Institut für Visualisierung und Interaktive Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Das Smartphone-Headset als intergierter Eye-Tracker

Alexander Jäger

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Daniel Weiskopf
Betreuer/in:	Dipl.-Inf. Christoph Schulz
Beginn am:	01. April 2017
Beendet am:	02. Oktober 2017
CR-Nummer:	I.4.8

Kurzfassung

Virtual Reality hat in den letzten Jahren deutlich an Popularität gewonnen und ist wieder ins Rampenlicht gerückt. Damit ist auch ein neues Feld für die Anwendung von Eye-Tracking entstanden, da VR in vielerlei Hinsicht von dieser Technologie profitieren kann. Während für die stationären high-end Geräte die Entwicklung in vollem Gang ist, wurde noch nicht versucht, diese Technologie in VR-Headsets für Smartphones umzusetzen. Da die Frontkamera des Smartphones das Auge ohne zusätzliche Beleuchtung im Headset nicht detailliert genug einfangen kann, teilte sich die Aufgabe in ein software- und hardwareseitiges Problem. Im Auftrag der Universität Stuttgart entwickelte ich zuerst eine simple Eye-Tracking-Anwendung für Android unter Verwendung der OpenCV Bibliothek, anschließend löste ich das Beleuchtungsproblem im Headset. Allerdings funktioniert die Software unter den neuen Beleuchtungsbedingungen im Headset nicht mehr. Schließlich werden Lösungsansätze aufgeführt, um ein funktionierendes Eye-Tracking-System im Smartphone-VR-Szenario umzusetzen.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Motivation und Ziel

Das Phänomen Virtual Reality (kurz VR) hat in den letzten Jahren ein großes Comeback gefeiert. Als in den 50er Jahren erstmals versucht wurde, eine virtuelle Realität zu erschaffen, konnten die Menschen sich wohl nur erträumen, was mit heutiger Technik diesbezüglich möglich ist. Mit VR-Headsets wie Oculus Rift von Facebook oder der PSVR von Sony lassen sich bereits verblüffend immersive virtuelle Realitäten erschaffen. Während solche Head-Mounted Displays (kurz HMDs) immer noch recht teuer sind (eine Oculus Rift kostet aktuell um die 500€), kam Google schnell auf eine kostengünstigere Alternative, um in die virtuelle Welt abtauchen zu können. Da in einer großen Anzahl an Smartphonemodellen ein Gyroskop verbaut ist, mit denen sich die Lage des Telefons im Raum bestimmen lässt, kann man das Smartphone zum Abspielen von VR-Anwendungen verwenden. So ist das Google Cardboard eine Kopfhalterung aus Pappe, die man sich auch einfach Zuhause basteln kann, in der 2 Linsen beigelegt sind. Als Monitor wird dann das VR-fähige Smartphone eingelegt, auf welchem man eine VR-Anwendung laufen lässt [Goo].

Diese VR-Methode bietet zwar nicht die Immersion wie z.B. bei einer Oculus Rift, allerdings können so etliche Konsumenten in diese Technik reinschnuppern, ohne großes Geld ausgeben zu müssen. Außerdem können viele Entwickler sich an VR-Anwendungen versuchen, da sie die nötige Hardware bereits besitzen. Diese “Handyhalterungen“ sind ein wichtiger Schritt für die VR-Entwicklung, da so deutlich mehr Menschen VR ausprobieren und verstehen können, was dem Rückhalt in der Gesellschaft für VR und somit auch direkt der Entwicklung der Technologie zu Gute kommt.

Ein wichtiger Meilenstein in der Entwicklung von VR und ein weiteres Hauptthema dieser Arbeit ist das bildbasierte Eye-Tracking. Im letzten Jahrzehnt ist Eye-Tracking insbesondere durch zwei Anwendungen im Bereich des Marketing bekannt geworden. Einerseits durch die stationäre Verwendung einer Webcam bei PCs, welche es möglich macht die optische Nutzerfreundlichkeit von Programmen und Webseiten zu analysieren [WP08], andererseits durch spezielle Brillen mit verbauten Eye-Trackern, welche z.B. dazu genutzt werden, um herauszufinden, wo eine Person beim Einkauf im Supermarkt hinschaut [SGM08].

Im Kontext von VR erfährt Eye-Tracking eine ganz neue Bedeutung. Erstens weiß dadurch die virtuelle Welt, wo der Benutzer hinschaut, und kann dies verwenden, um die Welt immersiver zu gestalten. Ein Beispiel hierfür wären virtuelle Charaktere die darauf reagieren, wenn man sie anschaut oder sich gekränkt fühlen, wenn man wegschaut, während sie zum Benutzer sprechen. Eine weitere Verwendungsmöglichkeit ist die Bedienung von Spielen oder Anwendungen über den Blickpunkt. Doch der aktuell wichtigste Fortschritt den Eye-Tracking für VR birgt, ist die Leistungssteigerung durch Foveated Rendering. Bei Foveated Rendering wird nur der Bereich des Bildes scharf aufgelöst, auf den der Blickpunkt des Nutzers gerichtet ist. Der Rest kann deutlich weniger stark aufgelöst sein,

da der Mensch selbst immer nur den Teil seiner Sicht scharf sieht, auf den sich seine Augen fokussieren. Durch diese Methode hätte man deutliche Leistungseinsparungen beim Rendering der Bilder, die anderweitig verwendet werden könnten, ganz ohne spürbare Nachteile für den Benutzer [SGE+15].

Dieses Jahr wurde bereits eine erste Brille veröffentlicht, welche ungefähr an die Leistung einer Oculus Rift heranreicht, aber gleichzeitig Eye-Tracking integriert hat [get]. Es ist auch bekannt, dass Facebook und HTC daran arbeiten, in den nächsten Modellen von Oculus Rift bzw. HTC Vive Eye-Tracking zu integrieren [Sau16, Gie17].

Die Motivation für diese Arbeit ist letztlich daraus abgeleitet, dass für die "großen" VR-Brillen intensiv an Eye-Tracking gearbeitet wird, aber für das Eye-Tracking im Kontext von Smartphone-VR nicht. Deshalb habe ich versucht Eye-Tracking in einem Headset wie Google Cardboard mit Hilfe des Smartphones umzusetzen.

1.2 Aufgabenstellung

Die folgenden Aufgaben dienen als Leitfaden dieser Arbeit.

Die erste Aufgabe besteht darin, mich in das Thema einzuarbeiten und einen realistischen Entwurf zur Entwicklung der Software zu machen. Der Entwurf muss unter anderem einen Zeitplan und die angestrebten Funktionalitäten der zu entwickelnden Software enthalten.

Die zweite Aufgabe besteht darin, das Auge mit Hilfe der Frontkamera meines Smartphones zu erfassen. Dazu muss ich mit Techniken des maschinellen Sehens ein Verfahren entwickeln, welches zuverlässig das Auge im Bild isoliert. Hier soll auch ein Messprotokoll angefertigt werden, dass die Erfassung des Auges unter unterschiedlichen Beleuchtungssituationen festhält.

Wenn das Auge erfasst werden kann, muss ein Kalibrierungsprozess entwickelt werden. Bei der Kalibrierung werden Bezugspunkte vom Auge erfasst, welche letztlich die Abbildung des Blickpunktes auf den Bildschirm erlauben. Ist der Kalibrierungsalgorithmus implementiert, muss ein Messprotokoll der Kalibrierung angefertigt werden.

Die letzte Aufgabe in Bezug auf das Programm besteht darin, das Abspielen von Videos bei gleichzeitigem Eye-Tracking zu ermöglichen. Hier sollen Umgebungslicht, Beschleunigungs- und Positionsdaten erfasst und protokollierte Frames mit einem Zeitstempel versehen werden.

Schließlich muss ich diese Ausarbeitung meiner Arbeit anfertigen und eine Abschlusspräsentation im Kolloquium halten.

1.3 Zeitplan

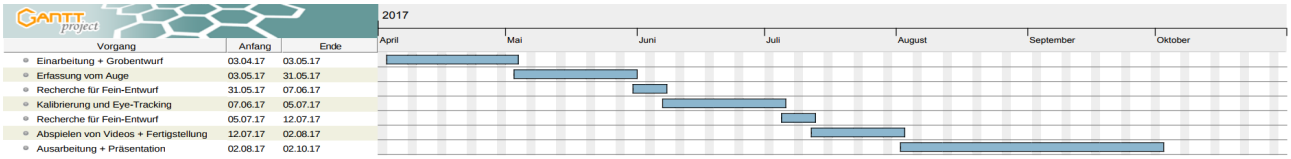


Abbildung 1.1: Zeitplan

2 Grundlagen

2.1 Was ist Eye-Tracking?

Eye-Tracking (zu deutsch auch: Blickerfassung) bezeichnet Techniken mit denen sich die Bewegungen des Auges festhalten lassen. Diese kann man in zwei Phasen unterteilen: Fixationen und Sakkaden. Bei einer Fixation sind die Augen auf einen Punkt fixiert und sehen diesen klar und deutlich, während Sakkaden die ruckartigen Bewegungen der Augen beim Wechsel zwischen zwei Fixationen beschreiben. Während einer Sakkade werden keine visuellen Informationen vom Auge aufgenommen und verarbeitet [Zim]. Deshalb besteht das, was ein normaler Mensch passiv als “Sehen“ empfindet, nur aus Fixationen und somit sind Sakkaden für Eye-Tracking generell nur von geringer Bedeutung. Primär geht es um das Festhalten und Auswerten der Fixationen des Auges.

Die Forschung und der Fortschritt der letzten Jahre beim Thema Eye-Tracking fokussiert sich hauptsächlich auf bildbasierte Systeme, bei denen die Augen unter Zuhilfenahme von Kameras erfasst werden. Zwar gibt es auch nicht-bildbasierte Methoden für Eye-Tracking [Duc07], allerdings sind diese nur in Ausnahmefällen praktikabel.

Nutzungsgebiete von Eye-Tracking sind sehr vielfältig und das hohe Interesse an der Technologie lässt vermuten, dass noch mehr dazukommen werden. Neben den in der Einleitung erwähnten populären Nutzungsmethoden von Eye-Tracking zur Analyse der optischen Benutzerfreundlichkeit von Software, der Analyse von Verkaufsorten (z.B. von Supermärkten) oder der hier behandelten Nutzung im Kontext von VR gibt es weitere relevante Anwendungsgebiete. Eye-Tracker werden beim Lasern der Augen zum Korrigieren von Fehlsichtigkeit verwendet, um Fehler und Verletzungen durch unbeabsichtigte Bewegungen des Auges während der Laserprozedur zu vermeiden. In der Psychologie erleichtert Eye-Tracking die Erforschung von Bild- und Bewegungswahrnehmung und im Bereich der Mensch-Computer-Interaktion ist Eye-Tracking besonders für gehandicapte Personen von sehr hoher Bedeutung. Hier kann der Aktionsspielraum von gelähmten Menschen durch den Einsatz von Augensteuerungen deutlich vergrößert werden, da die Augenmuskulatur auch im fortgeschrittenen Krankheitsstadium häufig noch funktionsfähig ist [Duc07].

2.2 Wie funktioniert bildbasiertes Eye-Tracking?

Bildbasierte Eye-Tracker, ob sie nun in Form einer Brille auf dem Gesicht des Nutzers sitzen oder als eine Webcam über dem betrachteten Monitor montiert sind, funktionieren immer nach dem gleichen Prinzip, welches sich in drei Abschnitte unterteilen lässt. Bildbasiert bedeutet in diesem Zusammenhang, dass die einzige Informationsquelle des Eye-Trackers die einzelnen Bilder sind, die die Kamera vom Auge aufnimmt und an das System weiterleitet.

Im ersten Schritt muss das Auge im Bild erkannt werden. Über Operationen der Bildverarbeitung werden markante Elemente des Auges hervorgehoben, so dass sich der Mittelpunkt der Pupille bestimmen lässt. Hier gibt es einige verschiedene Herangehensweisen, die sich in den Eigenschaften Genauigkeit, Performanz und Anwendbarkeit teilweise stark unterscheiden. Diese werden im Kapitel 2.4 genauer beschrieben.

Wenn die Position der Pupille zu jedem Zeitpunkt, in dem sie sichtbar ist, bestimmt werden kann, muss das System kalibriert werden. Bei der Kalibrierung von Eye-Trackern werden Referenzpunkte für das System gesammelt, mit dessen Hilfe es darauf schließen kann, wohin der Benutzer schaut. Die Kalibrierung ist entscheidend für die Genauigkeit der fertigen Anwendung, weshalb es auch hierfür einige unterschiedliche Methoden gibt. Klassiche Kalibrierungsmethoden zeigen nacheinander Punkte an unterschiedlichen Stellen innerhalb des Systems an, welches zu erfassen gilt. In aller Regel, wie auch in unserem Fall, ist das System ein Monitor. Diese Punkte werden vom Nutzer angeschaut, so dass der Eye-Tracker sich die dazugehörigen Koordinaten der Pupille abspeichern kann. Prinzipiell gilt: Je mehr Referenzpunkte kalibriert werden, desto genauer lässt sich das Eye-Tracking umsetzen. Bei hochwertigen Eye-Trackern betrachtet der Nutzer keine statischen Punkte, sondern einen sich bewegenden Punkt, der über die Bereiche des Monitors wandert. Hier kann ein richtiges Netz aus Referenzpunkten gesammelt werden, wodurch sich die Genauigkeit letztlich immens steigert.

Im letzten Schritt wird mit Hilfe der kalibrierten Bezugspunkte versucht die erfasste Pupillenposition auf einen Bereich im System abzubilden. Um das zu bewerkstelligen, zielt man darauf ab, aus den gesammelten Referenzpunkten ein überbestimmtes Gleichungssystem zu bilden, in welches man die erfassten Pupillenkoordinaten einsetzt. Aus solch einem Gleichungssystem erhält man letztlich eine Approximation des tatsächlich betrachteten Bereiches.

2.3 Computer-Vision Pipeline

Bildbasiertes Eye-Tracking ist der Computer Vision zugehörig. Computer Vision bezeichnet das wissenschaftliche Feld, in dem Software mit Daten von Bildsensoren arbeitet. Um aus den Rohbildern, die von den Bildsensoren gemacht werden, verwertbare Informationen zu gewinnen, müssen bestimmte Operationen in einer bestimmten Reihenfolge durchgeführt werden. Diese Reihenfolge wird allgemein als Computer Vison Pipeline bezeichnet.

2.3.1 Bildvorverarbeitung/Preprocessing

Nachdem die Kamerasensoren ein Bild gemacht haben, wird es so “präpariert“, dass die relevanten Elemente möglichst gut sichtbar sind und auf jeden Fall entdeckt werden können, sofern sie sich im Bild befinden.

Farbraumoperationen Einer der ersten Schritte in der Pipeline besteht oft daraus den Farbraum anzupassen. Gewöhnlicherweise wird das Bild aus einer Kamera im RGB-Farbraum übermittelt. Sprich: Jeder Pixel des Bildes ist durch ein Tripel aus den Intensitätswerten für die Farben Rot, Grün und Blau definiert. Dies ist zwar dem natürlichen Sehen nachempfunden, allerdings eignet sich

der Farbraum eher schlecht für Bildoperationen. In dieser Hinsicht ist der HSV-Farbraum deutlich beliebter. Hier setzt sich ein Pixel aus den Tripelwerten Farbton (Hue), Farbsättigung (Saturation) und Helligkeit (Value) zusammen. Damit lassen sich Bilder direkt nach beliebigen Farben, Sättigungs- und Helligkeitsgraden filtern, was im RGB-Raum ohne Weiteres so nicht möglich ist [GW06].

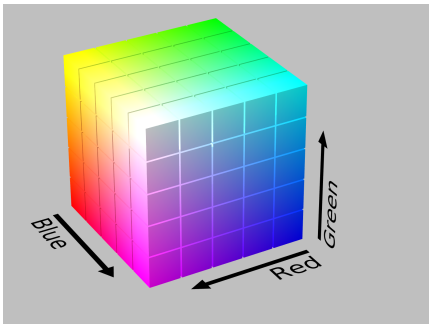


Abbildung 2.1: RGB Farbraum, Quelle: http://tinyurl.com/y8h7w3ok

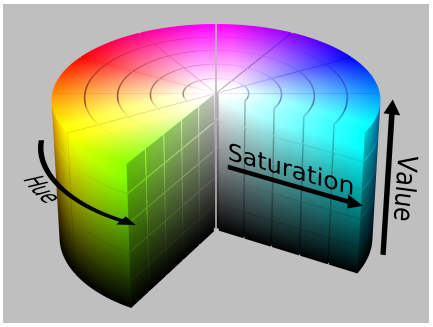


Abbildung 2.2: HSVFarbraum, Quelle: http://tinyurl.com/y83qz8zu

Eine weitere wichtige Operation ist das Umwandeln vom Farbbild in ein Graustufen- bzw. Schwarz-Weiß-Bild. Hierbei ist zu beachten, dass ein Farbbild mindestens 3 Kanäle hat (z.B. jeweils einen für Rot, Grün, Blau), aber ein Graustufenbild lediglich einen braucht. Dies verringert die Komplexität und damit die Performanz deutlich.

Abbildung 2.3: Umwandlung vom Farb- zum Graustufenbild, Quelle: http://tinyurl.com/jzp7anh

Smoothing/Glätten Um Bildrauschen zu verringern, werden oft lineare Filter angewandt. Dabei werden die Werte aller Pixel einander angeglichen, indem man jeden Pixel mit seinen Nachbarn vergleicht. Ist sein Wert deutlich höher als die seiner Nachbarn, wird sein Wert verringert, ist sein Wert deutlich kleiner, wird er erhöht. Durch solche Filter wird zwar die Schärfe des Bildes geringer, dies kommt aber einigen aufbauenden Operationen wie (z.B. bei der Kantendetektion) zu Gute [GW06].

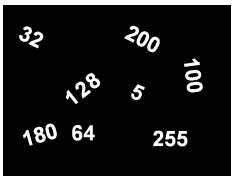
Abbildung 2.4: Gaußscher Weichzeichner angewandt, Quelle: http://tinyurl.com/y7lyu55a

2.3.2 Segmentierungsoperationen

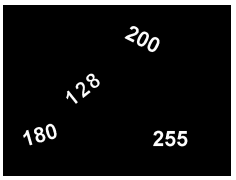
Ist das Bild vorbearbeitet worden, beginnt die Bildanalyse. Der erste Schritt hierbei sind sogenannte Segmentierungsoperationen. Das Ziel ist hierbei möglichst jedes Pixel einem Segment unterzuordnen. Es gibt pixelorientierte, kantenorientierte, regionenorientierte, modell- und texturbasierte Segmentierungsverfahren.

Thresholding Das Segmentieren eines Bildes nach einem bestimmten Schwellenwert wird Thresholding genannt. Beim maschinellen Sehen sind binäre Threshold-Operationen von großer Bedeutung. Eine binäre Threshold-Operation bildet ein Farb- oder Graustufenbild auf ein sogenanntes Binärbild ab. Bei einem Binärbild ist jeder Pixel durch einen Bit definiert. Thresholding ist ein pixelorientiertes Segmentierungsverfahren. Über- oder Unterschreitet die Intensität eines Pixels des Ausgangsbildes einen festgelegten Schwellenwert, ist der Pixel im Binärbild entsprechend weiß oder schwarz. Somit enthalten sie die kleinstmögliche Informationsmenge, die ein Bild besitzen kann. Diese Eigenschaft macht Binärbilder für maschinelles Sehen unabdingbar, denn dadurch können wichtige Bildinformationen von nicht wichtigen direkt und einfach getrennt werden. Die Herausforderung besteht allerdings darin das Bild so vorzubereiten, dass man durch einen Schwellenwert genau die Informationen erhält, die man auch benötigt [GW06].

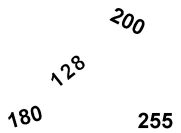
(a) Ausgangsbild



(b) Niedriger binärer Thresholdwert



(c) Hoher binärer Thresholdwert



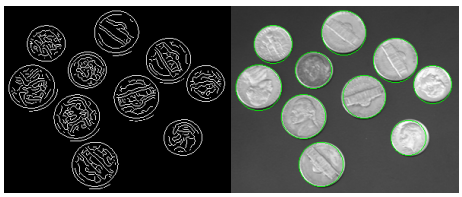
(d) Hoher binärer Thresholdwert invers

Abbildung 2.5: Beispiel für binäre Thresholds, Quelle: http://tinyurl.com/j52uwlm

Edgedetection Mit Hilfe von Kantendetektionsalgorithmen lassen sich Objekte in Bildern über ihre Kantenform segmentieren. Eine der im Kontext von Computer Vision bewährten Methoden hierfür ist der mehrstufige Canny-Edge-Detector. Angewandt auf ein geglättetes Graustufenbild werden große Schwankungen in der Intensität von benachbarten Pixeln als Kanten erkannt. Um dickere Kanten dünn zu machen, werden bei der Non-Maximum Surpression nur die Pixel als Kantenpixel akzeptiert, welche die größte Kantenstärke an einer erkannten Kante aufweisen. Edgedetection entspricht den kantenorientierten Segmentierungsverfahren [GW06].

Abbildung 2.6: Canny-Edge angewandt, Quelle: http://tinyurl.com/yc2wq8fg

Hough-Transformation Ein wichtige Methode der modellbasierten Segmentierung stellt die Hough-Transformation dar. Diese wird auf Kantenbilder angewandt und zumeist für die Detektion von Linien oder Kreisen verwendet. Möchte man alle Geraden in einem Kantenbild finden, zieht man nacheinander durch jeden Kantenpunkt alle möglichen Geraden. Je mehr andere Kantenpunkte auf einer von diesen Geraden liegen, desto eher entspricht diese Gerade einer richtigen Gerade des ursprünglichen Bildes. Möchte man Kreise detektieren, geht man davon aus, dass der jeweils abgefragte Kantenpunkt dem Mittelpunkt des Kreises entspricht. Dann wird der Radius des potentiellen Kreises stufenweise erhöht und gleichzeitig geprüft, wie viele Kantenpunkte sich auf der Kreislinie befinden. Je mehr Kantenpunkte gefunden werden, desto eher ist der gefundene Kreis ein Kreis des ursprünglichen Bildes. Die Hough-Transformation ist ein Brute-Force-Algorithmus, da die Geraden/Kreise nur durch das Ausprobieren aller möglichen Geraden/Kreise gefunden werden können. Deshalb sind diese Algorithmen eher rechenaufwändiger als die, die bisher vorgestellt wurden [STZP11].



(a) Kantenbild nach Canny-Edge **(b)** Resultat

Abbildung 2.7: Beispiel für Hough-Kreise-Detektion, Quelle: http://tinyurl.com/y9jw9smo

2.3.3 Merkmalsextraktion und Klassifikation

Nachfolgend kommen in der Computer Vision Pipeline die Merkmalsextraktion von Segmenten und deren Klassifizierung. Bei der Merkmalsextraktion geht es darum Segmente an Hand von Merkmalen vergleichbar zu machen. Mit Hilfe solcher Merkmale kann man anschließend die Segmente Klassifizieren und definierten Objekten zuordnen. Da bei dieser Arbeit der zentrale Punkt ist, das Auge (also ein einziges Objekt im Bild) zu identifizieren/segmentieren, sind hierfür keine besonderen Methoden der Merkmalsextraktion oder Klassifikation relevant.

2.4 Kontur-/Augenerkennung

Wie im Kapitel 2.2 angedeutet wurde, gibt es eine beträchtliche Zahl an unterschiedlichen Herangehensweisen um die Position des Auges zu tracken. Hochwertige Eye-Tracker zeichnen sich durch mehrstufige, komplexe Algorithmen aus, die rechenaufwendiger sind. Da für unsere Arbeit die generelle Umsetzbarkeit an erster und die Qualität des Eye-Trackers an zweiter Stelle steht, sind hier primitive, direkte Algorithmen gefragt. An dieser Stelle werden zwei Arten von Pupillenerkennung vorgestellt, welche für diese Arbeit relevant sind.

Eine Möglichkeit, um an den Mittelpunkt des Auges zu kommen, besteht darin seine Kreisförmigkeit auszunutzen. Die Circle-Hough-Transformation ist hierfür das einfachste Mittel. Wie im letzten Kapitel beschrieben wurde, ist diese Methode dafür gedacht, alle kreisförmigen Strukturen eines Bildes an Hand der davor detektierten Kanten festzustellen. Da die Pupille und die Iris frontal betrachtet kreisförmig sind, ist es naheliegend, dass diese Methode geeignet ist [ARM13].

Man kann die Pupille auch an Hand ihrer Schwärze erfassen. Geht man von einer gut beleuchteten Situation aus, ist die Pupille das dunkelste im Augenbereich. Die Iris ist je nach Farbe mal deutlich, mal nicht so deutlich heller als die Pupille und außerhalb der Iris ist das Auge annähernd weiß. Nur Wimpern besitzen einen ähnlich dunklen Farbton. So lässt sich mit Hilfe eines einfachen inversen binären Thresholds, der alle Farbwerte abgesehen von den dunkelsten herausfiltert, die Pupille sichtbar machen [ARM13, FTBK16].

3 Entwurf

3.1 Ausgangslage/Hardware

3.1.1 Das Headset

Als VR-HMD wurde für dieses Projekt die Universal 3D VR von Excelvan verwendet. Für unter 20€ hat es ein Gehäuse komplett aus Kunststoff und ein abnehmbares, dreiseitiges Kopfband, mit dem sich das Headset angenehm am Kopf fixieren lässt. Auf der Vorderseite ist eine Klappe, in welche Smartphones bis zu einer Größe von 6 Zoll eingelegt werden können. Zur Fixierung des Smartphones ist auf der Innenseite der Klappe eine Gummifläche mit mehreren Saugnäpfen angebracht. Außerdem sind zwei Linsen verbaut, die ein wenig horizontal verschoben werden können, damit die Sicht auch bei unterschiedlich großen Smartphones angepasst werden kann. Bei eingelegtem Smartphone ist der Innenraum des Headsets vollständig von Umgebungslicht abgeschirmt. Der Hohlraum zwischen dem Gesicht des Nutzers und den Linsen ist bei Benutzung ebenfalls gut abgeschirmt. Allerdings sind hier zwei kleine Schlitze unterhalb der Augen angebracht, die dazu verwendet werden können, externe Lichtquellen anzubringen.

Abbildung 3.1: Headset geöffnet. Quelle: http://tinyurl.com/ybbcoro2

Abbildung 3.2: Headset Frontseite. Quelle: http://tinyurl.com/ycrq6ctv

3.1.2 Das Smartphone

Als Hardwarebasis für unsere Software dient das Smartphone Blade V7 von ZTE. Es verfügt über ein 5,2 Zoll Full-HD Display und eine Frontkamera von 5 Megapixel. Android 6.0 dient als Betriebssystem [zte].

Abbildung 3.3: Zielgerät Blade V7

3.2 Trennung von Software- und Hardwareproblem

Eine der ersten zentralen Fragen bestand darin herauszufinden, ob die Beleuchtung im Anwendungsszenario gut genug ist, damit die Innenkamera das Auge ausreichend präzise einfangen kann. Da bei normaler Verwendung keinerlei Beleuchtungsquellen vorhanden sind abgesehen vom Display, wurde zuerst geprüft, wie die Beleuchtung vom Auge ist bei unterschiedlichen Helligkeitseinstellungen des Bildschirms. Hier wurde allerdings schnell klar, dass der Monitor nicht als Beleuchtungsquelle geeignet ist, weil die Linse den Bildschirminhalt bei höherer Helligkeit stärker reflektiert. Je stärker die Reflexion in der Linse ist, desto schlechter ist die Sichtbarkeit auf das Auge und umso schwieriger ist es relevante Augenmerkmale fehlerfrei zu segmentieren.

Diese Tatsache birgt bereits die erste Erkenntnis, dass Headsets solcher Art in ihrer normalen Form nicht für Eye-Tracking geeignet sind. Um so etwas zu realisieren, muss eine individuelle Lösung für das Beleuchtungsproblem gefunden werden. Deshalb habe ich mich entschieden, das Beleuchtungsproblem hinten anzustellen und davor eine Softwarelösung zu finden, ausgehend von einer guten, gleichmäßigen Ausleuchtung des Auges. Bei der Entwicklung wurde also vorrangig nicht im Headset getestet, sondern das Smartphone mit der Hand in einer ähnlichen Lage und Entfernung gehalten, wie es im Headset der Fall ist.

3.3 Softwareentwurf

3.3.1 Entwicklungsumgebung und Camera2 API

Als Entwicklungsumgebung dient das Android Studio 2.3.3 von Google. Das Smartphone auf dem entwickelt wird, wurde unter anderem deshalb gewählt, weil es auf Android 6 läuft und somit über die zweite Umsetzung der Android Kamera-API verfügt. Die Camera2-API wurde von Google mit Android 5 eingeführt und soll die ursprüngliche API langfristig ablösen. Die erste Camera-API ist zwar recht simpel und dadurch benutzerfreundlich, allerdings war vielen Entwicklern ein Dorn im Auge, dass man kaum direkten Zugriff auf Kamerafunktionen und generell nur primitive Interaktionsmöglichkeiten mit der Kamera hatte. Dementsprechend ist die Camera2-API deutlich komplexer und mächtiger als die erste ausgefallen. Theoretisch kann man hiermit einige Features wie z.B. Fokus, Weißausgleich, ISO-Werte usw. manuell einstellen und beeinflussen, was für unser Vorhaben vermutlich von Nützen wäre. Der Haken besteht hierbei allerdings darin, dass die Smartphoneproduzenten die jeweilige Hardware ihrer Geräte auf die Camera2-API explizit abstimmen müssen. Dies erlauben sich aktuell aber nur die großen Produzenten wie Samsung, HTC oder Google selbst. Noname-Geräte wie die von ZTE gehören da bisher nicht dazu. Das bedeutet wiederum, dass solche Modelle zwar die Camera2-API benützen können, aber nicht von den Fortschritten dieser profitieren. Nach einiger Einarbeitung in die Materie wurde also klar, dass die Camera2-API entgegen erster Erwartungen völlig nutzlos für unser Vorhaben ist [mar].

3.3.2 OpenCV

Open Source Computer Vision (kurz: OpenCV) ist eine Bibliothek von diversen Methoden der Computer Vision Pipeline. Es ist frei verfügbar unter einer BSD Lizenz und insbesondere für Echtzeitanwendungen gedacht. Entwickelt in C/C++ bietet OpenCV auch Schnittstellen für Python und Java. Da OpenCV alle gängigen Plattformen inklusive Android unterstützt, dient es hervorragend als Basiswerkzeug für das Projekt [tea]. OpenCV4Android kommt mit einer eigenen Kamera-API daher, welche sich von der echten Kamera-API allerdings kaum unterscheidet. Um OpenCV4Android nutzen zu können, muss das entsprechende Paket in die Entwicklungsumgebung eingebunden werden und es muss auf dem Zielgerät der OpenCVManager installiert sein, der im Play Store kostenlos bezogen werden kann.

3.3.3 Klassendiagramm

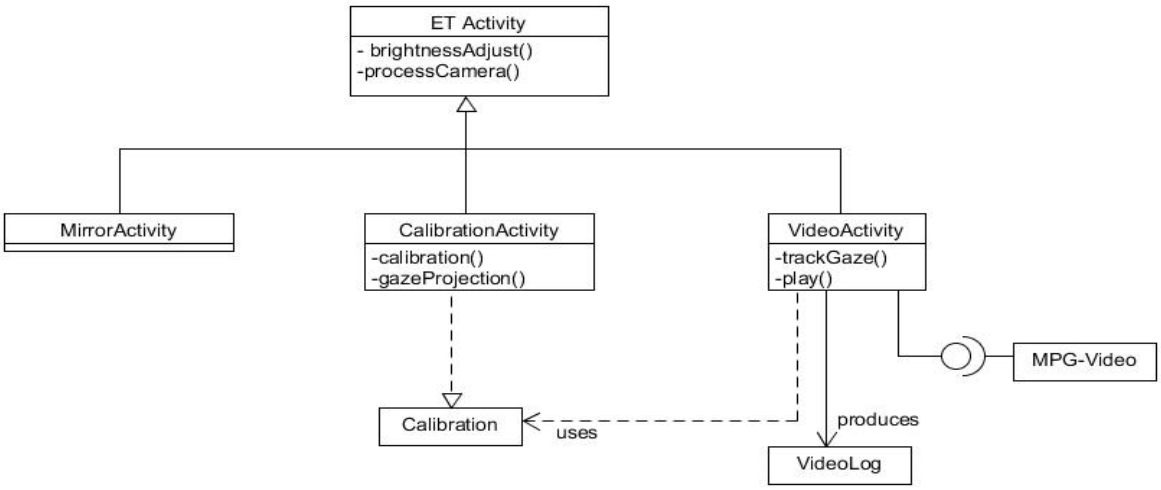


Abbildung 3.4: Klassendiagramm der Zielsoftware

In der oberen Figur ist der erste Entwurf für die Zielsoftware dargestellt. Eine Hauptaktivität erbt zwei Methoden an drei Aktivitäten, welche auch den Fenstern entsprechen. Die zwei Methoden erlauben das Ändern der Helligkeit des Bildschirms und den Zugriff auf die Kamera. Die MirrorActivity entspricht dem Fenster, indem die Ausgabe der Kamera zu sehen ist. Dieses Fenster dient in erster Linie der Entwicklung, um direkt sehen zu können, was die Kamera bzw. das Programm sieht. In der CalibrationActicity wird die Kalibrierung und anschließende Abbildung der Blickrichtung umgesetzt. Wenn die Kalibrierung vollendet ist, kann man zur Anwendung des Eye-Tracking mit der VideoActivity ein Video im MPG-Format abspielen. Während dem Video werden die getrackten Daten mitsamt Timestamp in einem VideoLog gesichert.

3.3.4 Vorgehensmodell

Als Vorgehensmodell dient das Wasserfallmodell, wobei zuerst das Eye-Tracking umgesetzt werden soll und erst danach Anwendungen dafür (Videoplayer). Da die Entwicklung auf ein Software- und Hardwareproblem zweigeteilt wurde, sollte nach der Fertigstellung des Eye-Trackers das Beleuchtungsproblem angegangen werden. Wenn das gelöst werden kann, muss kontrolliert werden, ob der Eye-Tracker auch im modifizierten Headset sich so verhält, wie bisher. Falls der Eye-Tracker nicht mehr funktioniert, muss er angepasst werden.

4 Implementierung

4.1 Protokollierung

Die Fortschritte und Ergebnisse werden in Form von Bildern und Logging protokolliert. Ein Log ist eine Systemnachricht, die während der Laufzeit eines Programms entsteht. Man kann solche Nachrichten selbst einfügen, um Informationen über das System zu bestimmten Zeitpunkten zu bekommen. Zu der Ausführung einer Anwendung entsteht ein Protokoll aus Logging-Nachrichten. Solch ein Bericht lässt sich in einer Text-Datei exportieren.

Abgesehen davon werden hauptsächlich Screenshots und Bilder zur Protokollierung verwendet. Für Bilder aus der Anwendung habe ich eine Methode geschrieben, welche die ihr übergebenen Matrizen als Bitmap abspeichert.

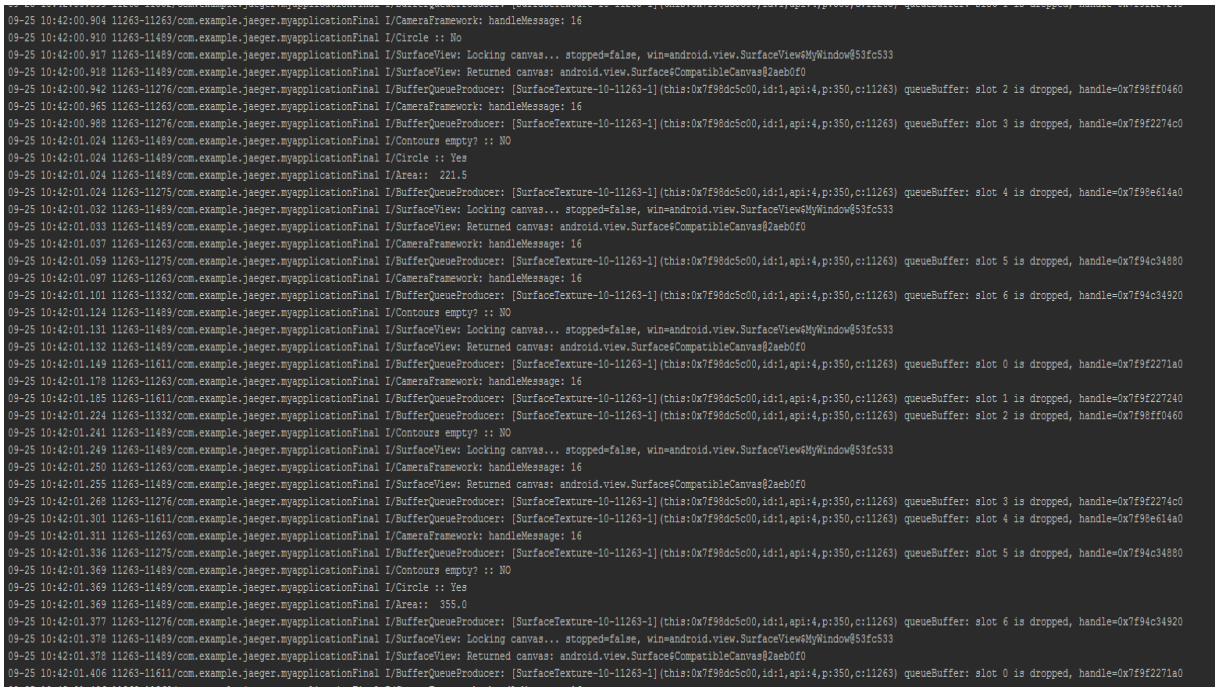


Abbildung 4.1: Screenshot von einem Ausschnitt aus einer Logdatei

4.2 Grundstruktur und OpenCV-Workflow

Da diese Eye-Tracking-Software auf den Bildern der Kamera basiert, ist es wichtig im Manifest die Erlaubnis zur Benutzung der Kamera anzufordern. Außerdem speichern wir Screenshots und Videos, weshalb wir auch die Erlaubnis zum Beschreiben des Speichers brauchen.

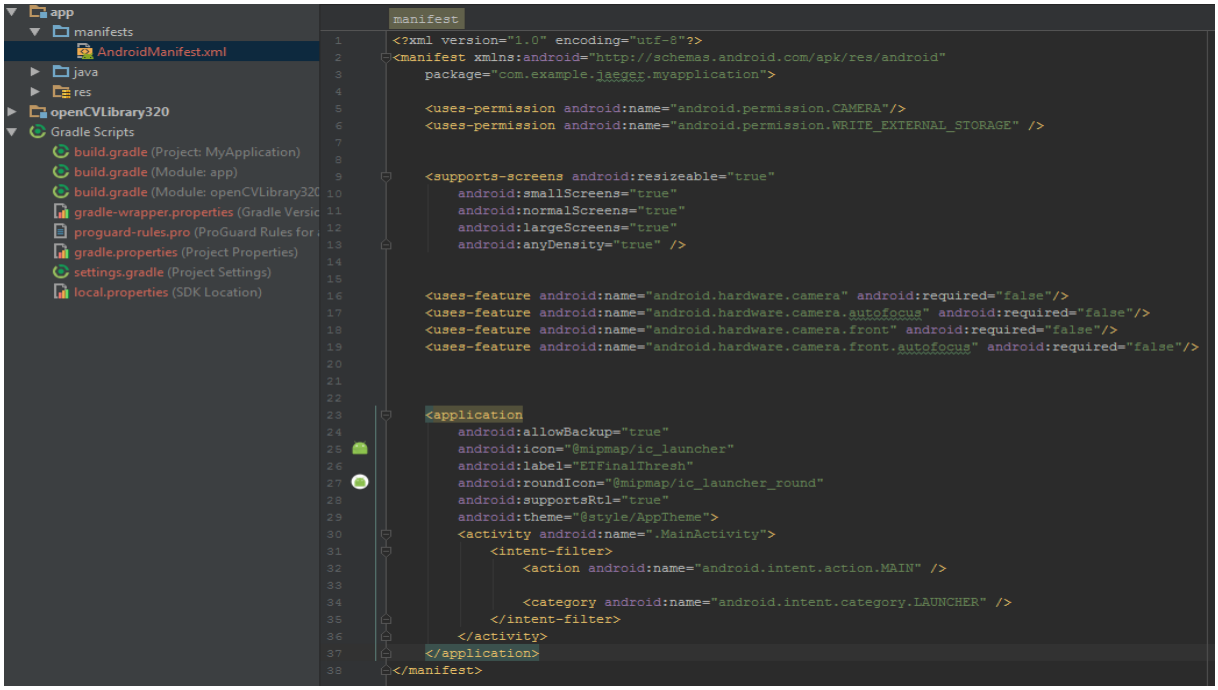


Abbildung 4.2: Manifest der Anwendung

Des Weiteren brauchen wir ein Layout für die Ausgabe der Kameradaten. Dies ist das Fenster, in dem das Bild der Kamera angefordert, bearbeitet und ausgegeben wird.

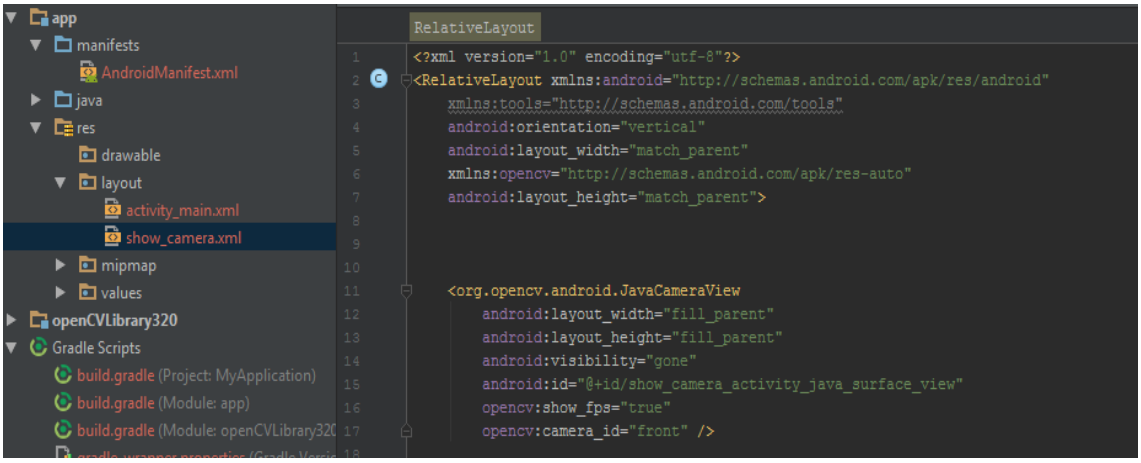


Abbildung 4.3: Layout für das Hauptfenster

Die MainActivity implementiert die CameraViewListener2-Schnittstelle von OpenCV, welche über ein CameraBridgeViewBase-Objekt auf die Kamera zugreift und den Stream als View ausgibt. Die Schnittstelle stellt drei Methoden zur Kamerainteraktion zur Verfügung: onCameraViewStarted, onCameraViewStopped und onCameraFrame. OnCameraViewStarted wird einmalig zum Beginn des Videostreams ausgeführt und hauptsächlich dafür verwendet, die Matrizen zu initialisieren, welche in der onCameraFrame-Methode gebraucht werden. OnCameraViewStopped wird zum Schluss einmalig ausgeführt und hier werden die Matrizen wieder freigegeben. Die OnCameraFrame-Methode bietet als Parameter den aktuellen Inputframe aus der Kamera und muss eine Matrix zurückgeben, die dann das Ausgabebild auf dem verbunden View ist. Diese Methode ist die wichtigste im gesamten Programm, da hier die komplette Bildbearbeitung für die Erkennung des Auges durchgeführt wird.

```
public void onCameraViewStarted(int width, int height) {

    mRgba = new Mat(height, width, CvType.CV_8UC4);
    mRgbaF = new Mat(height, width, CvType.CV_8UC4);
    mRgbaT = new Mat(width, width, CvType.CV_8UC4);

    circles = new Mat();
    mGray = new Mat();
    mCanny = new Mat();
    edges = new Mat();

}

public void onCameraViewStopped() {
    mRgba.release();
    mRgbaF.release();
    mRgbaT.release();

    circles.release();
    mGray.release();
    mCanny.release();
    edges.release();
}

public Mat onCameraFrame(CvCameraViewFrame inputFrame) {

    // TODO Auto-generated method stub
    mGray = inputFrame.gray();
    mCanny = inputFrame.gray();
    // Rotate mRgba 90 degrees
    Core.transpose(mGray, mRgbaT);
    Imgproc.resize(mGray, mRgbaF, mRgbaF.size(), 0,0, 0);
```

Abbildung 4.4: Kameraschnittstellen von OpenCV

Die onCreate-Methode von Android wird zur Initialisierung der Anwendung genutzt. Zuerst muss man abfragen, ob die Zugriffsrechte für Kamera und Speicher gegeben sind, da mit Android 6 alle Rechte vom Benutzer einmalig zur Laufzeit gegeben werden müssen. Danach wird der View für die Kamera hergestellt, über die CameraBridgeViewBase mit OpenCV verbunden und gestartet. Schließlich wird eine Flagge gesetzt, damit das Display immer angeschaltet bleibt während der Ausführung, und die Actionbar entfernt, damit das Kamerabild auf dem vollen Bildschirm ausgegeben werden kann.

```
64    protected void onCreate(Bundle savedInstanceState) {
65        Log.i(TAG, "called onCreate");
66        super.onCreate(savedInstanceState);
67        //Do we have permission for camera?
68        int permissionCheck = ContextCompat.checkSelfPermission(this,
69                CAMERA);
70        //Do we have permission for writing on the storage?
71        int permissionCheckWrite = ContextCompat.checkSelfPermission(this,
72                WRITE_EXTERNAL_STORAGE);
73
74        //If not, then get the permission now
75        if(permissionCheck != 0){requestPermission();}
76        if(permissionCheckWrite != 0){requestPermission();}
77
78        //Screen shall stay on at any time
79        getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
80
81        //Get the cameraView
82        setContentView(R.layout.show_camera);
83
84        //Connect OpenCV and the cameraView
85        mOpenCvCameraView = (JavaCameraView) findViewById(R.id.show_camera_activity_java_surface_view);
86
87        //Set the view to visible
88        mOpenCvCameraView.setVisibility(SurfaceView.VISIBLE);
89
90        //Start preview from camera
91        mOpenCvCameraView.setCvCameraViewListener(this);
92
93        //Hide the Actionbar
94        hideSystemUI();
95    }
```

Abbildung 4.5: onCreate-Methode

Die Methoden onPause und onDestroy schließen den View, wenn die Anwendung pausiert oder gestoppt wird, und onResume lädt die OpenCV-Bibliothek mit Hilfe der BaseLoaderCallback-Klasse, wenn die Anwendung nach einer Pause fortgesetzt werden soll.

```
97      private BaseLoaderCallback mLoaderCallback = new BaseLoaderCallback(this) {...};
13
14
15      @Override
16      public void onPause()
17      {
18          super.onPause();
19          if (mOpenCvCameraView != null)
20              mOpenCvCameraView.disableView();
21      }
22
23      @Override
24      public void onResume()
25      {
26          super.onResume();
27          if (!OpenCVLoader.initDebug()) {
28              Log.d(TAG, "Internal OpenCV library not found. Using OpenCV Manager for initialization");
29              OpenCVLoader.initAsync(OpenCVLoader.OPENCV_VERSION_3_2_0, this, mLoaderCallback);
30          } else {
31              Log.d(TAG, "OpenCV library found inside package. Using it!");
32              mLoaderCallback.onManagerConnected(LoaderCallbackInterface.SUCCESS);
33          }
34      }
35
36      public void onDestroy() {
37          super.onDestroy();
38          if (mOpenCvCameraView != null)
39              mOpenCvCameraView.disableView();
40      }
41
```

Abbildung 4.6: onPause, onDestroy, onResume

4.3 Augenerfassung

4.3.1 Erster Versuch: Houghtransformation

Im ersten Versuch das Auge halbwegs stabil zu erfassen, verwendete ich die HoughCircles-Methode von OpenCV. Im folgenden Screenshot ist der dazugehörige Code zu sehen.

```
public Mat onCameraFrame(CvCameraViewFrame inputFrame) {

    // TODO Auto-generated method stub
    mGray = inputFrame.gray();
    //mCanny = inputFrame.gray();

    // Rotate mRgba 90 degrees
    Core.transpose(mGray, mRgbaT);
    Imgproc.resize(mGray, mRgbaF, mRgbaF.size(), 0,0, 0);

    //Smoothing
    Imgproc.blur(mGray, mGray,new Size(7,7), new Point(2, 2));
    //Imgproc.Canny(mGray, mCanny, edges,1, 50);
    Imgproc.HoughCircles(mGray, circles, Imgproc.CV_HOUGH_GRADIENT, 1, 1000, 50, 90, 0, 1000);

    int radius = 0;

    int totalCirclesDetected = 0;
    if (circles.cols() > 0) {
        for (int x = 0; x < Math.min(circles.cols(), 5); x++) {
            double vCircle[] = circles.get(0, x);
            if (vCircle == null)
                break;
            Log.i("Circle :", "Yes ");
            totalCirclesDetected++;
            Point center = new Point((int) vCircle[0], (int) vCircle[1]);

            radius = (int) Math.round(vCircle[2]);
            // draw the found circle
            Imgproc.circle(mGray, center, 3, new Scalar(255,255,255), 5);
            Imgproc.circle(mGray, center, radius, new Scalar(255, 255, 255), 2);
        }
    } else {
        Log.i("Circle :", "No");
    }

    return mGray; // This function must return
}
```

Abbildung 4.7: Erfassung per Houghtransformation

Zuerst werden die benötigten Matrizen mit dem Inputframe ausgestattet. Da die Canny-Methode, welche in der Houghtransformation der erste Schritt ist, auf Graustufenbildern funktioniert und keine Farbinformationen benötigt, beziehen wir das Inputframe direkt in schwarz-weiß. Danach wird die Matrix, die auf dem Display ausgegeben werden soll, um 90° gedreht, damit das Bild in der waagerechten Haltung des Smartphones richtig ausgerichtet ist. Anschließend wird Bildrauschen mit Hilfe von Glättungsmethoden wie GaussianBlur oder MedianBlur reduziert und letztlich die Houghtransformation angewendet. Diese verlangt einige Parameter. Einerseits das Graustufenbild, das auf Kreise geprüft werden soll, andererseits eine leere Matrix, in der die gefundenen Kreise

über Mittelpunktkoordinaten und Radiuslänge festgehalten werden. Der dritte Parameter hat keine Bedeutung, da es hier keine Alternativen gibt. Der fünfte Parameter bestimmt, wie groß der Abstand zwischen gefundenen Kreisen sein soll. Da wir möglichst nur einen Kreis gefunden haben möchten und zwar die Pupille, gehört hier ein recht hoher Wert rein. Dann folgt der obere Schwellenwert für den internen Canny-Algorithmus, wobei der untere Schwellenwert automatisch auf die Hälfte vom oberen gesetzt wird. Hier gilt die Regel, je kleiner die Schwellenwerte, desto feiner die Kantendetektion und umso mehr potentielle Fehldetektionen. Der achte Parameter beschreibt die Schwelle ab wie vielen, auf den potentiellen Kreis passenden, gefundenen Kantenpunkten ein Kreis als solcher gezählt wird. Auch hier gilt je niedriger der Wert, desto mehr gefundene Kreise. Die letzten beiden Parameter stehen für die untere und die obere Grenze der Länge der gesuchten Radien. Nach der Houghtransformation werden die Kreise aus der dazugehörigen Matrix gelesen, auf dem Ausgabebild mit der circle-Funktion eingezeichnet und diese Matrix wieder zurückgegeben. Die auskommentierte Canny-Matrix diente zum Herausfinden der optimalen Werte für den entsprechenden Schwellenwert in der Houghfunktion. So kann man einsehen, auf welchem Kantenbild die Houghfunktion nach Kreisen sucht. Um passende Werte für die restlichen Parameter herauszufinden, war vorrangig Ausprobieren angesagt.

Nach einigem Durchprobieren von Parametern und unterschiedlichen Glättungsmethoden war mein bestes Ergebnis, dass wenn man auf die Linse der Kamera schaut, die Iris als einziger Kreis erkannt wird. Allerdings habe ich keinen Weg finden können, wie man das Auge nicht verliert, sobald man nicht mehr genau in die Linse schaut und die Iris/Pupille nicht mehr einen idealen Kreis darstellt. Darüber hinaus ist die Hough-Methode generell sehr rechenlastig, weshalb die Bilderwiederholrate nie über 6 Bilder die Sekunde steigen konnte. Aus diesen Gründen habe ich die Strategie geändert und versucht die Pupille über ihre Farbe festzumachen.

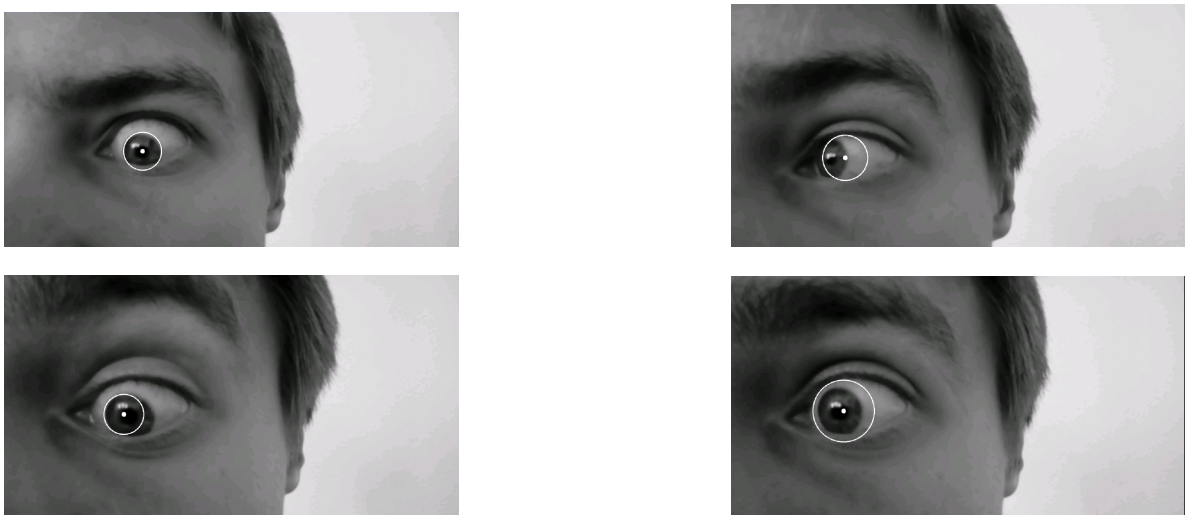
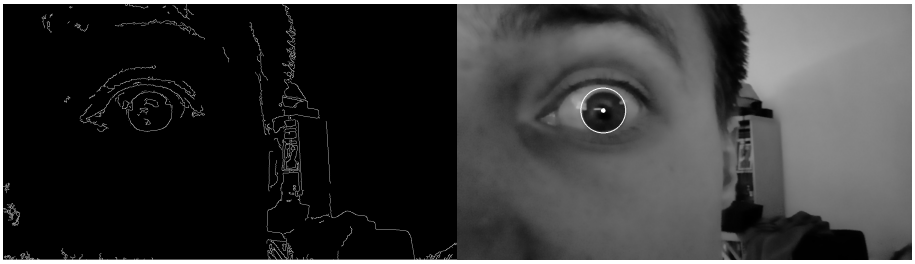


Abbildung 4.8: Augenerkennung per HoughCircles



(a) Kantenbild nach Canny **(b)** Erkannter Kreis durch HoughCircles

Abbildung 4.9: Erfolgreiche Detektion

4.3.2 Tracking an Hand der Farbe

Wie im Abschnitt 2.4 bereits erwähnt, sind die dunkelsten Partien im Augenbereich mit Abstand die Pupille und Augenwimpern. Deshalb folgte die Idee, die Pupille mit Hilfe eines niedrigen inversen binären Thresholds zu segmentieren. Wenn also nur noch die Pixel dargestellt werden, die minimal weniger dunkel sind als absolut schwarz, dann müssten nur noch die Pupille und Wimpern im Bild zu sehen sein.

```
public Mat onCameraFrame(CvCameraViewFrame inputFrame) {

    // TODO Auto-generated method stub
    mGray = inputFrame.gray();

    //Rotate picture for right orientation
    Core.transpose(mGray, mRgbaT);
    //Simple smoothing
    Imgproc.blur(mGray, mGray, blurSize, blurPoint);

    //Seekbar for Thresholdvalue ranging from 0-20
    seekBarH.setOnSeekBarChangeListener(new SeekBar.OnSeekBarChangeListener() {...});


    //Areasizes for filtering out wrong segments
    double targetArea = 0;
    double minArea = 100;
    double maxArea = 3000;
    boolean eyeFound = false;

    //If threshold got changed
    if(changed) {

        Imgproc.threshold(mGray, mGray, valueH, 255, Imgproc.THRESH_BINARY_INV);


        mGray2 = mGray.clone();
        Imgproc.findContours(mGray2, contours, findContoursHierarchy, Imgproc.RETR_EXTERNAL, Imgproc.CHAIN_APPROX_SIMPLE);


        int possibleCircles = 0;
        //If contours found in the right range, save them
        if (contours.size() > 0) {
            Log.i(CAL,"Contours empty? : NO");
            for (int x = 0; x < Math.min(contours.size(), 10); x++) {
                MatOfPoint c = contours.get(x);
                if (Imgproc.contourArea(c) < maxArea && Imgproc.contourArea(c) > minArea) {
                    possibleCircles++;
                    possibleEyes.add(c);

                } else {
                }

            }
```

Abbildung 4.10: Erfassung per Threshold

Auch hier reicht uns ein Graustufenbild und eine einfache Glättung. Um den richtigen Wert für den Threshold zu finden, brauchen wir eine Leiste, die diesen direkt einstellt. Hierfür nützen wir eine Seekbar, welche einen Wertebereich von 0 bis 20 aufweist. Die Threshold-Methode von OpenCV ist im Vergleich zur Hough-Methode ziemlich simpel. Wir haben eine Eingabe- und Ausgabematrix, welche in unserem Fall die selbe ist. Dann kommt der eigentliche Wert des Thresholds, den wir direkt mit unserer Leiste einstellen, gefolgt von dem Intensitätswert der Pixel, die über dem Schwellwert liegen. Damit wir ein klassisches, schwarz-weißes Binärbild erhalten, ist dieser Wert auf das Maximum von 255 eingestellt. Der letzte Parameter bestimmt die Art des Thresholds, was in unserem Fall der binäre, inverse Threshold ist. Hat der Benutzer den richtigen Wert gefunden, müsste dort, wo im Graustufenbild die Pupille zu sehen war, ein möglichst einsamer, weißer Punkt zu sehen sein. In meinen Versuchen war der optimale Wert normalerweise nicht höher als 5, allerdings kann bei äußerst starker Beleuchtung durchaus sein, dass man hier etwas weiter nach oben muss. Zu Beginn der Anwendung sieht man das Graustufenbild der Frontkamera, sobald man einen Wert über die Leiste eingestellt hat, wechselt die Sicht auf das entsprechende Binärbild. Dann folgt die Segmentierung des Binärbildes unter Anwendung der findContours-Methode von OpenCV. Diese erkennt alle zusammenhängenden Segmente/Konturen des Binärbildes und speichert sie als Punktemengen in der contours-Matrix ab. Um zu entscheiden, welches Segment die Pupille ist, filtern wir sie nach der Größe. Da die Beleuchtungssituation, Entfernung der Linse zum Auge und Augenposition direkte Auswirkung auf die Größe des sichtbaren Pupillensegments hat, muss ein Wertebereich hierfür festgelegt werden. Durch Ausprobieren habe ich hier den Bereich zwischen 100 und 3000 Pixeln als optimal bestimmt. Im darauffolgenden Schritt werden also alle gefundenen Segmente, die sich nicht in dem Größenbereich befinden, aussortiert.

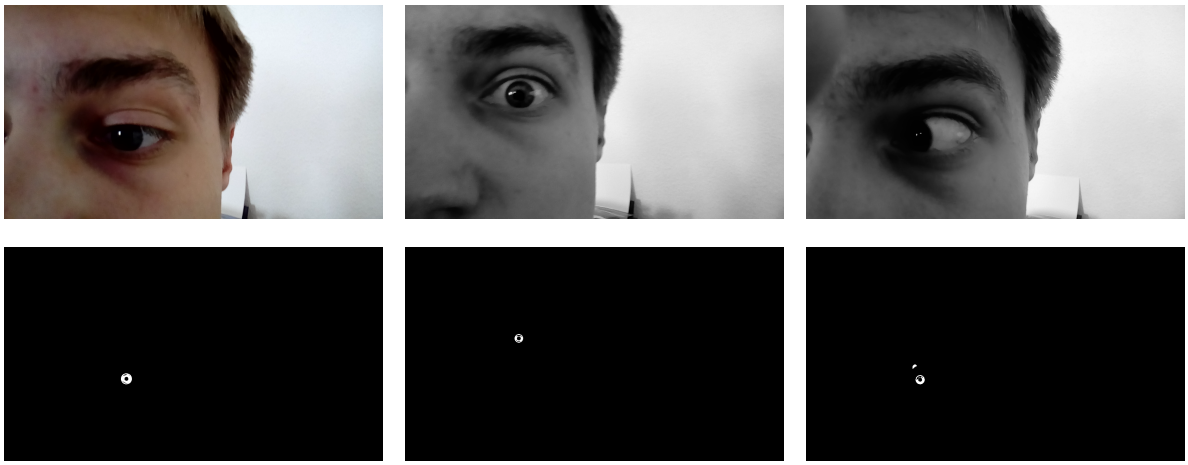


Abbildung 4.11: Augenerkennung per Threshold. Oben Ausgangsbild, unten Ausgabe nach Threshold-Operation

```
        //If multiple eyecandidates found, take the biggest one
        if (possibleCircles >= 1) {
            if (possibleCircles != 1) {
                for (int x = 0; x < possibleEyes.size(); x++) {
                    MatOfPoint c = possibleEyes.get(x);
                    if (Imgproc.contourArea(c) > targetArea) {

                        targetArea = Imgproc.contourArea(c);
                        pupil = c;
                    }
                }
                Log.i(CAL,"Eye found : Yes");

                pupil.convertTo(pupil2f, CvType.CV_32F);

                //draws a circle around the found contour
                Imgproc.minEnclosingCircle(pupil2f, center, radius);
                Imgproc.circle(mGray, center, 3, white, 5);
                Imgproc.circle(mGray, center, (int) radius[0], red, 2);
                eyeFound = true;

            } else {

                pupil = possibleEyes.get(0);
                targetArea = Imgproc.contourArea(pupil);


                Log.i(CAL,"Eye found : Yes");

                pupil.convertTo(pupil2f, CvType.CV_32F);

                //draws a circle around the found contour
                Imgproc.minEnclosingCircle(pupil2f, center, radius);
                Imgproc.circle(mGray, center, 3, white, 5);
                Imgproc.circle(mGray, center, (int) radius[0], red, 2);
                eyeFound = true;
            }
        }


    } else {
        Log.i(CAL,"Eye found : No");
        eyeFound = false;
    }

}
```

Abbildung 4.12: Erfassung per Threshold Fortsetzung

Wenn nun immer noch mehr als ein Kandidat zur Verfügung steht, wird der größte von allen ausgewählt, weil davon auszugehen ist, dass alle kleineren Segmente Bildrauschen entsprechen. Schließlich haben wir unser Pupillensegment definiert und lassen mit Hilfe der minEnclosingCircle-Methode einen Kreis um die Punktemenge berechnen. Diese gibt uns die Koordinaten und den Radius der Pupille, so dass der entsprechende Kreis in das Ausgabebild eingezeichnet werden kann.

4.4 Kalibrierung

```
if(calibration && eyeFound){

    //Drawing the Grid
    Imgproc.line(mGray, line1verA, line1verB, red);
    Imgproc.line(mGray, line2verA, line2verB, red);
    Imgproc.line(mGray, line3verA, line3verB, red);
    Imgproc.line(mGray, line1horA, line1horB, red);
    Imgproc.line(mGray, line2horA, line2horB, red);
    Imgproc.line(mGray, line3horA, line3horB, red);


    //Get the current eyeposition and save it
        switch (calStatus) {
            case 1:
                Imgproc.circle(mGray, fixPoint1, 3, red, 5);
                eyePositions1.add(center.clone());
                break;
            case 2:
                Imgproc.circle(mGray, fixPoint2, 3, red, 5);
                eyePositions2.add(center.clone());
                break;
```

Abbildung 4.13: Einzeichnen vom Gitter und Aufnehmen der Augenpositionen

```
        case 9:
            Imgproc.circle(mGray, fixPoint9, 3, red, 5);
            eyePositions9.add(center.clone());
            break;
    }

    //When we have 10 positions, we calculate the average of them and take it as the reference point
    if (calCalculator == 10) {
        eyePosition1 = calcAvgPoint(eyePositions1);

        for(int i = 0; i < eyePositions1.size(); i++){
            Log.i(CAL,"Positions 1: " + eyePositions1.get(i).toString());
        }
        Log.i(CAL ,"Final Position 1: " + eyePosition1.toString());
        calStatus = 2;
        calibration = false;
    }
    if (calCalculator == 20) {
        eyePosition2 = calcAvgPoint(eyePositions2);

        for(int i = 0; i < eyePositions2.size(); i++){
            Log.i(CAL ,"Positions 2: " + eyePositions2.get(i).toString());
        }
        Log.i(CAL ,"Final Position 2: " + eyePosition2.toString());
        calStatus = 3;
        calibration = false;
    }
    if (calCalculator == 30) {
        eyePosition3 = calcAvgPoint(eyePositions3);
```

Abbildung 4.14: Nach 10 Positionen wird der Durchschnitt errechnet und als finaler Referenzpunkt abgespeichert

Für die Kalibrierung unterteilen wir den Bildschirm in ein Gitter mit neun gleichgroßen Feldern. Der Mittelpunkt der Felder dient als der Fixpunkt, auf den man schaut, wenn die entsprechenden Pupillenkoordinaten gesammelt werden. Da die Kalibrierung eine Abfolge von Zuständen ist und wir zu jedem Zeitpunkt an die onCameraFrame-Methode gebunden sind, habe ich den Kalibrierungsprozess in Form eines Zustandsautomaten umgesetzt. Wenn man erstmalig auf den Kalibrierungsbutton drückt, erscheint das Gitter auf dem Display mit einem Punkt in der Mitte des ersten Feldes links oben. Drückt man ein zweites Mal auf den Button, werden in den zehn darauffolgenden Frames die erfassten Mittelpunkte der Pupille in einer Liste gespeichert. Anschließend gelangen wir wieder in den Zustand davor, nur dass jetzt im zweiten Feld der Mittelpunkt angezeigt wird. Wenn man den neuen Punkt fixiert hat, drückt man nochmals auf den Knopf und die nächsten zehn Punkte der Pupille werden festgehalten, bevor der Fixpunkt in die Mitte vom dritten Feld springt und der Nutzer wieder den Knopf betätigen muss. Dieser Prozess wird neun Mal wiederholt, so dass die Anwendung danach die nötigen Daten hat, um den Blick auf den Bildschirm abzubilden. Der aktuelle Stand der Kalibrierung wird an Hand des Zählers calCalculator festgehalten und die Fallunterscheidung bei den neun Feldern wird durch den Zähler calStatus umgesetzt. Für jede der neun Augenpositionen ist eine ArrayList angelegt, in welche je zehn Punkte abgespeichert werden. Jedes Mal wenn ein Punkt abgespeichert wurde, wird calCalculator um einen Wert inkrementiert. Haben wir zehn Punkte gesichert, ist calCalculator auf einem Wert, der durch zehn teilbar ist. In diesem Fall wird der endgültige Referenzpunkt so bestimmt, in dem der Durchschnitt aller x- und y-Koordinaten der zehn Punkte errechnet wird. Dieser Punkt wird gespeichert, der calStatus um eins erhöht und der Kalibrierungsstatus calibration auf false gesetzt. Durch Betätigen des Buttons werden wieder zehn Augenpunkte festgehalten, deren Durchschnitt errechnet und als Referenzpunkt gespeichert. Im letzten Schritt erreicht der Kalibrierungszähler den Wert 90 und hat somit alle Daten, die für das Eyetracking nötig sind. Die endgültigen Referenzpunkte werden alle in eine Liste gespeichert und der Kalibrierungsstatus so geändert, dass das Gitter verschwindet und der Calibration-Button seine Funktion verliert. Der boolean-Wert calibrated wird auf true gesetzt und somit wird in jedem Durchlauf der onCameraFrame-Methode das Eye-Tracking an Hand der aktuellen Pupillenposition und der Liste mit den Referenzpunkten umgesetzt.

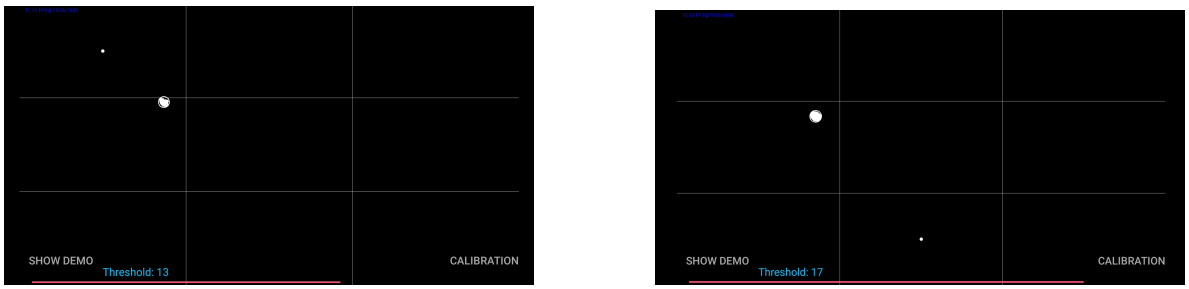


Abbildung 4.15: Ausschnitte aus dem Kalibrierungprozess

```
    calCalculator++;

if (calCalculator == 90) {
    eyePosition9 = calcAvgPoint(eyePositions9);

    for(int i = 0; i < eyePositions9.size(); i++){
        Log.i(CAL ,"Positions 9: " + eyePositions9.get(i).toString());
    }
    Log.i(CAL ,"Final Position 9: " + eyePosition9.toString());
    calStatus = 1;
    calibration = false;
    calibrated = true;
    calCalculator = 1;
    startingInt = 3;
    Log.i(CAL,"Calibration finished. Calibrated Positions: ");

    Log.i(CAL,"Position1: " + eyePosition1.toString());
    Log.i(CAL,"Position2: " + eyePosition2.toString());
    Log.i(CAL,"Position3: " + eyePosition3.toString());
    Log.i(CAL,"Position4: " + eyePosition4.toString());
    Log.i(CAL,"Position5: " + eyePosition5.toString());
    Log.i(CAL,"Position6: " + eyePosition6.toString());
    Log.i(CAL,"Position7: " + eyePosition7.toString());
    Log.i(CAL,"Position8: " + eyePosition8.toString());
    Log.i(CAL,"Position9: " + eyePosition9.toString());


    allEyePositions.add(eyePosition1);
    allEyePositions.add(eyePosition2);
    allEyePositions.add(eyePosition3);
    allEyePositions.add(eyePosition4);
    allEyePositions.add(eyePosition5);
    allEyePositions.add(eyePosition6);
    allEyePositions.add(eyePosition7);
    allEyePositions.add(eyePosition8);
    allEyePositions.add(eyePosition9);

}
```

Abbildung 4.16: Wurde der letzte Referenzpunkt berechnet, wird die Kalibrierung abgeschlossen

4.5 Abbildung des Blickpunktes

```
private void eyetracking(List<Point> calibratedPositions, Point center){
    double min = 100000.;
    double secondmin = 100000.0;
    double thirdmin = 100000.0;

    for(int i = 0; i < 9; i++){
        double dist = distanceAB(center, calibratedPositions.get(i));

        if(dist < min){
            min = dist;
            closest[0] = i;
        }
        if(dist < secondmin && dist != min){
            secondmin = dist;
            closest[1] = i;
        }
        if(dist < thirdmin && dist != min && dist != secondmin){
            thirdmin = dist;
            closest[2] = i;
        }
    }

    Log.i("Eyetracker", "Closest Field 1: " + closest[0]);
    Log.i("Eyetracker", "Closest Field 2: " + closest[1]);
    Log.i("Eyetracker", "Closest Field 3: " + closest[2]);

    switch (closest[0]) {
        case 0:
            Imgproc.rectangle(mGaze, rect1A, rect1B, red, 8);
            trackedField = 0;
            break;
        case 1:
            Imgproc.rectangle(mGaze, rect2A, rect2B, red, 8);
            trackedField = 1;
            break;
        case 2:
            Imgproc.rectangle(mGaze, rect3A, rect3B, red, 8);
            trackedField = 2;
            break;
        case 3:
            Imgproc.rectangle(mGaze, rect4A, rect4B, red, 8);
            trackedField = 3;
            break;
```

Abbildung 4.17: Abbildung des Blickbereiches

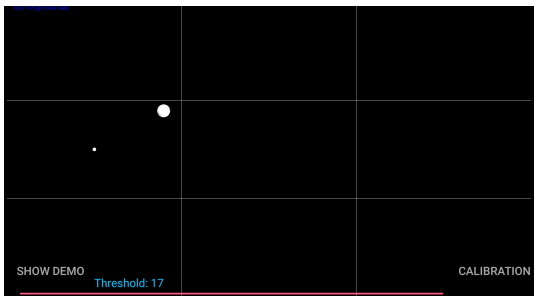
Ist die Kalibrierung abgeschlossen, wird die eyetracking-Methode in jedem Durchlauf von onCameraFrame ausgeführt, falls die Pupille detektiert ist. Die eyetracking-Methode braucht die Liste der Referenzpunkte und den aktuellen Mittelpunkt der Pupille als Parameter. Dann werden die Entfernungen der Referenzpunkte zum aktuellen Punkt errechnet. Der Referenzpunkt, der die kürzeste Distanz aufweist, entspricht dem Feld, in dem sich der Blick des Nutzers befindet. Dieses Feld wird in der Ausgabematrix markiert. Wegen Gründen der Fehlerbehebung werden die nächstgelegenen drei Referenzpunkte berechnet, angezeigt wird aber nur das nächste.

4.6 Funktionen und Grafische Oberfläche

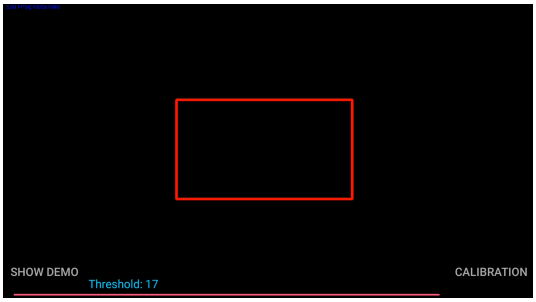
Wenn man die Anwendung aufmacht, sieht man auf dem Bildschirm die Sicht der Frontkamera, zwei Buttons am linken und rechten, unteren Rand und eine Leiste entlang des unteren Randes, unterhalb der beiden Buttons. Die Leiste dient dazu den Wert des Thresholds einzustellen, wobei die Leiste Werte zwischen 0 und 20 annehmen kann. Da die Erkennung der Pupille auf dem Threshold basiert, sind die beiden sichtbaren Buttons bis zum erstmaligen Verändern der Leiste funktionslos. Hat man einen Wert eingestellt, ändert sich die Kameraausgabe von einem Graustufen- zu einem Binärbild. Wenn der richtige Wert gefunden wurde, ist ein weißer Punkt an der Stelle der Pupille zu sehen, um den ein weißer Kreis gezeichnet ist. An dieser Stelle kann man die Calibration-Taste drücken und das Kalibrierungsgitter erscheint über dem Bild. Um die Kalibrierung eines Referenzpunktes zu starten, muss die Calibration-Taste gedrückt werden. Das Feld, in dem sich ein Mittelpunkt befindet, ist das Feld welches als nächstes Kalibriert wird. Sobald das letzte Feld Kalibriert ist, beginnt das Eye-Tracking und das Binärbild wird ausgeblendet. Nun wird das betrachtete Feld jeweils rot umrandet und die ShowDemo-Taste kann aktiviert werden. Mit dieser Taste wird der Algorithmus zur Bestimmung der Genauigkeit gestartet.

(a) Bildansicht zu Beginn

(b) Eingestellter Threshold



(c) Kalibrierungsprozess



(d) Nach durchgeführter Kalibrierung

Abbildung 4.18: Ausschnitte der GUI

4.7 Lösung des Beleuchtungsproblems

Nachdem die Software funktionsfähig war, blieben wenige Wochen um sich dem Hardwareproblem mit der Beleuchtung im Headset zu widmen. Die erste Idee für eine gute, zuverlässige Beleuchtung in einem VR-Headset lag auf der Hand. Wie die großen Vorbilder von FOVE oder das Projekt an der TU Braunschweig [SGE+15] vorgemacht haben, ist die praktischste Beleuchtungsmethode im Kontext von VR-Eye-Tracking mit Infrarotlicht. Der ausschlaggebende Aspekt hierbei ist, dass Infrarotlicht zwar für das menschliche Auge unsichtbar ist, für Kamerasensoren allerdings nicht. Das gilt auch für Smartphonekameras, was man ganz einfach testen kann, indem man den Infrarotsender an einer Fernbedienung filmt, während man Tasten betätigt. Wie beim Projekt in Braunschweig wäre die optimale Lösung ein Ring mit verbauten Infrarot-LEDs, der direkt vor der Linse angebracht wird und sich im aufgesetzten Zustand direkt vor dem Auge des Nutzers befindet. Um solch eine Beleuchtung umzusetzen, braucht es jedoch ein wenig handwerklich-elektrotechnische Erfahrung insbesondere im Löten, welche mir fehlt. Ich habe zwar nach fertigen Infrarotleuchten gesucht, welche man im Headset platzieren könnte, aber in der Größenordnung ließ sich leider nichts geeignetes finden.

Abbildung 4.19: Auch Smartphonekameras erkennen Infrarotlicht.

Die andere Möglichkeit ist herkömmliches, für den Menschen sichtbares Licht zu verwenden. Einerseits könnte man versuchen ein Loch in den vorderen Bereich des Headsets zu bohren, so dass Licht aus der Umgebung durch das Loch auf das Auge fällt. Aber abgesehen davon, dass diese Option das Beleuchtungsproblem nicht endgültig löst, weil man dann immer noch darauf achten muss, dass das Umgebungslicht stimmt, dürfte man sich bei der Benutzung auch kaum bewegen, da dies Einfluss auf die Beleuchtung hätte. Also suchte ich nach einer Leuchte, die möglichst klein und betriebsbereit ist, und wurde fündig. Auf Amazon fand ich die AtomLight von Polymath Products [pol]. Dies ist eine Mikro-LED, konzipiert als kleiner Schlüsselanhänger. Dreht man den Schlüsselring nach rechts, geht das Licht an, dreht man ihn nach links, geht es aus. Mit den Maßen 1,4x1x1 cm hat es die optimale Größe für unser Vorhaben. Glücklicherweise verfügt das Headset über kleine Luftschlitze direkt unterhalb der Augen. Dadurch ließ sich die AtomLight so einbauen, dass der Schlüsselring zum An- und Ausschalten unterhalb des Headsets heraushängt, während die LED das Auge senkrecht beleuchtet.

Abbildung 4.20: Links: AtomLight, Rechts: Headset mit eingebautem AtomLight

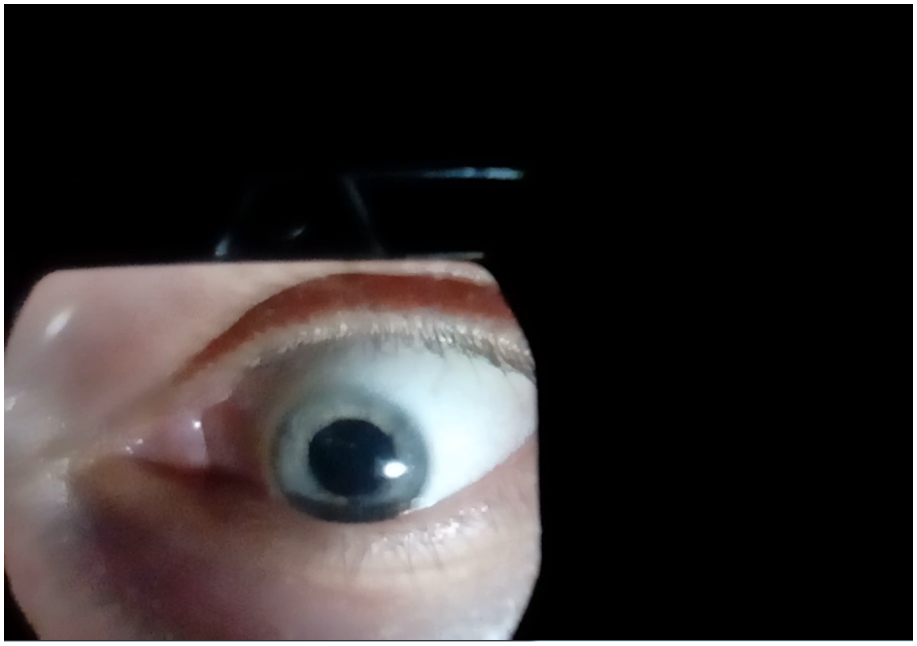


Abbildung 4.21: Kameraperspektive im Headset mit Beleuchtung

5 Ergebnisse

5.1 Performanz

Die Performanz lässt sich für diese Anwendung recht einfach und präzise an der Bildwiederholungsrate während der Ausführung festhalten. Um einen Vergleich zu haben, ließ ich die onCameraFrame-Methode ohne jegliche Modifikationen laufen, sodass die Ausgangsframes direkt an den View weitergegeben wurden. Die Bildfrequenz lag so bei 15-17 Frames die Sekunde. Dieser Wert entspricht dem Maximum für das Gerät, wenn man die Kamera über OpenCV betreibt. Beim Ausführen der Anwendung sinkt die Bildfrequenz zwar auf 6-9 Frames die Sekunde, allerdings ist das noch ein Wert, der bei der Verwendung nicht allzu sehr irritiert. Insbesondere im Vergleich zum ersten Ansatz mit der Houghtransformation, wo 6 Frames die Sekunde das Maximum war, ist das ein akzeptabler Wert. Das System auf dem getestet wurde, verfügt über einen MT6753 Octa Core Prozessor mit einer Taktrate von 1,3 GHz und einen Arbeitsspeicher mit 2 GB RAM [zte].

5.2 Genauigkeit

Um die Genauigkeit und generelle Funktionstüchtigkeit der Eyetracking-Anwendung zu testen, habe ich einen simplen Abfragealgorithmus geschrieben. Dieser zeigt sequentiell 10 unterschiedliche Punkte mit zufälligen Koordinaten für jeweils 20 Frames an. Die ersten 5 dienen dazu, den aktuellen Punkt zu fixieren, während in den restlichen 15 Frames abgeglichen wird, ob das erfasste Feld mit dem Feld, in dem sich der angezeigte Punkt befindet, übereinstimmt. Das Ergebnis wird im Log gespeichert.

```
if(calibrated && showDemo){

    if(demoCal % 20 == 0) {
        Random rndm = new Random();
        Random rndm2 = new Random();
        int i1 = rndm.nextInt(1920 - 0);
        int i2 = rndm2.nextInt(1080 - 0);
        demoP.x = i1;
        demoP.y = i2;

        appendLog("Showing Point: " + demoP.toString());
        Imgproc.circle(mGaze, demoP, 3, white, 5);

        double closest = 10000.;

        for(int i = 0; i < 9; i++) {
            double dist = distanceAB(demoP, fixPointAll.get(i));
            if(dist < closest){
                closest = dist;
                actField = i;
            }

        }

    }

    if(demoCal % 20 < 5){
        Imgproc.circle(mGaze, demoP, 3, white, 5);
    }
```

Abbildung 5.1: Algorithmus für die Demonstration der Genauigkeit

Der Integerwert demoCal dient als Zähler für den Fortschritt des Prozesses und wird nach jedem einzelnen Frame inkrementiert. Ist er durch 20 teilbar, bedeutet das, dass ein neuer Punkt definiert werden muss. Dafür werden die x- und y-Koordinaten per Zufallszahl aus dem Wertebereich der Resolution des Displays (1920x1080 Pixel) bestimmt, der Punkt eingezeichnet und das Feld, in dem der Punkt sich befindet, über die Entfernung zu den Mittelpunkten aller Felder errechnet. In den vier darauffolgenden Frames wird nur der Punkt eingezeichnet, damit man Zeit hat, diesen zu fixieren.

```
        if(demoCal % 20 > 4){
            Imgproc.circle(mGaze, demoP, 3, white, 5);
            appendLog("Located in Field: " + actField);
            appendLog("Field being watched: " + trackedField);
            if(actField == trackedField){
                appendLog("Right Field found");
                demoResults[frameCounter] = true;
            }else{
                appendLog("False Field found");
                demoResults[frameCounter] = false;
            }

            frameCounter++;
        }
        demoCal++;

        if(demoCal == 200){
            showDemo = false;
            int positiveDetections = 0;
            for(boolean x : demoResults){
                if(x == true){
                    positiveDetections++;
                }
            }
            appendLog(positiveDetections + " correctly tracked Fields of " + frameCounter);
            showDemo = false;
        }
    }
```

Abbildung 5.2: Algorithmus für die Demonstration der Genauigkeit Fortsetzung

Während der letzten 15 Frames wird kontrolliert, ob das Feld detektiert wurde, in dem sich der zufällige Punkt befindet. Wenn dem so ist, wird im Array demoResults ein true abgelegt, falls nicht ein false. Hat der Zähler demoCal den Wert 200 erreicht, ist der Prozess am Ende und es wird kontrolliert, wie viele Übereinstimmungen festgestellt wurden.

Bei optimaler Beleuchtung habe ich Ergebnisse mit über 140 von 150 möglichen Treffern erhalten. Allerdings muss man bedenken, dass es ein Zufallsalgorithmus ist und es somit keine Kontrolle darüber gibt, wie viele Punkte in Nähe der Grenzlinie zweier Felder erscheinen. Die Grenzbereiche sind logischerweise Fehleranfälliger als die Innenbereiche der Felder.

5.3 Lösungsansätze für die Software

Obwohl das Auge mit der neuen Leuchte nun klar und deutlich von der Kamera im Headset aus zu erkennen ist, entspricht die Beleuchtung nicht dem, wovon bei der Software ausgegangen wurde. Einerseits schwankt nun der Helligkeitswert der Pupille deutlich in Abhängigkeit von ihrer Position, wodurch es nun nicht mehr möglich ist, diese durch den einfachen inversen Threshold klar zu segmentieren. Andererseits wird die Lichtquelle nun sehr deutlich vom Auge reflektiert, wodurch ein heller Punkt auf der Pupille zu sehen ist. Die Position des Punktes steht in Abhängigkeit zur Position vom Auge, was ihn prinzipiell als Anhaltspunkt für Eye-Tracking geeignet macht. Das Problem besteht hier allerdings darin, dass wenn man zur rechten Seite des Bildschirms schaut, also

von der Kameralinse weg, befindet sich der Reflexionspunkt im Bild nicht mehr auf dem dunklen Pupillen/Iris-Bereich sondern auf der weißen Fläche des Augapfels. Ich habe mit einem hohen binären Threshold diesen Reflexionspunkt gut segmentieren können, aber eben nur für die Fälle, wo er sich auf dem dunklen Bereich befindet. Wenn es auf die weiße Fläche übergeht, verschwimmt der segmentierte Bereich und ändert seine Form, da die Helligkeitswerte zwischen Augenfläche und Reflexion zu nah beieinander liegen. Da dies nicht ausreicht, um die rechte Seite des Blickfeldes zu kalibrieren, ist Eye-Tracking alleine an Hand dieses Merkmals nicht möglich.

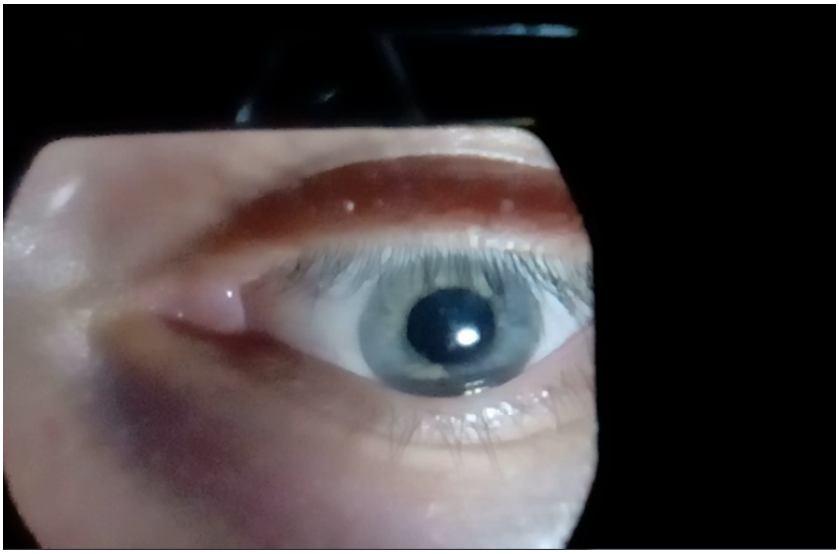


Abbildung 5.3: Reflexion der Lichtquelle beim Blick in Richtung Kamera

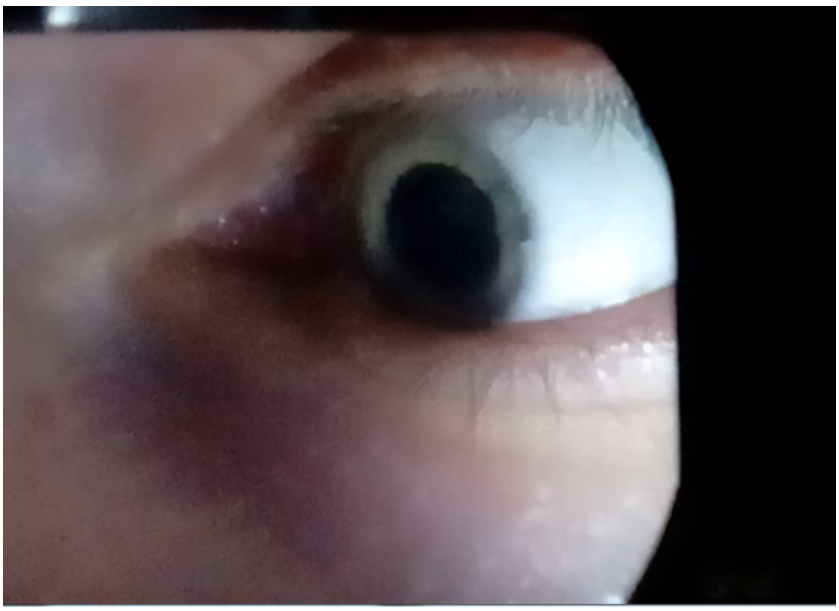


Abbildung 5.4: Reflexion verschwindet beim Blick in entgegengesetzter Richtung zur Kamera

6 Zusammenfassung und Ausblick

Die Idee Eye-Tracking auf einem Smartphone im VR-Szenario umzusetzen, war die Ausgangsmotivation für diese Arbeit. Dafür muss einerseits eine Software entwickelt werden, welche das Eye-Tracking umsetzt, und andererseits eine angemessene Beleuchtungsmethode für das Auge gefunden werden, da das Auge für die Kamera sonst nicht klar genug erkennbar ist. Um eine Softwarebasis zu haben, entwickelte ich zuerst die Eye-Tracking-Software unter Anwendung von Methoden der Computer Vision und der OpenCV Bibliothek. Die Erfassung vom Auge wurde zuerst mit der Houghtransformation für Kreise umgesetzt, welche sich allerdings als unzuverlässig und sehr rechenlastig entpuppte. Also suchte ich nach einer Möglichkeit, das Auge an Hand der Farbe und nicht nur an Hand der Form zu erfassen. Der zweite Ansatz segmentiert die Pupille mit Hilfe ihrer dunklen Farbe ausgehend davon, dass das Auge gut ausgeleuchtet ist und somit als dunkelster Bereich nur die Pupille übrig bleibt. Dies geschieht mit einem sehr niedrigen inversen binären Threshold, der alle Pixel außer den dunkelsten ignoriert. Schließlich bleibt nur das Pupillensegment über, so dass ein Kreis um das Segment das Auge gut genug repräsentiert. Im zweiten Schritt findet eine Kalibrierung statt, in der der Eye-Tracker sich die Positionen der Pupille speichert, wenn der Nutzer auf die zu trackenden Bereiche schaut. So hat der Eye-Tracker ein kleines Feld aus Pupillenkoordinaten, in dem sich die erfasste Pupille bewegt. Das beobachtete Feld wird durch die geringste Distanz vom erfassten Pupillenpunkt zu den kalibrierten Referenzpunkten bestimmt. Das Beleuchtungsproblem wurde schließlich mit einer kleinen LED-Leuchte im Augenbereich des Headsets gelöst. Da die Beleuchtungssituation sich hier jedoch stark von der unterscheidet, von welcher bei der Entwicklung des Eye-Trackers ausgegangen wurde, funktioniert dieser im Headset leider nicht mehr. Es ist aber offensichtlich, dass das Problem letztlich lösbar ist. Ansätze hierfür wären zum Beispiel andere Beleuchtungsmethoden zu verwenden wie z.B. Infrarotlicht, vielleicht würde es aber auch reichen, einfach die Position der Beleuchtungsquelle zu ändern. Andere Headsets bieten womöglich mehr Platz für bessere Beleuchtungssysteme und schließlich lassen sich sicherlich auch robustere Eye-Tracking-Verfahren auf Basis der Androidplattform implementieren, so dass die Anforderungen an die Beleuchtung niedriger sind.

Literaturverzeichnis

[ARM13] A. Al-Rahayfeh, F. Miad. Eye Tracking and Head Movement Detection: A State-of-Art Survey. In *IEEE Journal of Translational Engineering in Health and Medicine 1*. 2013. doi: 10.1109/JTEHM.2013.2289879. (Zitiert auf Seite 18)

[Duc07] A. Duchowski. *Eye Tracking Methodology: Theory and Practice*, Kapitel 5. Springer, 2007. (Zitiert auf Seite 13)

[FTBK16] W. Fuhl, M. Tonsen, A. Bulling, E. Kasneci. Pupil detection for head-mounted eye tracking in the wild: an evaluation of the state of the art. In *Machine Vision and Applications*. 2016. (Zitiert auf Seite 18)

[get] getfove.com. FOVE | Eye Tracking VR Headset. URL http://www.getfove.com/. (Zitiert auf Seite 10)

[Gie17] H. Gieselmann. Unknown Fate: Erstes VR-Spiel mit Eye-Tracking. *heise.de*, 2017. URL https://www.heise.de/newsticker/meldung/Unknown-Fate-Erstes-VR-Spiel-mit-Eye-Tracking-3810176.html. (Zitiert auf Seite 10)

[Goo] Google Inc. Google Cardboard – Google VR. URL https://vr.google.com/cardboard/. (Zitiert auf Seite 9)

[GW06] R. C. Gonzalez, R. E. Woods. *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006. (Zitiert auf den Seiten 15, 16 und 17)

[mar] marginz. Camera2. URL http://www.marginz.co.nz/snap/?page_id=152. (Zitiert auf Seite 21)

[pol] polymath. AtomLight. URL https://www.amazon.co.uk/AtomLight-Keyring-Kit-marking-Function-UK-made/dp/B0733DXWBN. (Zitiert auf Seite 38)

[Sau16] M. Sauter. Oculus übernimmt Eye-Tracking-Startup. *golem.de*, 2016. URL https://www.golem.de/news/virtual-reality-oculus-uebernimmt-eye-tracking-startup-1612-125297.html. (Zitiert auf Seite 10)

[SGE+15] M. Stengel, S. Grogorick, M. Eisemann, E. Eisemann, M. A. Magnor. An Affordable Solution for Binocular Eye Tracking and Calibration in Head-mounted Displays. In *Proceedings of the 23rd ACM international conference on Multimedia*, S. 15–24. ACM, 2015. (Zitiert auf den Seiten 10 und 38)

[SGM08] H. Schröder, F. Groth, J. Mennenöh. Eye-Tracking im Einzelhandel - Ein Leitfaden für die Blickaufzeichnung. *Marketing Review St. Gallen*, (6):38–42, 2008. (Zitiert auf Seite 9)

[STZP11] M. Soltani, S. Toosi Zadeh, H. Pourreza. Fast and Accurate Pupil Positioning Algorithm using Circular Hough Transform and Gray Projection. 5, 2011. (Zitiert auf Seite 17)

[tea] O. team. OpenCV. URL http://opencv.org/. (Zitiert auf Seite 21)

[WP08] M. Wedel, R. Pieters. A review of Eye-Tracking research in marketing. In N. K. Malhotra, Herausgeber, *Review of Marketing Research*, Band 4, S. 123–147. M.E.Sharpe, 2008. (Zitiert auf Seite 9)

[Zim] U. Zimmermann. Lexikon Eintrag - Sakkade. URL https://eyetracking.ch/glossar-sakkade/. (Zitiert auf Seite 13)

[zte] zte. ZTE Blade V7 - Herstellerseite. URL https://www.ztemobile.de/product/blade-v7/. (Zitiert auf den Seiten 20 und 41)

Alle URLs wurden zuletzt am 29. 09. 2017 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift