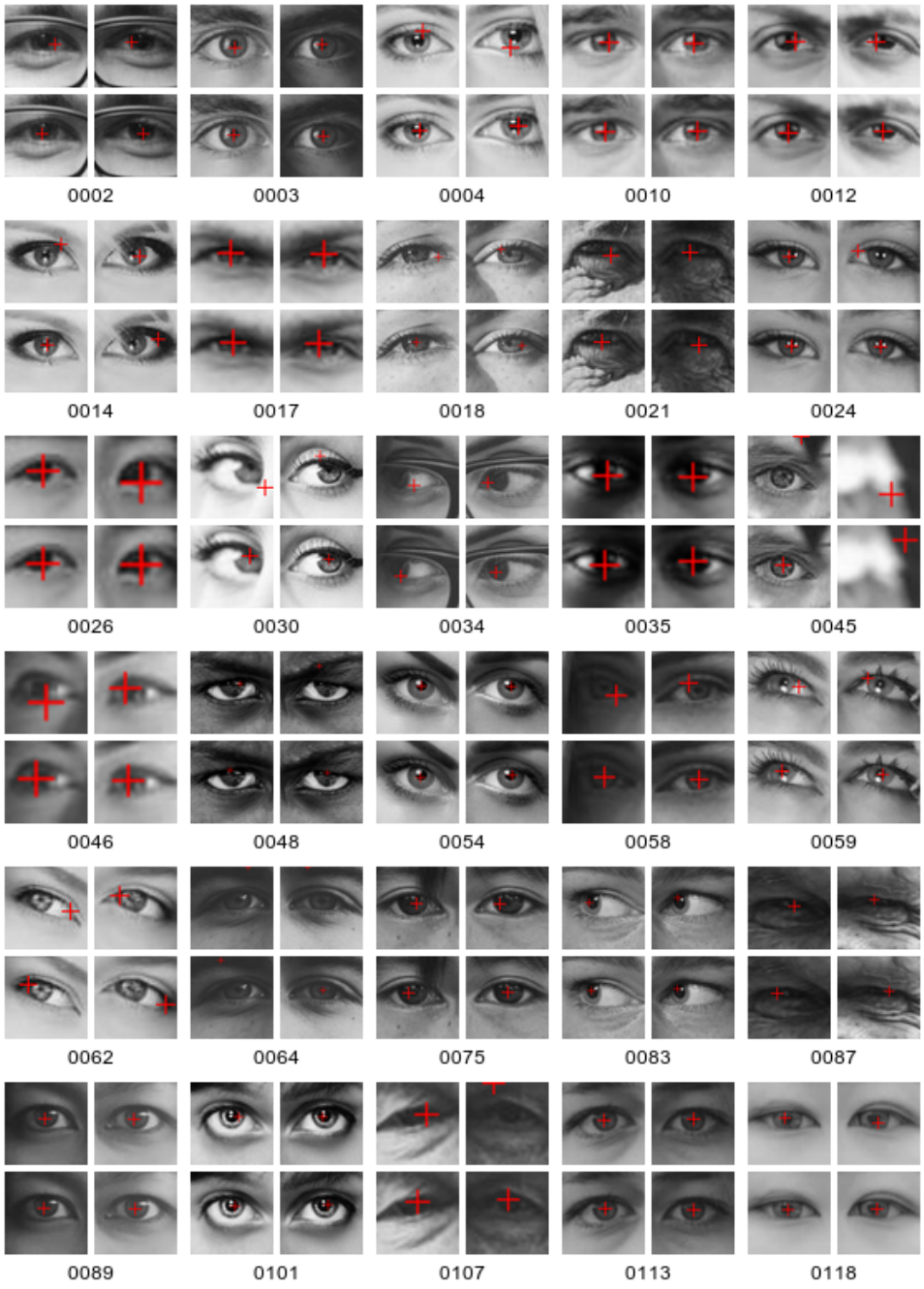

Figure C.2: Comparison of eyeLike and Gaze's pupil detections, showing eyeLike's images above Gaze's. The original images can be seen in Figure C.1. Note that bigger cross markers mean smaller eye image crops.

Figure C.3: Comparison of solutions to the PnP problem using EPnP on the left and the iterative Levenberg–Marquardt optimization in OpenCV's solvePnP function. Pexels images 0000, 0025, 0031, 0044; cropped.

Figure C.4: Comparison of solutions to the PnP problem using the five landmarks model and six landmarks of 68 landmarks model. The left column is Model A described in Section 5.2.3, the middle column Model B, the right column the 68 landmarks model. Pexels images 0000, 0025, 0031, 0044; cropped.

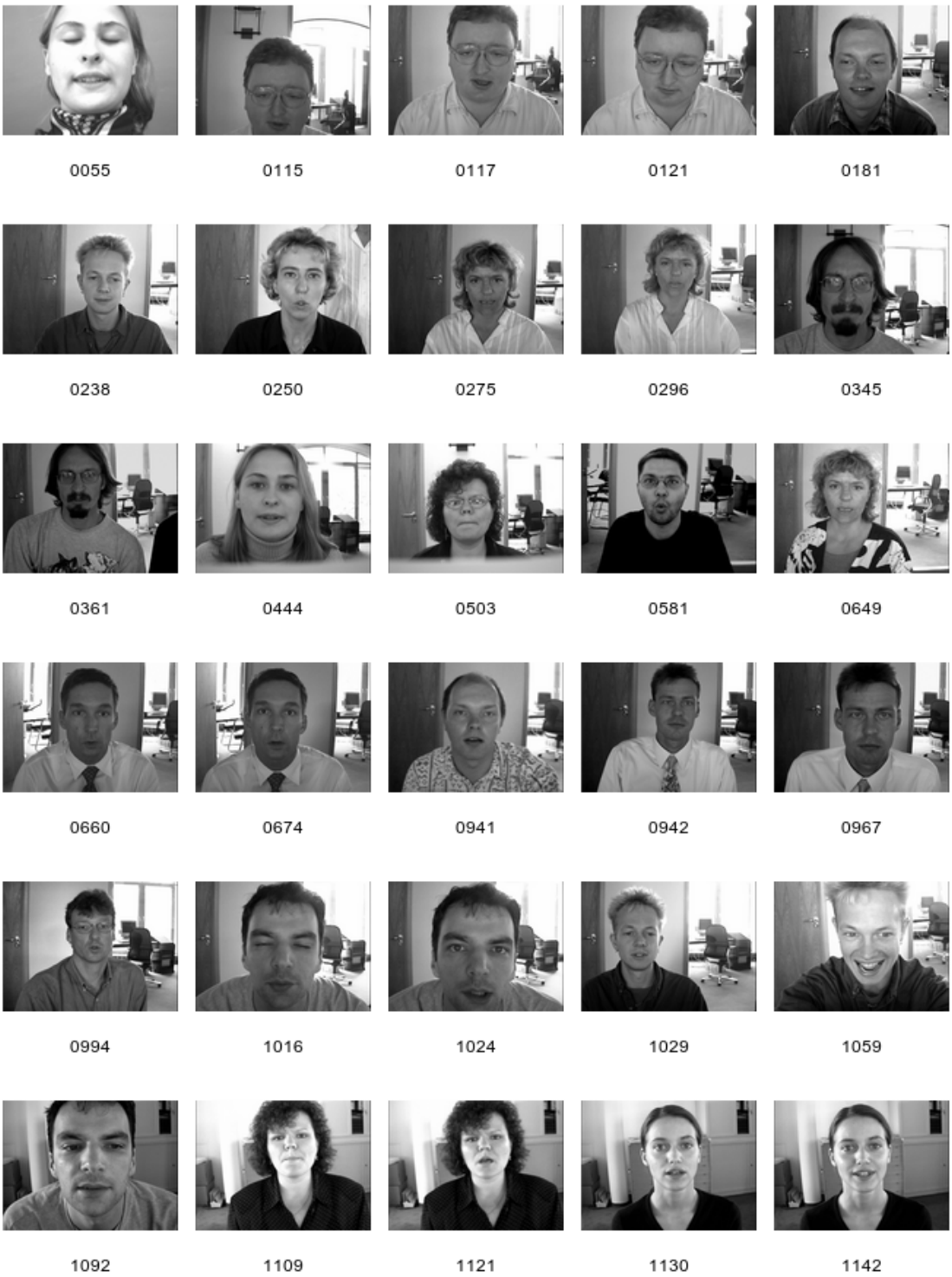

Figure C.5: Some example faces from the BioID dataset. The numbers refer to the image names.

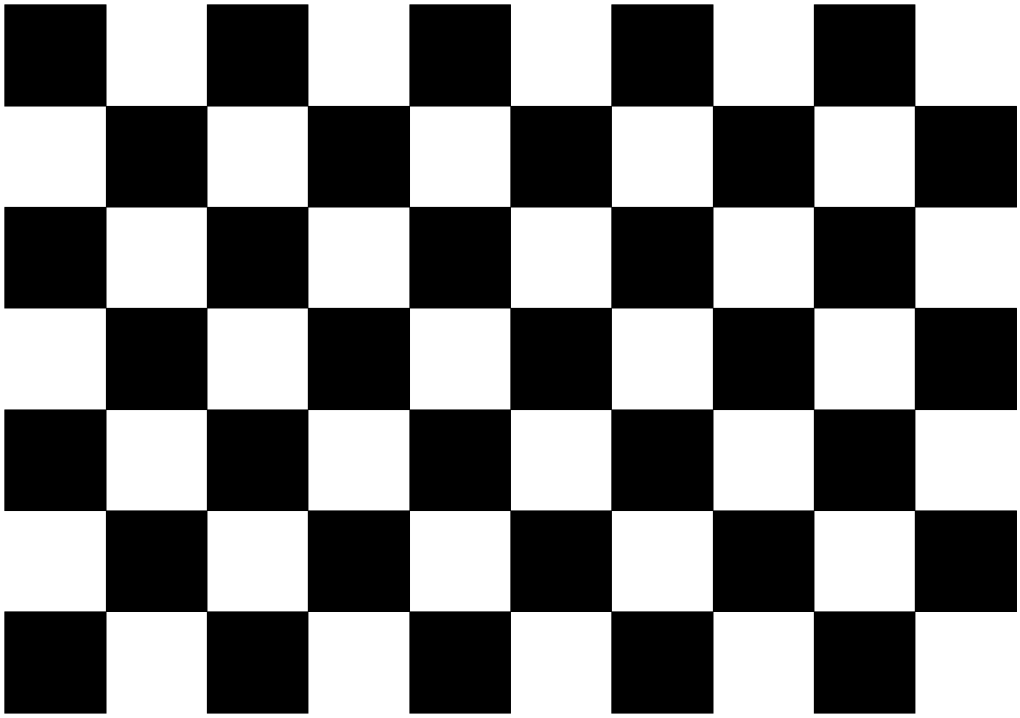

Figure C.6: OpenCV checkerboard pattern to calibrate a camera.

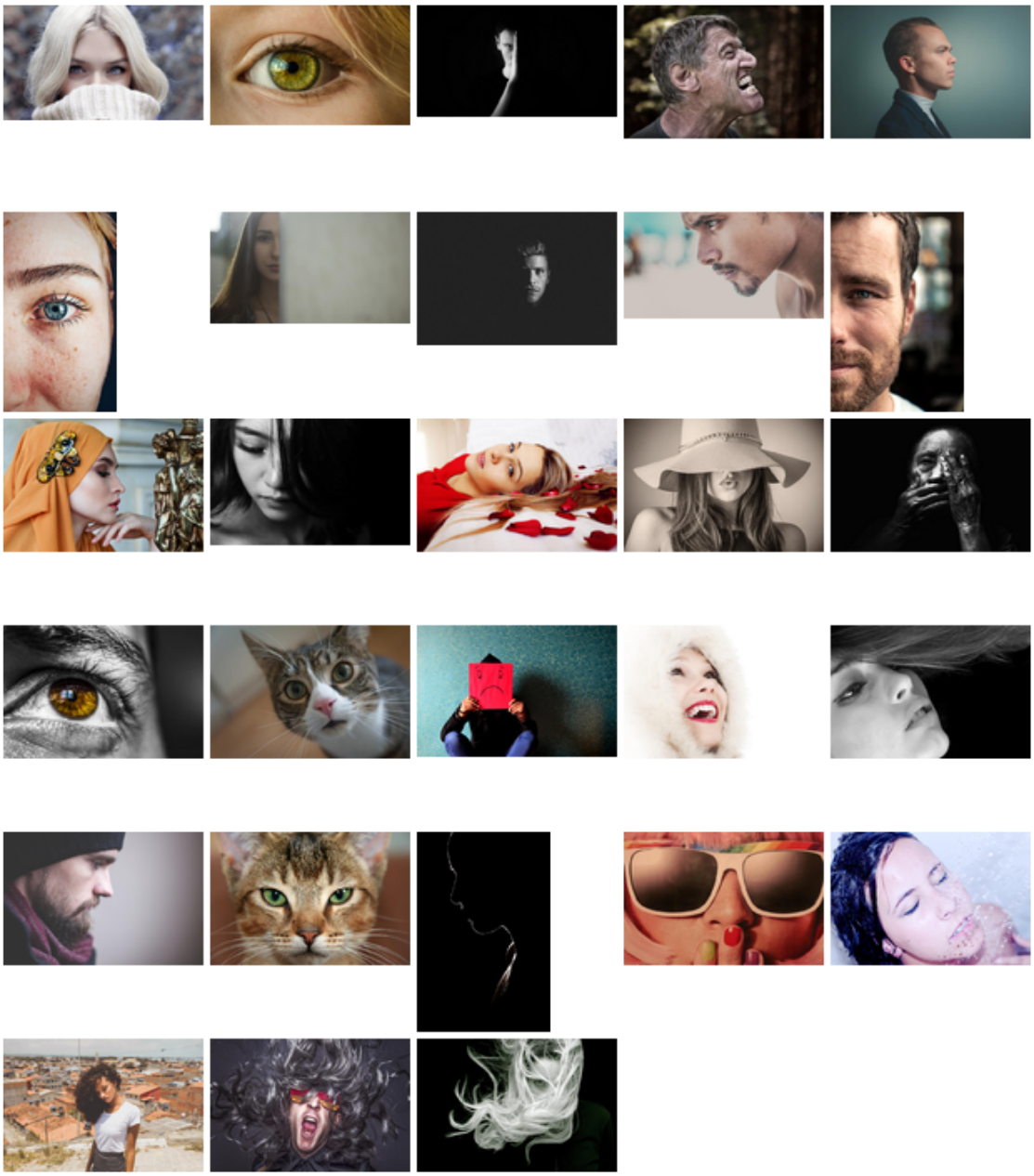

Figure C.7: Images from the Pexels dataset in which Dlib's face detector does not detect any faces. Especially occlusions and strongly tilted heads are difficult.

Appendix D

Code Listings

Code Listing D.1: Example camera calibration output. (`camera_calib.yml`)

```yaml
%YAML:1.0
---
calibration_time: "Tue Dec  5 09:30:19 2017"
image_width: 640
image_height: 480
board_width: 9
board_height: 6
square_size: 1.5000000130385160e-03
aspectRatio: 1.
flags: 2
camera_matrix: !!opencv-matrix
   rows: 3
   cols: 3
   dt: d
   data: [ 7.2395110193974654e+02, 0., 2.9022173109416730e+02, 0.,
       7.2395110193974654e+02, 2.2835668198565884e+02, 0., 0., 1. ]
distortion_coefficients: !!opencv-matrix
   rows: 5
   cols: 1
   dt: d
   data: [ 2.9279643919854315e-01, -1.7443738777411206e+00,
       -9.3538108198059321e-03, -1.2016547499971768e-02,
       2.5922017246373050e+00 ]
avg_reprojection_error: 5.0769582509873890e-01
```

Code Listing D.2: The `init_pipeline()` method. To extend it properly, a new `else if` case has to be added. (`src/gaze/gaze_tracker.cpp`)

```
void GazeTracker::init_pipeline(const std::string subject_id) {
  if (this->initialized) {
    return;
  }
  YAML::Node config = util::get_config()["pipeline"];

  for (YAML::Node step_config : config) {
    std::string type  = step_config["type"].as<std::string>();
    PipelineStep* step;
    if (!type.compare("EyeLike")) {
      step = new pipeline::EyeLike();
    } else if (!type.compare("FaceLandmarks")) {
      step = new pipeline::FaceLandmarks();
    } else if (!type.compare("GazePointCalculation")) {
      step = new pipeline::GazePointCalculation();
    } else if (!type.compare("GazeCapture")) {
      step = new pipeline::GazeCapture();
    } else if (!type.compare("HeadPoseEstimation")) {
      step = new pipeline::HeadPoseEstimation();
    } else if (!type.compare("PupilLocalization")) {
      step = new pipeline::PupilLocalization();
    } else if (!type.compare("SourceCapture")) {
      step = new pipeline::SourceCapture();
    } else {
      step = new pipeline::FallbackStep();
    }

    this->pipeline_steps.push_back(step);
  }
  this->pipeline = new Pipeline(this->pipeline_steps, true);
}
```

Appendix E

References

Appel, K., Pipa, G., & Dresler, M. (2017). Investigating consciousness in the sleep laboratory – an interdisciplinary perspective on lucid dreaming. *Interdisciplinary Science Reviews*. doi:10.1080/03080188.2017.1380468

Barnes, N. (2010). Publish your computer code: It is good enough. *Nature*. doi:10.1038/467753a

Bekerman, I., Gottlieb, P., & Vaiman, M. (2014). Variations in eyeball diameters of the healthy adults. *Journal of Ophthalmology*, *2014*. doi:10.1155/2014/503645

Biggs, J. (2016). The Eye Tribe Tracker Pro offers affordable eye tracking for $199. https://techcrunch.com/2016/01/14/the-eye-tribe-tracker-pro-offers-affordable-eye-tracking-for-199/.
Accessed: 2018-01-26

Chennamma, H. R., & Yuan, X. (2013). A survey on eye-gaze tracking techniques. *arXiv*. arXiv:1312.6410v1

Constine, J. (2016). Oculus acquires eye-tracking startup The Eye Tribe. https://techcrunch.com/2016/12/28/the-eye-tribe-oculus/. Accessed: 2018-01-26

Creative Commons. (2018). CC0 1.0 Universal. https://creativecommons.org/publicdomain/zero/1.0/legalcode. Accessed: 2018-01-13

Crisp, S. (2013). Camera sensor size: Why does it matter and exactly how big are they? https://newatlas.com/camera-sensor-size-guide/26684/. Accessed: 2018-01-28

Dalmaijer, E. (2015). Webcam eye tracker. http://www.pygaze.org/2015/06/webcam-eye-tracker/. Accessed: 2018-01-26

Dalmaijer, E., Mathôt, S., & van der Stigchel, S. (2014). PyGaze: An open-source, cross-platform toolbox for minimal-effort programming of eye tracking experiments. *Behavior Research Methods*, *46*(4), 913–921. doi:10.3758/s13428-013-0422-2

Davson, H. (2017). Human eye. In *Encyclopædia britannica*. Encyclopædia Britannica, Inc. Accessed: 2017-12-11

Driessen, V. (2010). A successful git branching model. http://nvie.com/posts/a-successful-git-branching-model/. Accessed: 2018-01-02

Duc, A. H., Bays, P., & Husain, M. (2008). Eye movements as a probe of attention. In *Progress in Brain Research* (pp. 403–411). Elsevier. doi:10.1016/s0079-6123(08)00659-6

Duchowski, A. (2007). *Eye tracking methodology* (Second Edition.). Springer.

Duchowski, A. T. (2002). A breadth-first survey of eye-tracking applications. *Behavior Research Methods, Instruments, & Computers*, *34*(4), 455–470. doi:10.3758/bf03195475

Fan, H., & Zhou, E. (2016). Approaching human level facial landmark localization by deep learning. *Image and Vision Computing*, *47*, 27–35. doi:10.1016/j.imavis.2015.11.004

Ferhat, O. (2012). *Eye-tracking with webcam-based setups: Implementation of a real-time system and an analysis of factors affecting performance* (Master's thesis). Universitat Autònoma de Barcelona.

Fini, M. R. R., Kashani, M. A. A., & Rahmati, M. (2011). Eye detection and tracking in image with complex background. In *3rd International Conference on Electronics Computer Technology*. IEEE. doi:10.1109/icectech.2011.5942050

Frischholz. (2018). Face detection & recognition homepage. Accessed: 2018-01-14

George, A., & Routray, A. (2016). Fast and accurate algorithm for eye localisation for gaze tracking in low-resolution images. *IET Computer Vision*, *10*(7), 660–669. doi:10.1049/iet-cvi.2015.0316

Gross, H., Blechinger, F., & Achtner, B. (2008). Survey of optical instruments. In H. Gross (Ed.), (Vol. 4, pp. 1–87). Wiley.

Hansen, D. W., & Ji, Q. (2010). In the eye of the beholder: A survey of models for eyes and gaze. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *32*(3), 478–500. doi:10.1109/TPAMI.2009.30

Hansen, D. W., & Pece, A. E. (2005). Eye tracking in the wild. *Computer Vision and Image Understanding*, *98*(1), 155–181. doi:10.1016/j.cviu.2004.07.013

Helmholtz, H. (1866). *Handbuch der physiologischen Optik [Handbook of physiological optics].* (P. W. Brix, G. Decher, F. C. O. von Feilitzsch, F. Grashof, F. Harms, H. Helmholtz, et al., Eds.) (Vol. IX). Gustav Karsten.

Huey, E. B. (1908). *The psychology and pedagogy of reading*. The Macmillian Company.

Hume, T. (2012). Simple, accurate eye center tracking in OpenCV. http://thume.ca/projects/2012/11/04/simple-accurate-eye-center-tracking-in-opencv/. Accessed: 2018-01-11

Jesorsky, O., Kirchberg, K. J., & Frischholz, R. W. (2001). Robust face detection using the Hausdorff distance. In *Third International Conference on Audio- and Video-based Biometric Person Authentication* (pp. 90–95). Springer. doi:10.1007/3-540-45344-X_14

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., et al. (2014). Caffe: Convolutional architecture for fast feature embedding. *arXiv*. arXiv:1408.5093

Judd, T., Ehinger, K., Durand, F., & Torralba, A. (2009). Learning to Predict Where Humans Look. In *12th International Conference on Computer Vision* (pp. 2106–2113). IEEE. doi:10.1109/ICCV.2009.5459462

King, D. E. (2009). Dlib-ml: A machine learning toolkit. *Journal of Machine Learning Research*, *10*, 1755–1758. doi:10.1145/1577069.1755843. http://dlib.net/

King, D. E. (2014). Dlib 18.6 released: Make your own object detector! http://blog.dlib.net/2014/02/dlib-186-released-make-your-own-object.html. Accessed: 2018-01-14

King, D. E. (2015). Max-Margin Object Detection. *arXiv*. arXiv:1502.00046

Krafka, K., Khosla, A., Kellnhofer, P., Kannan, H., Bhandarkar, S., Matusik, W., & Torralba, A. (2016). Eye tracking for everyone. In *IEEE Conference on Computer Vision and Pattern Recognition* (pp. 2176–2184). IEEE. doi:10.1109/CVPR.2016.239. http://gazecapture.csail.mit.edu/index.php

Lemaignan, S., Garcia, F., Jacq, A., & Dillenbourg, P. (2016). From real-time attention assessment to "with-me-nes" in human-robot interaction. In *11th ACM/IEEE International Conference on Human-Robot Interaction*. IEEE. doi:10.1109/hri.2016.7451747

Lepetit, V., Moreno-Noguer, F., & Fua, P. (2009). EPnP: An accurate $O(n)$ solution to the PnP problem. *International Journal of Computer Vision*, *81*(2), 155–166. doi:10.1007/s11263-008-0152-6

Levenberg, K. (1944). A method for the solution of certain non-linear problems in least squares. *Quaterly of Applied Mathematics*, *2*(2), 164–168. doi:10.1090/qam/10666

Li, D., Winfield, D., & Parkhurst, D. J. (2005). Starburst: A hybrid algorithm for video-based eye tracking combining feature-based and model-based approaches. In *IEEE Conference on Computer Vision and Pattern Recognition*. IEEE. doi:10.1109/cvpr.2005.531

Luepke, L. (2005). Apple iSight review. https://www.cnet.com/products/apple-isight/review/. Accessed: 2018-01-18

Mahler, P. (2017). Eye tracker prices – an overview of 15+ eye trackers. https://imotions.com/blog/eye-tracker-prices/. Accessed: 2017-01-26

Mallick, S. (2016). Head pose estimation using OpenCV and Dlib. https://www.learnopencv.com/head-pose-estimation-using-opencv-and-dlib. Accessed: 2018-01-10

Marquardt, D. W. (1963). An algorithm for least-squares estimation of nonlinear parameters. *Journal of the Society for Industrial and Applied Mathematics*, *11*(2), 431–441. doi:10.1137/0111030

Nagamatsu, T., Kamahara, J., & Tanaka, N. (2009). Calibration-free gaze tracking using a binocular 3D eye model. In *Extended Abstracts on Human Factors in Computing Systems*. ACM Press. doi:10.1145/1520340.1520543

Newman, R., Matsumoto, Y., Rougeaux, S., & Zelinsky, A. (2000). Real-time stereo tracking for head pose and gaze estimation. In *4th International Conference on Automatic Face and Gesture Recognition*. IEEE. doi:10.1109/afgr.2000.840622

Noureddin, B., Lawrence, P. D., & Birch, G. E. (2012). Online removal of eye movement and blink EEG artifacts using a high-speed eye tracker. *IEEE Transactions on Biomedical Engineering*, *59*(8), 2103–2110. doi:10.1109/tbme.2011.2108295

Ohno, T., & Mukawa, N. (2004). A free-head, simple calibration, gaze tracking system that enables gaze-based interaction. In *Symposium on eye tracking research & applications*. ACM Press. doi:10.1145/968363.968387

Open Source Initiative. (2018). The MIT license. https://opensource.org/licenses/MIT. Accessed: 2018-01-01

Papoutsaki, A., Sangkloy, P., Laskey, J., Daskalova, N., Huang, J., & Hays, J. (2016). WebGazer: Scalable webcam eye tracking using user interactions. In *25th International Joint Conference on Artificial Intelligence* (pp. 3839–3845). AAAI.

Park, K. R., Lee, J. J., & Kim, J. (2002). Facial and eye gaze detection. In *Biologically Motivated Computer Vision* (pp. 368–376). Springer. doi:10.1007/3-540-36181-2_37

Patacchiola, M., & Cangelosi, A. (2017). Head pose estimation in the wild using convolutional neural networks and adaptive gradient methods. *Pattern Recognition*, *71*, 132–143. doi:10.1016/j.patcog.2017.06.009

Patney, A., Kim, J., Salvi, M., Kaplanyan, A., Wyman, C., Benty, N., et al. (2016). Perceptually-based foveated virtual reality. In *SIGGRAPH*. ACM. doi:10.1145/2929464.2929472

Periketi, P. R. (2011). *Gaze estimation using sclera and iris extraction* (Master's thesis). University of Kentucky.

Rossignol, J. (2017). Apple acquires german eye tracking firm SensoMotoric Instruments. https://www.macrumors.com/2017/06/26/apple-acquires-sensomotoric-instruments/. Accessed: 2018-01-27

Sagonas, C., Antonakos, E., Tzimiropoulos, G., Zafeiriou, S., & Pantic, M. (2016). 300 Faces In-The-Wild Challenge: Database and results. *Image and Vision Computing*, *47*, 3–18. doi:10.1016/j.imavis.2016.01.002. Special Issue on Facial Landmark Localisation "In-The-Wild"

Sagonas, C., Tzimiropoulos, G., Zafeiriou, S., & Pantic, M. (2013). 300 Faces in-the-Wild Challenge: The first facial landmark localization challenge. In *IEEE International Conference on Computer Vision*. IEEE.

Schöning, J., Faion, P., Heidemann, G., & Krumnack, U. (2016). Eye tracking data in multimedia containers for instantaneous visualizations. In *IEEE Workshop on Eye Tracking and Visualization*. IEEE. doi:10.1109/ETVIS.2016.7851171

Shelhamer, M., & Roberts, D. C. (2010). Magnetic scleral search coil. In *Vertigo and Imbalance: Clinical Neurophysiology of the Vestibular System* (pp. 80–87). Elsevier. doi:10.1016/s1567-4231(10)09006-4

Sirohey, S. A., & Rosenfeld, A. (2001). Eye detection in a face image using linear and nonlinear filters. *Pattern Recognition*, *34*(7), 1367–1391. doi:10.1016/s0031-3203(00)00082-0

Soltany, M., Zadeh, S. T., & Pourreza, H.-R. (2011). Fast and accurate pupil positioning algorithm using circular Hough transform and gray projection. In *Computer Science and Information Technology*.

Swennen, G. R. J. (2006). Three-dimensional cephalometry. In G. R. J. Swennen, F. A. C. Schutyser, & J.-E. Hausamen (Eds.), (pp. 183–226). Springer. doi:10.1007/3-540-29011-7_4

Timm, F., & Barth, E. (2011). Accurate eye centre localisation by means of gradients. In L. Mestetskiy & J. Braz (Eds.), *International Conference on Computer Vision Theory and Applications* (Vol. 1, pp. 125–130). INSTICC; SciTePress. doi:10.5220/0003326101250130

Troy, K. (2017). Stackoverflow developer survey 2017. https://stackoverflow.blog/2017/03/22/now-live-stack-overflow-developer-survey-2017-results/. Accessed: 2018-01-01

Vilks, A. (2017). *Faces in the Wild – locating faces in mobile eye tracking videos* (Master's thesis). Osnabrück University.

Viola, P., & Jones, M. (2001). Rapid object detection using a boosted cascade of simple features. In *IEEE Conference on Computer Vision and Pattern Recognition*. IEEE. doi:10.1109/CVPR.2001.990517

Wedel, M., & Pieters, R. (2008). A review of eye-tracking research in marketing. In *Review of Marketing Research* (pp. 123–147). Emerald Group Publishing Limited. doi:10.1108/s1548-6435(2008)0000004009

Weinberg, S. M., & Marazita, M. L. (2009). 3D facial norms database. Accessed: 2018-01-05. NIDCR Grant: 1-U01-DE020078

Wikipedia contributors. (2017a). Line–plane intersection. *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Line-plane_intersection. Accessed: 2018-01-15

Wikipedia contributors. (2017b). Angle of view. *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Angle_of_view. Accessed: 2018-01-19

Wikipedia contributors. (2018). Levenberg–marquardt algorithm. *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Levenberg-Marquardt_algorithm. Accessed: 2018-01-15

Xiong, C., Huang, L., & Liu, C. (2014). Calibration-free gaze tracking for automatic measurement of visual acuity in human infants. In *36th International Conference on Engineering in Medicine and Biology Society*. IEEE. doi:10.1109/embc.2014.6943752

Yarbus, A. L. (1967). *Eye movements and vision*. (L. A. Riggs, Ed.). Plenum Press.

Zhou, E., Fan, H., Cao, Z., Jiang, Y., & Yin, Q. (2013). Extensive facial landmark localization with coarse-to-fine convolutional network cascade. In *IEEE International Conference on Computer Vision Workshops*. IEEE. doi:10.1109/iccvw.2013.58

Appendix F

Declaration of Authorship

I hereby certify that the work presented here is, to the best of my knowledge and belief, original and the result of my own investigations, except as acknowledged, and has not been submitted, either in part or whole, for a degree at this or any other university.

Osnabrück, February 1, 2018

Sebastian Höffner