h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Hochschule Darmstadt

– Fachbereich Informatik –

Entwicklung eines Prototypen der visuellen Hörhilfe

Abschlussarbeit zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

vorgelegt von

Marius Dege

Matrikelnummer: 751244

Referent	:	Prof. Dr. Stefan Rapp
Korreferent	:	Prof. Dr. Ronald Moore

ERKLÄRUNG

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, 20. Juni 2020

Marius Dege

ABSTRACT

This work will document the functionality and further development of the visual hearing aid that has been in development by Udo Gebelein and Prof. Dr. Stefan Rapp for a while. Beyond the documentation of the work from Udo and Stefan, we will talk about the basic idea of the visal hearing aid, why there is a need for such a device. There will be a brief overview on the underlying technology aswell. Furthermore a mobile App will be created in combination with an early, wearable prototype that will sit inside a hat. The development of the mobile app will be done in the Qt Framework to allow cross compiling on different devices in the future. Because there are many ways to create such an app with existing openGL code, an overview on different possible solutions will be given. Those solutions will be compared in terms of usability, advantages and best practice. With the mobile app and the hat, it is possible to test and try the hearing aid in public, without anyone noticing it. Goal is to give the reader insights of the product and technology while having a prototype that is mobile and can easily be used with different devices for visualization.

ZUSAMMENFASSUNG

Seit einiger Zeit entwickeln Herr Udo Gebelein und Prof. Dr. Stefan Rapp eine visuelle Hörhilfe. Diese Arbeit dokumentiert die Funktionalität und Weiterentwicklung dieser Hörhilfe. Über die Dokumentation der bisherigen Arbeit hinaus wird auf die Grundidee des visuellen Hörgerätes eingegangen und erläutert, warum es einen Bedarf für ein solches Gerät gibt. Dazu verschafft sie einen kurzen Überblick über die zugrunde liegende Technologie. Darüber hinaus wird eine mobile App in Kombination mit einem frühen, tragbaren Prototypen erstellt, der in einer Kopfbedeckung sitzen wird. Die mobile App wird mit dem Qt Framework realisiert, um mit Cross-Compilierung verschiedenen Geräten anzusprechen. Da es viele Möglichkeiten gibt, eine solche Anwendung mit bereits vorhandenem OpenGL-Code zu erstellen, wird ein Überblick über verschiedenen möglichen Lösungen gegeben und diese im Bezug auf Benutzerfreundlichkeit, Vorteile und bewährte Verfahren verglichen. Mit der mobilen App und der Kopfbedeckung ist es möglich, die Hörhilfe in der Öffentlichkeit zu testen, ohne dass diese für aussenstehende erkennbar ist. Ziel ist es, dem Leser Einblicke in das Produkt und die Technologie zu geben und gleichzeitig einen Prototypen zu entwickeln, der mobil ist und leicht mit verschiedenen mobilen Geräten verwendet werden kann.

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

LISTINGS

ABKÜRZUNGSVERZEICHNIS

API Application Programming Interface

QML Qt Modeling Language

ITD Interaurale Zeit-differenz

ILD Interaurale Pegeldifferenz

HRTF Head-Related-Transfer-Function

SRP-PHAT Steered Response Power with Phase Transform

HSDA Hierarchical Searchwith Directivity model and Automatic calibration

SSL Sound Source Localization

SST Sound Source Tracking

GUI Graphical User Interaface

JSON JavaScript Object Notation

ODAS Open embeddeD Audition System

IMU inertial measurement unit

SMD Surface Mounted Devices

HAL Hardware Abstraction Layer

GPS Global Positioning System

Teil I

THESIS

1 EINLEITUNG

Der Mensch besitzt viele Fähigkeiten, um seine Umgebung wahrzunehmen. Eine bedeutende Fähigkeit in diesem Zusammenhang ist das Hören. Unser Gehör ist ein sehr komplexes Zusammenspiel aus mehreren Organen. Es besitzt die Möglichkeit Geräusch-Signale zu empfangen und zu verarbeiten. Das Gehör ist bis heute nicht in seiner ganzen Funktionsweise der Signalverarbeitung erforscht. Bisher sind technische Ansätze nicht in der Lage die Leistungsfähigkeit des Gehörs zu erreichen. [4] Wir nutzen unser Gehör für die Kommunikation, aber auch zur Orientierung. Wie viele andere Fähigkeiten, sehen wir das Hören als selbstverständlich an. Erst wenn man in seiner Hörfähigkeit eingeschränkt ist, merkt man, wie wichtig das Hören ist. Es beeinträchtigt nicht nur die Geräuschorientierung, sondern ist auch wichtig für die Sprachentwicklung. So haben Menschen mit angeborenem Hörfehler auch oft Probleme bei der Sprachentwicklung. [6]

1.1 MOTIVATION

Ein Verlust oder eine Einschränkung des Hörens kann zu erheblichen Problemen im alltäglichen Leben führen. Betrachtet man den Hörverlust auf einem Ohr, ist die Einschränkung nicht so stark wie bei einem kompletten Hörverlust. Eine gravierende Beeinträchtigung wäre in diesem Fall der Verlust des Richtungshörens. [4] Dadurch leidet nicht nur die Lebensqualität, sondern es kann auch zu Gefahrensituationen zum Beispiel im Straßenverkehr kommen. Da es bisher nur Hörhilfen gibt, die sich auf die Wiederherstellung des Gehörs konzentrieren, soll diese Arbeit eine Hörhilfe weiterentwickeln, die es hörgeschädigten Menschen erlaubt, die Richtung eines Geräusches zu erkennen. Die Besonderheit dieser Hörhilfe ist ein visuelles Feedback an den Benutzer, was bei konventionellen Hörhilfen bislang fehlt.

1.2 ZIEL DER ARBEIT

Diese Arbeit wird sich dem Problem des eingeschränkten Hörens widmen, indem das menschliche Hörvermögen genauer betrachtet und erklärt wird. Es geht hierbei darum, wie das Gehirn Geräusche wahrnimmt und auswertet. Darüber hinaus wird ein Prototyp vorgestellt, der es Menschen mit eingeschränktem Hörvermögen ermöglicht, Geräusche im Raum wieder zu orten. Des weiteren wird die Technologie, die der Prototyp nutzt, vorgestellt. Da der Prototyp noch verhältnismäßig neu ist, wird sich die Arbeit auch mit der Weiterentwicklung des Prototyps beschäftigen. Ziel ist es, dem Leser einen Einblick in die Problematik des Richtungshören zu geben und einen Prototyp zu entwickeln, der im echten Leben einsetzbar ist.

1.3 GLIEDERUNG

Die Arbeit gibt zunächst eine Übersicht der wissenschaftlichen Grundlage des menschlichen Gehörs, um dem Leser ein Verständnis der Problematik zu vermitteln. Die bisherige Arbeit und die zugehörige Technologie werden ebenfalls vorgestellt, bevor der Entwicklungsprozess und die genutzten Werkzeuge des Prototyps dokumentiert werden. Dabei wird die Migration der Software auf mobile Geräte sowie die Weiterentwicklung der Software dokumentiert. Am Ende folgt noch ein Teil mit möglichen neuen Ideen für die Weiterentwicklung der Visualisierung, gefolgt von einem Fazit und einer Bewertung.

2 MENSCHLICHES RICHTUNGSHÖREN

Dieser Abschnitt beschäftigt sich mit dem menschlichen Ohr, damit der Leser ein grundlegendes Verständnis der Thematik erlangt und die Inhalte der Arbeit besser verstehen kann. Die Komplexität des Hörorgans kann an dieser Stelle nicht in seine Gänze dargestellt werden. Daher werden nur die Aspekte, die für das Thema Richtungshören relevant sind, aufgegriffen. Die Folgen einer Einschränkung des Hörens werden ebenfalls behandelt, um die Gefahren und Auswirkungen im Alltag einer hörgeschädigten Person besser verstehen zu können. Wie nahezu alle Organe des Menschen, ist auch das Ohr sehr komplex. Die Wahrnehmung von Geräuschen setzt einen komplexen Prozess der Signalverarbeitung zwischen Gehirn und Gehörorgan voraus. Die Technik des Erkennens von Richtungen wird genauer untersucht. Man unterscheidet zwischen verschiedenen Arten des Hörens, wobei jede Art hat einen unterschiedlichen Nutzen für uns hat. Zum einen versteht man unter „Binauralem Hören“ das Hören mit beiden Ohren, während man unter „Monauralem Hören“ das Hören mit einem Ohr meint. [4] In den nächsten beiden Teilabschnitten werden beide Arten des Hörens genauer beschrieben und erklärt, wofür der Mensch diese nutzt und benötigt.

2.1 BINAURALES HÖREN

Binaurales Hören beschreibt die Fähigkeit des Menschen, mit seinen beiden Ohren Schall aufzunehmen und auszuwerten. Dass der Mensch zwei Ohren hat, ist hierbei ausschlaggebend. Trifft ein Schallsignal seitlich auf den Kopf eines Menschen, erreicht der Schall die beiden Ohren zu unterschiedlichen Zeiten und Lautstärken. Abbildung 6.1 veranschaulicht dieses Phänomen. Der Schall trifft von links auf den Kopf und wird zuerst von dem linken Ohr aufgenommen. Nach einer kurzen Zeitverzögerung trifft er auch an dem rechten Ohr ein, jedoch mit einer geringeren Lautstärke. Der Vergleich dieser unterschiedlichen Werte des Zeitpunktes und des Pegels ermöglicht es, den Schall räumlich zu orten. [14]

Die beiden Unterschiede des Schalls an dem Ohr werden Interaurale Zeitdifferenz (ITD) und Interaurale Pegeldifferenz (ILD) genannt. Die folgenden Teilabschnitte werden diese kurz genauer erläutern.

2.1.1 *Interaurale Zeitdifferenz*

Wie bereits erwähnt, beschreibt die ITD den zeitlichen Unterschied, mit dem ein Schallsignal von beiden Ohren eines Menschen empfangen wird. Die zeitliche Differenz kann hier je nach Winkel der Schallwelle bis zu 680-800 μs betragen, wobei die größte Steigung bei 11μs/° liegt. Die Zeitdifferenz

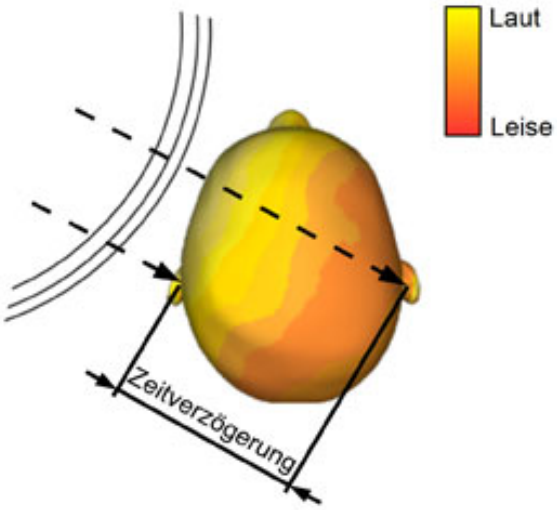

Abbildung 2.1: Schallwelle trifft auf den Kopf [12]

in Abhängigkeit des Winkels lässt sich durch eine sinusförmige Kurve beschreiben. Hierbei kann es aber dazu kommen, dass bei höheren Frequenzen die Amplitude einer Schallwelle zwischen den beiden Ohren mehr als eine Periode der Welle beträgt. Dadurch ist der Winkel nicht mehr eindeutig zuzuordnen und liefert keine Informationen bezüglich der Richtungsbestimmung. Die Richtungsbestimmung mit der ITD ist also nur bei Frequenzen bis ca. 1,3 kHz möglich. [1, 14]

2.1.2 *Interaurale Pegeldifferenz*

Die Interaurale Pegeldifferenz beschreibt also den unterschiedlichen Pegel einer Schallwelle, der von den beiden Ohren empfangen wird. Die maximale Pegeldifferenz kann bis zu 30dB erreichen. Ähnlich wie bei der ITD, spielt die Frequenz hierbei ebenfalls eine Rolle. Unterhalb von ca. 800 Hz liefert ILD nur sehr geringe Informationen, da die Pegeldifferenz zu klein ist. Bei einer Frequenz über ca. 3 kHz wächst die Pegeldifferenz monoton in Abhängigkeit vom Winkel und eignet sich daher gut für die Richtungsbestimmung bei hohen Frequenzen. Auch das Außenohr beeinflusst die Pegeldifferenz. [14]

2.2 DUPLEX-THEORIE DES RICHTUNGSHÖRENS

Wie oben beschrieben, sind die interauralen Differenzen in gewissen Frequenzbereichen nicht mehr fähig, genügend Informationen zur Richtungsbestimmung zu liefern. Die Duplex-Theorie besagt, dass das Gehirn aufgrund dieser Einschränkungen bei verschiedenen Frequenzen die Informationen eher von der Pegel- beziehungsweise der Zeitdifferenz entnimmt. Daher wird die interaurale Zeitdifferenz bei Frequenzen unter 1 kHz als primäre Informationsquelle genutzt, während bei höheren Frequenzen die interaurale Pegeldifferenz als Informationsquelle genutzt wird. Unterschiedliche Frequenzen setzen also ein Zusammenspiel aus den beiden Differenzen voraus, damit die Richtung einer Schallquelle bestimmt werden kann.

2.3 MONAURALES HÖREN

Im vorherigen Abschnitt wurde beschrieben, wie der Mensch mit beiden Ohren die Richtung von Schallwellen bestimmen kann. Es gibt noch eine andere Art um Schallwellen zu lokalisieren. Monaurales Hören nutzt nur ein Ohr. Um eine Richtung mit nur einem Ohr zu erkennen, ist das äußere Ohr entscheidend. Eintreffende Schallwellen werden, basierend auf ihrer Richtung, durch die Ohrmuschel so beeinflusst, dass das Gehirn in der Lage ist, daraus Informationen zur Lokalisierung zu entnehmen. Dabei beeinflusst nicht nur das Ohr, sondern auch der Kopf und der Körper eines Menschen die Schallwellen. Diese Eigenschaft nennt sich Head-Related-Transfer-Function (HRTF). Anders als beim binauralem Hören, dient das monaurale Hören als primäre Quelle, um abzuschätzen, ob die Schallwellen von vorne, hinten, oben oder unten kommen. [13]
Durch das Zusammenspiel von binauralen und monauralen Hören können Menschen also die Richtung von Schallwellen erkennen und sich somit basierend auf Geräuschen orientieren. Der nächste Abschnitt wird sich damit befassen, wofür diese Fähigkeit wichtig ist und welche Folgen es für einen Menschen haben kann, dem diese Fähigkeit ganz oder teilweise fehlt.

2.4 FOLGEN VON EINGESCHRÄNKTEM RICHTUNGSHÖREN

Zuvor wurde beschrieben, wie ein Mensch die Richtung von Schallwellen erkennen kann. Nun wird beschrieben, wofür diese Fähigkeit genutzt wird und welche Einschränkungen ein Mensch durch den Verlust dieser Fähigkeit erfährt. Probleme beim Richtungshören treten nicht nur dann auf, wenn eine Person auf einem Ohr taub oder teilweise taub ist, sondern auch oft bei der Verwendung eines Cochlea-Implantats. [14]

Das Gehör lokalisiert, klassifiziert Geräusche und kann diese auch teilweise ausblenden. So kann man sich in einem Gespräch auf den Gesprächspartner konzentrieren und die Umgebungsgeräusche ausblenden. Diese Fähigkeit basiert wiederum auf dem Richtungshören. Einer Person mit eingeschränktem Richtungshören fehlt diese Fähigkeit und es kommt zur so-

genannten "Party deafness". Sie kann bei Gesprächen die Umgebungsgeräusche nicht oder nur teilweise ausblenden und hat Schwierigkeiten, in einer lauten Umgebung, den Gesprächspartner richtig zu verstehen. [18] Eine weitere Einschränkung ist, dass Gefahrensituationen nicht richtig erkannt oder wahrgenommen werden können, was vor allem im Straßenverkehr eine wichtige Rolle spielen kann. Menschen mit eingeschränktem Richtungshören haben demnach Probleme hinsichtliche der Orientierung oder der Einschätzung von Gefahren. Der nächste Abschnitt wird sich mit der bisherigen Arbeit an einer visuellen Hörhilfe widmen, um dem Leser nun einen Lösungsansatz zur Behandlung, beziehungsweise zur Hilfe bei eingeschränktem Richtungshören zu erläutern.

3

BISHERIGE ARBEIT

Dieser Abschnitt wird einen Einblick in die bisherige Arbeit von Herrn Udo Gebelein und Prof. Dr. Stefan Rapp geben. Herr Udo Gebelein leidet selbst an eingeschränktem Hörvermögen und kam auf die grundlegende Idee, ein Gerät zu entwickeln, das Menschen mit eingeschränktem Hörvermögen helfen kann. Ziel war, dass das Gerät das Hörvermögen im Bezug auf die Wahrnehmung von Geräuschquellen verbessert, beziehungsweise wiederherstellt. Das Gerät soll über eine ausreichende Zahl an Mikrophonen verfügen, um möglichst alle Richtungen abzudecken. Durch diese Mikrophone werden dann sämtliche Umgebungsgeräusche aufgenommen und mit verschiedenen Algorithmen ausgewertet. Dadurch soll das Gerät die Richtung, aus der die Geräusche kommen, erkennen und an den Nutzer weiterleiten. Normale Hörgeräte geben dem Nutzer meistens ein akustisches Feedback, aber die Idee von Herr Udo Gebelein hat ein visuelles Feedback an den Nutzer vorgesehen. Daher kommt auch der Name „Visual Hearing Aid“. Folgende Abbildung zeigt den ersten entwickelten Prototyp.

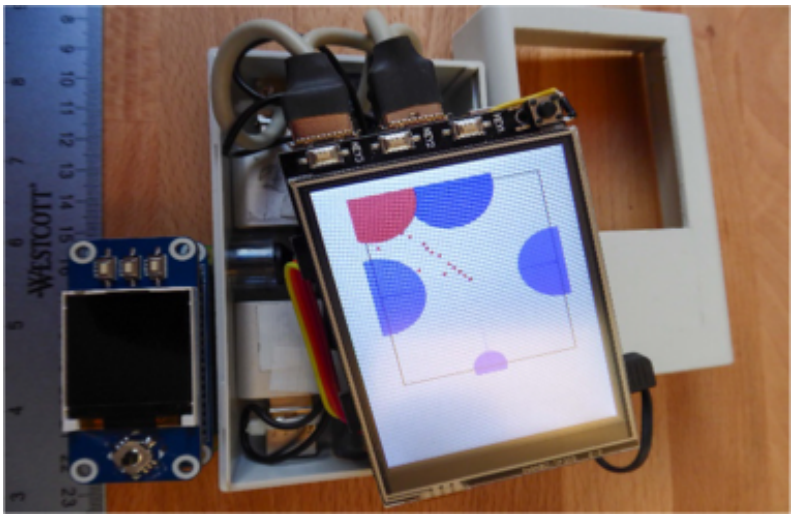

Abbildung 3.1: VisualHearingAid Prototyp mit einem kleinen Monitor [3]

Der erste Prototyp basiert auf 4 einzelnen USB-Mikrofonen, die an einem Raspberry Pi angeschlossen sind und einem Bildschirm. Die 4 blauen Punkte auf jeder Seite repräsentieren jeweils ein Mikrophon. Die Größe der Punkte wird dabei durch die Größe des Schallpegels bestimmt. Der rote Punkt in der linken Ecke steht für ein Geräusch, das seinen Ursprung in diesem Fall vorne links hat. Die kleineren Punkte stehen für vergangene Geräusche und wandern mit der Zeit in die Mitte. Dabei werden die Schallquellen durch bekannte Algorithmen lokalisiert (Siehe 2.1.2 und 2.1.1). Dieser Prototyp

wurde allerdings weiterentwickelt, da die USB-Mikrofone nicht zuverlässig genug sind, um eine vernünftige Lokalisierung zu realisieren. [3]

Der zweite Prototyp basiert wieder auf einem Raspberry Pi. Anstatt der USB-Mikrofone wird ein Entwicklerboard mit mehreren Mikrophonen genutzt. Der nächste Abschnitt stellt die dort genutzte Technologie vor und geht genauer auf die technische Realisierung ein.

Herr Udo Gebelein und Herr Prof. Dr. Stefan Rapp haben ihre Arbeit im Rahmen der Konferenz Elektronische Sprachsignalverarbeitung 2020 (ESSV) in Magdeburg vorgestellt.

4 BISHERIGER PROTOTYP

4.1 TECHNOLOGIE

Dieser Abschnitt bietet eine Übersicht über die verwendeten Technologien, die der Prototyp nutzt. Zuerst werde ich die verwendete Hardware vorstellen und anschließend die verwendete Software und den Systemaufbau des Prototyps.

4.1.1 *Hardware*

Die grundlegende Hardware besteht aus einem Raspberry Pi in Kombination mit dem Matrix Creator, der an den GPIO Port des Raspberry Pis angeschlossen ist. [8, 11] Der Matrix Creator ist ein Entwicklerboard für IoT Projekte, das neben einigen Sensoren für das Erfassen der Umgebung ebenfalls ein SMD Mikrofon-Array mit 8 Mikrofonen besitzt. Mit Hilfe des Mikrofon Arrays können die Umgebungsgeräusche aufgenommen werden. Zusätzlich wird noch ein Gerät zur Visualisierung genutzt. Das übernimmt bei dem bisherigen Prototyp der Raspberry Pi selbst, indem er an einen externen Monitor angeschlossen ist. Im Vergleich zum ersten Prototyp ist die Qualität der Mikrofone besser und der Matrix Creator bringt viele Sensoren mit sich, die in Zukunft hilfreich sein könnten. Die Hardware ist damit relativ überschaubar, kostengünstig und leicht zu bedienen.

4.1.2 *Software*

Dieser Abschnitt stellt die verwendetet Software vor, die auf dem Raspberry Pi und dem Gerät zur Visualisierung läuft. Die Software besteht aus zwei verschiedenen Programmen, die über einen lokalen Socket auf dem Raspberry Pi kommunizieren. Die von dem Matrix Creator aufgenommenen Umgebungsgeräusche werden mit der open source Software Open embeddeD Audition System (ODAS) [5] ausgewertet. ODAS ist eine kostenlose Bibliothek, die der Lokalisierung, Verfolgung und Trennung von Schallwellen dient. ODAS ist in C programmiert und für den Einsatz im embedded Bereich angepasst. ODAS setzt hierbei auf bekannte Algorithmen und verbessert diese, um den Rechenaufwand zu minimieren, was für den embedded Einsatz wichtig ist. Dafür wird das vorhandene Verfahren Steered Response Power with Phase Transform (SRP-PHAT) mit Hierarchical Searchwith Directivity model and Automatic calibration (HSDA) kombiniert. Das resultierende Verfahren wird SRP-PHAT-HSDA genannt und verbessert das grundlegende Verfahren, indem der Raum zunächst mit einem groben Raster gefolgt von einem feinen Raster abgetastet wird, um Speicherzugriffe zu reduzieren.

Dazu wird die Anzahl der abzutastenden Mikrofone reduziert, indem ein Richtwirkungsmodell genutzt wird, um nicht signifikante Mikrofonpaare zu ignorieren. Den Entwicklern zufolge erreicht das Verfahren nahezu gleiche Ergebnisse im Vergleich mit anderen Verfahren zur Lokalisierung von Geräuschen, benötigt aber 4 bis 30 mal so wenig Rechenleistung. ODAS arbeitet dabei mit ähnlichen Methoden wie das menschliche Gehör. Es nutzt ebenfalls die Zeitdifferenz des Schalls bei allen Mikrofonen und ermittelt so die Richtung aus der der Schall kommt. [5] Hauptaufgaben sind hierbei die SSL und die SST. Darunter versteht man das Lokalisieren von Geräuschen und daraufhin das Verfolgen des lokalisierten Geräusches. Damit ist es möglich, unerwünschte Geräusche auszublenden oder sich nur auf eine bestimmte Art von Geräuschen zu fokussieren.

Die damit klassifizierten Geräusche werden dann über einen lokalen Socket an das Gerät zur Visualisierung im JSON Format gesendet. (siehe 4.1)

```
1
2  {
3    "timeStamp": 2,
4    "src": [
5        { "x": -0.081, "y": -0.112, "z": 0.990, "E": 0.144 },
6        { "x": 0.066, "y": 0.021, "z": 0.998, "E": 0.252 },
7        { "x": -0.663, "y": 0.285, "z": 0.692, "E": 0.095 },
8        { "x": -0.582, "y": 0.801, "z": 0.141, "E": 0.042 }
9    ]
10 }
```

Listing 4.1: Beispiel Daten im JSON Format.

Die Daten bestehen aus einem timeStamp und einem src-Objekt. Das src-Objekt besteht dabei aus mehreren Tuples mit jeweils X, Y, Z und E Werten. Dabei ergeben sich aus den ersten drei Werten die Koordinaten im Raum. Der E-Wert steht in diesem Fall für Energie und bestimmt die Größe beziehungsweise die Lautstärke des eigentlichen Geräusches.
Auf dem verwendeten Gerät zur Visualisierung läuft die Software, die von Herrn Udo Gebelein und Prof. Dr. Stefan Rapp entwickelt wurde und empfängt die JSON Strings. Gleichzeitig wird eine Szene in OpenGl gerendert, die mit einem Overlay versehen ist, auf dem man ein Zifferblatt einer analogen oder digitalen Uhr anzeigen kann. Das Uhr-Overlay ist dafür gedacht, dass das Programm zum Beispiel auf einer Smartwatch dauerhaft laufen kann und die Funktionalität der Uhr nicht verloren geht. Anhand der Daten aus dem JSON-String wird ein Object einer Klasse `SoundEvent` erstellt (siehe 5.2). Die Soundevents werden in einem Vector gespeichert und mit einer Kugel (siehe 5.4) in der Szene repräsentiert. Die verschiedenen Eigenschaften eines `Soundevents` bestimmen dann Größe und Position auf dem Bildschirm. Alle empfangenen `Soundevents` werden als Kugel auf dem Bildschirm gerendert. Dabei wandern sie mit zunehmender Zeit vom äußeren Rand in Richtung Mitte des Bildschirms. Dadurch entsteht eine Art Zeitstrahl. Die folgende Abbildungen zeigen ein Beispiel einer gerenderten Szene mit dem analogen

Uhr-Overlay und dem Systemaufbau des Prototypen von Herrn Udo Gebelein und Prof. Dr. Stefan Rapp.

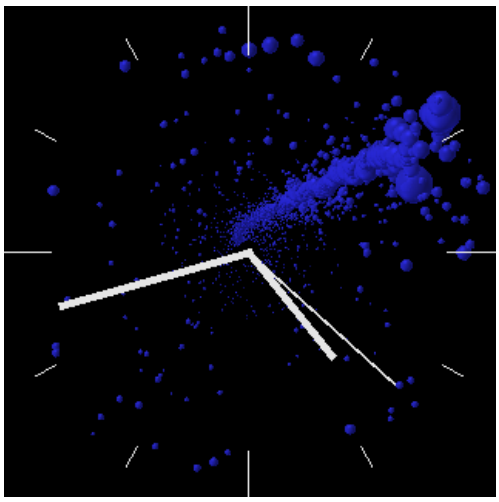

Abbildung 4.1: Visualisierung mit einem Uhr Overlay [3]

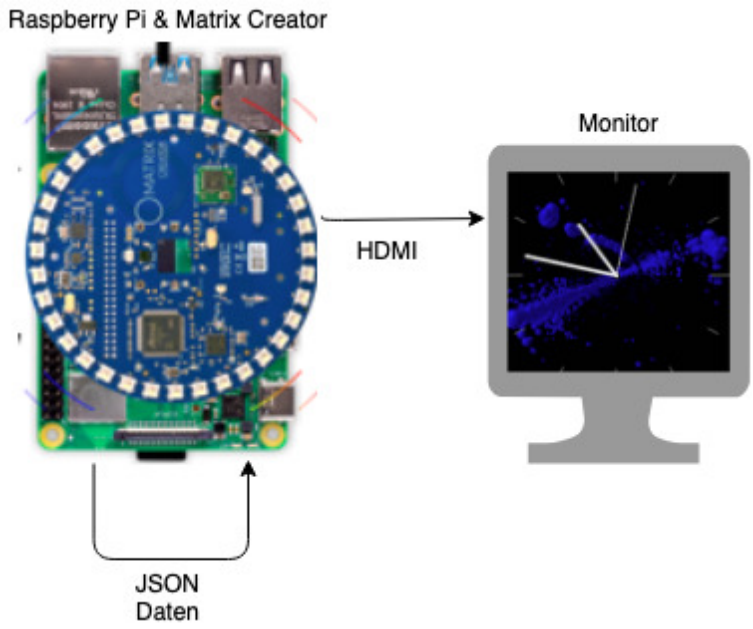

Abbildung 4.2: Systemaufbau als Zeichnung

5 WEITERENTWICKLUNG DES PROTOTYPS

In diesem Abschnitt wird die Entwicklung des Prototyps dokumentiert. Der bisherige Prototyp ist im Bezug auf die Visualisierung an den Raspberry Pi gebunden. Die Visualisierung wird auf dem Raspberry Pi gerendert und über einen angeschlossenen Bildschirm angezeigt (siehe 4.2). Mit dem vorhandenen Prototyp ist es schwierig, das System in der echten Welt zu testen, da man an einen festen Monitor gebunden ist. Ziele des neuen Prototyps sind Mobilität, Diskretion und Koordination zu erreichen.

- Mobilität: Das System soll tragbar und mobil sein.
- Diskretion: Es soll möglichst unauffällig sein, das heißt es darf nicht von außen direkt erkennbar sein, dass jemand das System gerade nutzt oder bei sich trägt.
- Koordination: Das System soll sich ausgehend von einem gewählten Koordinatensystem orientieren und die Visualisierung dementsprechend anpassen, damit der Benutzer jederzeit brauchbare Informationen erhält.

Diese Anforderungen sind durch folgende Maßnahmen gedeckt: Die Mobilität und Diskretion wird erreicht, indem der Raspberry Pi mit der kompletten Peripherie in einer Kappe oder einem Hut versteckt wird. Dabei soll sichergestellt werden, dass die Mikrophone nicht zu stark abgeschirmt werden. Da der Raspberry Pi eine Stromquelle benötigt, wird noch eine Powerbank benötigt. Dazu wird ein mobiles Gerät zur Visualisierung genutzt, wodurch ein Proband das System mit Kappe und Smartphone oder Smartwatch nutzen kann, ohne dass Außenstehende erkennen können, dass es sich dabei um eine Hörhilfe handelt. Eine Smartwatch zur Visualisierung ist hinsichtlich Diskretion und Nutzbarkeit die bessere Variante. Die Koordination der Visualisierung wird durch das Auslesen der Bewegungssensoren des Smartphones beziehungsweise der Smartwatch und dem Matrix Creator errechnet.

Die Entwicklung des Prototypen lässt sich in zwei Hauptaufgaben unterteilen. Einerseits muss die Hardware eingerichtet und in einer Kappe oder einem Hut untergebracht werden. Dabei ist es wichtig, dass der Raspberry Pi mit dem Matrix Creator am besten waagerecht und fest in der Kappe oder dem Hut sitzt. Da neben der Hardware noch eine Stromquelle verbaut wird, sollte die gewählte Kopfbedeckung ausreichend Platz über dem Kopf haben.

Die andere Aufgabe besteht darin, die schon vorhandene Software in eine App zu integrieren. Der ursprüngliche Code ist in C++ geschrieben und nutzt OpenGL zur Visualisierung. Als Entwicklungsumgebung wird aus verschiedenen Gründen Qt Creator benutzt, der im nächsten Abschnitt vorgestellt wird.

5.1 QT CREATOR

Dieser Abschnitt wird das Qt Framework vorstellen, wobei alle Informationen der offiziellen Dokumentation von Qt selbst entnommen sind. Qt Creator ist ein umfangreiches Framework, um Software zu entwickeln. Es dient als Entwicklungsumgebung und bietet eine Vielzahl an Funktionen, die es unter anderem ermöglichen, Applikationen für verschiedene Zielsysteme per Cross-Compiling zu entwickeln und dabei die gleiche Codebasis zu nutzen. Dies bietet viele Vorteile, wenn man mobile Apps entwickelt und mehrere Plattformen ansprechen will. Zudem bietet Qt Creator eine eigene Modeling-Language, die im nächsten Teilabschnitt 5.1.2 vorgestellt wird. Es gibt ebenfalls die Möglichkeit, verschiedene Programmiersprachen zu verwenden. So wird zum Beispiel eine Programmierschnittstelle über eine Klasse bereitgestellt, die es ermöglicht, eine Java Funktion in einem C++ Projekt aufzurufen. [10]

Für die Entwicklung der App sind vor allem folgende Punkte des Qt Framework hilfreich:

- Qt ermöglicht App-Programmierung in C++, was den Vorteil bietet, dass man den vorhandenen Code wiederverwenden kann.
- Das Cross-Compiling Feature ermöglicht es, Android und iOS Geräte für die Applikation zu nutzen.
- Es bietet viele Möglichkeiten OpenGL zu verwenden und stellt einige Funktionen zur Verfügung, die es erleichtern, existierenden OpenGL-Code wiederzuverwenden.
- Mit der eigenen Modeling-Language kann schnell und einfach eine Benutzeroberfläche erstellt werden.

Die Visualisierungen werden mittels OpenGL beziehungsweise OpenGL ES erstellt, daher ist es ebenfalls von Vorteil, dass man in Qt existierenden OpenGL-Code wiederverwenden kann. Um OpenGL zu nutzen, gibt es bei Qt viele verschieden Möglichkeiten, die teilweise aber schon veraltet sind oder nicht mehr genutzt werden sollten. Die Entscheidung, welcher Ansatz in diesem Fall am sinnvollsten ist, wird in den nächsten Abschnitten genauer diskutiert und es wird ein kurzer Einblick in verschiedene Möglichkeiten gegeben. Dabei wird kurz auf die Vor- und Nachteile eingegangen und erläutert, welcher Ansatz für verschiedene Zwecke geeignet ist.

In den folgenden Teilabschnitten wird das Qt Framework genauer vorgestellt und auf einige Mechanismen eingegangen, die man als Entwickler kennen muss, um mit diesem Framework arbeiten zu können.

5.1.1 *Signale und Slots*

Eine der größten Besonderheit von dem Qt Framework ist das Signal- und Slot-Prinzip. In anderen Frameworks wird für die Kommunikation zwischen

Objekten ein Callback-Mechanismus verwendet. Dabei wird einer Funktion eine Callback Funktion mitgegeben, die zu einer bestimmten Zeit ausgeführt wird um gewisse Ereignisse zu signalisieren. Qt nutzt hierfür das Signal- und Slot-Prinzip. Ein Signal wird ausgesendet, wenn ein bestimmtes Ereignis stattfindet. Mit diesem Signal kann man nun beliebig viele Slots verbinden, die dann durch das Signal ausgeführt werden. Viele Qt-Objekte haben vordefinierte Signale, man kann aber auch als Entwickler eigene Klassen mit Signalen und Slots ausstatten. Signale und Slots sind in Qt typensicher, können beliebig viele Argumente von jedem Typ besitzen und sind lose gekoppelt. Das heißt, dass eine Klasse, die ein Signal aussendet, nicht weiß, welche Slots dieses Signal empfangen. Damit eine Klasse diesen Mechanismus nutzen kann, muss sie von `QObject` oder eine Unterklasse von `QObject` erben. Signale sind öffentliche Funktionen und können auch außerhalb einer Klasse aufgerufen werden. Sie werden mit dem `emit` Statement gesendet und die passenden Slots werden direkt ausgeführt, bevor der eigentliche Programmablauf weiter ausgeführt wird. Es gibt allerdings auch `queued connections`, die Slots asynchron aufrufen. Signale müssen nur deklariert und nicht implementiert werden, da sie von Qt automatisch generiert werden. Slots hingegen sind normale C++ Funktionen und können auch, wie jede andere C++ Funktion aufgerufen werden. Die Besonderheit besteht darin, dass sie mit Signalen verbunden und von jeder Klasse aufgerufen werden können, auch wenn die Klasse eigentlich keinen Zugriff hat. Hinsichtlich der Performance sind Signale und Slots etwa 10 mal langsamer als der Callback-Mechanismus mit nicht virtuellen Funktionen. Allerdings sind die Performanceeinbußen durch die einfache Benutzung und die vielen Möglichkeiten durchaus hinzunehmen. [10]
Mit dem Signal- und Slot-Prinzip stellt Qt ein einfaches und mächtiges Werkzeug zur Kommunikation zwischen Objekten, auch zwischen Objekten aus der GUI und den C++ Klassen im Hintergrund. Für die Entwicklung der visuellen Hörhilfe ist dieses Prinzip sehr hilfreich. Der nächste Teilabschnitt wird nun die eigene Modeling-Language on Qt vorstellen, bevor genauer auf die Verwendung von OpenGL in Qt eingegangen wird.

5.1.2 *QML und Szenengraph*

Dieser Abschnitt schafft einen Überblick über die Modeling-Language von Qt namens Qt Modeling Language (QML). Dabei wird die Sprache selbst vorgestellt, aber es wird auch auf den Szenengraph, der in Qt für das Rendern in QML genutzt wird, eingegangen. Wenn man mit dem Qt Creator arbeiten möchte, wird man sich zwangsläufig mit dem Begriff des Szenengraphs beschäftigen müssen. QML ist eine deklarative Programmiersprache für Qt Applikation. Sie erlaubt es, Benutzeroberflächen zu erstellen und bietet gleichzeitig Möglichkeiten, das Verhalten und die Interaktionen zwischen verschiedenen Objekten der Benutzeroberflächen zu steuern. Die Syntax von QML ist im JSON Format definiert, aber JSON an sich wird nicht unterstützt. Dadurch ist die Sprache sehr gut lesbar und leicht zu lernen. QML ist so auf-

gebaut, dass jedes Element mit anderen Elementen interagieren und über eine Verbindung verfügen kann. Die Elemente werden deklarativ erstellt. Sie besitzen optionale Eigenschaften, die man frei setzen kann. Eine wichtige Komponente hierbei ist JavaScript, was in QML komplett unterstützt wird. Mit JavaScript und der Möglichkeit, Eigenschaften eines Elements dynamisch zu ändern, kann man eine komplette Applikationen in QML entwickeln, ohne ein Backend zu haben.[10]

Das Modul `QtQuick` (Qt User Interface Creation Kit) erlaubt es den Entwicklern ohne großen Aufwand verschiedene Animationen, Effekte oder Ansichten zu nutzen. Um Benutzereingaben, Menüs oder andere Steuerelemente zu erstellen, wird das Modul `QtQuickControlls` bereitgestellt, was bereits über eine Vielzahl an vorgefertigten Elementen verfügt, bei denen man das Aussehen und Verhalten ändern kann. Module werden einfach mit einem Import Statement eingebunden und können dann genutzt werden. Der Codeausschnitt im Beispiel 5.1 zeigt ein einfaches grünes Rechteck und einen Button. Bei einem Klick auf den Button ändert sich die Farbe des Rechtecks.

```
1  import QtQuick 2.14
2  import QtQuick.Controls 2.0
3  (...)
4    Rectangle {
5      id: rectangle1
6      color: "red"
7      width: 100
8      height: 100
9  }
10 Button {
11     id: button1
12     onClicked: rectangle1.color = "green"
13 }
```

Listing 5.1: Einfaches QML Beispiel

Damit eine Benutzeroberfläche in QML gerendert werden kann, wird ein Szenengraph verwendet. Der Begriff des Szenengraph ist wie folgt definiert: Ein Szenengraph beschreibt eine Datenstruktur, die die logische und räumliche Anordnung von Elementen in einem zwei- oder dreidimensionalen Raum beschreibt.[15] Jedes Element in QML wird in einem Szenengraph repräsentiert. Allerdings besteht dieser aus verschiedenen Nodes, die wiederum zusammen einen Node-Tree bilden. Daher ist der Begriff des Szenengraphs in diesem Fall eher ein Pseudonym für eben diesen Node-Tree. Eine Node wird dabei aus verschiedenen Datentypen erstellt, wobei jeder davon verschiedene Aufgaben hat. Mehrere Nodes ergeben den Szenengraph, der dann von einem Renderer verarbeitet und gerendert wird. Die Nodes selber besitzen allerdings keinen Render-Code. Eine Node wird durch die Klasse `QSGGeometryNode` repräsentiert. Diese Klasse beschreibt eine Node, indem sie Informationen über die Geometrie und das Material der Node beziehungsweise des `QQuckItem` speichert. Unter Geometrie versteht man die Form des Elements. Das kann eine Linie, ein Dreieck oder eine komplizierte 3D Struk-

tur sein. Diese Geometrie wird durch die Klasse `QSGGeometry` repräsentiert. Das Material kann durch verschiedene Klassen definiert werden. Bei einem einfachen, einfarbigen Material, kann man die Klasse `QSGFlatColorMaterial` verwenden. Für ein Shader-Material wird die Klasse `QSGMaterialShader` bereitgestellt und für ein Material, was auf einer Textur basiert, gibt es die Klasse `QSGTextureMaterial`. Es gibt noch weitere Klassen, die ein Material genauer spezifizieren. Die Abbildung 5.1 veranschaulicht die Klassenhierarchie der Node-Klassen.

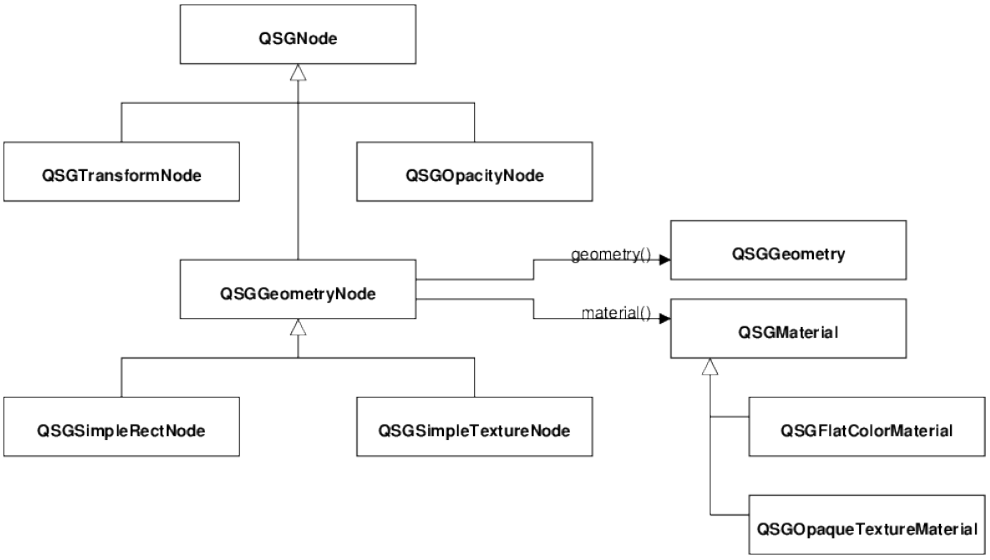

Abbildung 5.1: Szenengraph Klassendiagramm [2]

Als Entwickler besteht die Möglichkeit, dass man auch eigene Datentypen erstellen kann, aus denen dann eine Node entsteht. Dafür muss die eigenen Klasse von einer bestimmten Klasse erben. Dieses Vorgehen wird genauer im Abschnitt 5.1.3.1 beschrieben. In der eigenen Klasse muss man dann die Funktion `QQuickItem::updatePaintNode()` reimplementieren. Dort kann man seine eigene Geometrie und das Material erstellen. Wichtig ist aber, dass man nur die dafür vorgesehenen Klassen mit dem QSG Prefix verwendet. Abbildung 5.2 zeigt den internen Aufbau und Ablauf beim Rendern des Szenengraphs in QML.

Damit ist der interne Ablauf des Renderns und die Szenengraph API von QML für diese Zwecke ausreichend erklärt. Allerdings gibt es Methoden zum Rendern von OpenGL-Code, die den Szenengraph automatisch erstellt. Dies wird ebenfalls genauer im Abschnitt 5.1.3.1 vorgestellt.

5.1.3 *OpenGL in Qt*

Dieser Abschnitt beschäftigt sich mit der Verwendung von OpenGL in einem Qt Projekt. Zunächst wird einen kurzen Einblick in die Qt Welt im Bezug auf OpenGL geben und erklärt, wie man existierenden OpenGL-Code in einem neuen Projekt verwenden kann, ohne alles neu zu schreiben. Da

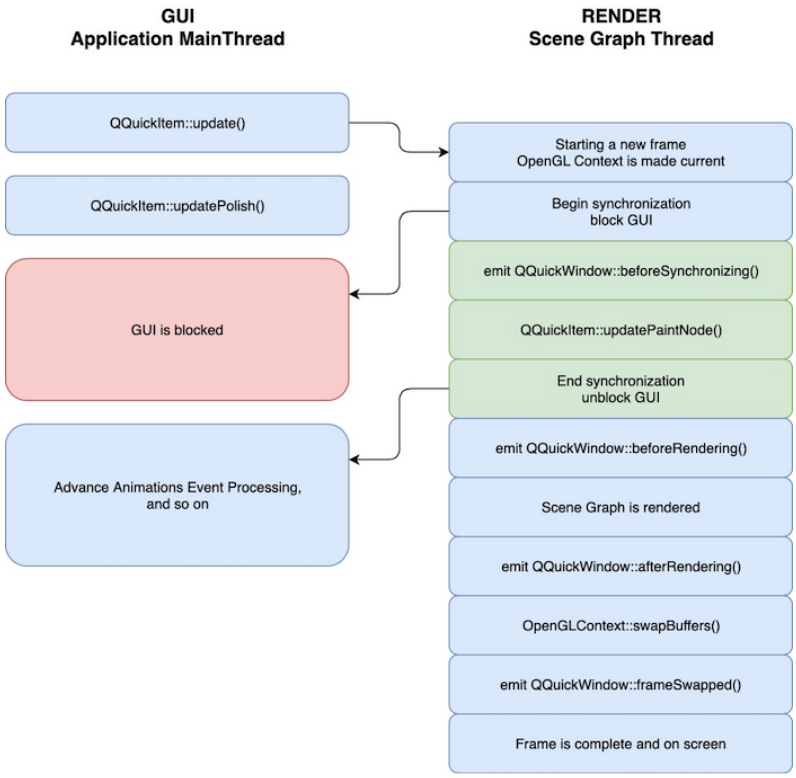

Abbildung 5.2: Szenengraph in QML mit Flow-Chart [10]

es in Qt oft viele verschiedenen Möglichkeiten und Module für den gleichen Zweck gibt, werden drei verschiedene konkrete Ansätze für OpenGL in Qt vorstellen, bevor die Entwicklung der App mit einem dieser Ansätze genauer dokumentiert wird.

5.1.3.1 *OpenGL API*

Ziel hierbei ist es nicht OpenGL zu erklären, sondern viel mehr auf die abweichende Benutzung von OpenGL in Qt einzugehen. Um in Qt eine einfache Entwicklung zu garantieren, wird eine API für OpenGL bereitgestellt. Sie erleichtert die Benutzung, aber kümmert sich auch darum, dass die jeweils richtige OpenGL Version für das Zielsystem benutzt wird. Die wichtigsten Klassen hierfür sind `QOpenGLFunctions` und `QOpenGLShaderProgram`.
Die Klasse `QOpenGLFunctions` ist für die normalen OpenGL Funktionen zuständig. Sie dient als Convenience-Wrapper indem sie alle OpenGL Funktionen abkapselt. Mit der Klasse `QOpenGLShaderProgram` lässt sich mit geringem Aufwand ein Shader-Programm erstellen.

Um die `QOpenGLFunctions` Klasse als Entwickler zu nutzen, muss man eine eigenen Klasse erstellen. Diese Klasse muss von `QOpenGLFunctions` erben. Man kann allerdings auch ein eigenes Objekt der `QOpenGLFunctions` Klasse im Code erstellen, ohne dass die eigene Klasse von ihr erbt. In beiden Fällen müssen die Funktionen initialisiert werden, da-

mit die Funktionen an den aktuellen OpenGL Kontext gebunden sind. Wenn man von `QOpenGLFunctions` erbt, reicht es, die Funktion `QOpenGLFunctions::initializeOpenGLFunctions()` aufzurufen. Danach kann die OpenGL Funktionen wie gewohnt aufrufen. Falls man ein eigenes Objekt der `QOpenGLFunctions` Klasse benutzt, muss man die `initializeOpenGLFunctions()` Funktion über dieses Objekt aufrufen und alle folgenden OpenGL Funktionen ebenfalls darüber aufrufen. Qt empfiehlt jedoch von der Klasse zu erben. Neben dem Initialisieren der OpenGL Funktionen muss man die Herangehensweisen bei der Shaderprogrammierung beachten. Die dafür vorgesehene Klasse `QOpenGLShaderProgram` kapselt das Kompilieren und Linken von Vertex- und Fragmentshadern ab. Dadurch muss man sich als Entwickler um weniger Details kümmern. Mit dieser Klasse ist es möglich mit zwei Funktionsaufrufen einen Shader zu kompilieren und zu linken. Die beiden Funktionen schaffen also eine Schnittstelle, die dem Entwickler eine einfache Einbindung von OpenGL-Code ermöglicht und stellen sicher, dass die richtige OpenGL Version genutzt wird, auch beim Cross-Compiling. [10]

Im folgenden Teil werden die drei verschiedenen Ansätze genauer beschrieben. Jeder dieser Ansätze basiert darauf, dass ein Fenster erstellt wird, in dem mit OpenGL eine Szene gerendert werden kann. Allerdings unterscheiden sie sich in der Herangehensweisen, den verwendeten Modulen und sind entweder für den Desktop oder mobilen Einsatz geeignet. Die Benutzung der verschiedenen Ansätze wird im Bezug auf das Wiederverwenden von existierendem OpenGL-Code erklärt und es wird auf alle notwendigen Schritte eingegangen.

1. QOpenGLWindow
   Der erste Ansatz baut auf einer desktop-orientierten Variante ohne QML auf. Dafür wird die Klasse `QOpenGLWindow` genutzt. Diese Klasse ist eine Weiterentwicklung der Klasse `QWindow`, die für das Erstellen und Verwalten von Fenstern in Qt zuständig ist aber QML nutzt. Die Klasse bietet allerdings die Möglichkeit, OpenGL direkt zu verwenden.

   Um die Klasse zu verwenden, muss man lediglich eine eigene Klasse erstellen und von ihr erben. Folgende drei virtuelle Funktionen müssen in der eigenen Klasse reimplementiert werden:

   - `initializeGL()`
   - `resizeGL()`
   - `paintGL()`

   Diese Funktionen sind OpenGL typisch und tauchen auch im ursprünglichen Code auf (siehe 5.2.1). Um nun den existierenden OpenGL-Code in einer von `QOpenGLWindow` abgeleiteten Klasse auszuführen, muss man also lediglich den Inhalt der Funktionen aus dem ursprünglichen Code einfügen. Wie zuvor beschrieben muss man allerdings darauf achten, dass man die OpenGL-API richtig verwendet (siehe 5.1.3.1).

Hat man die drei virtuellen Funktionen implementiert und alle notwendigen Änderungen an dem vorhandenen Code erledigt, kann man in der `main()` ein Objekt der eigenen Klasse erstellen und die Funktion `show()` aufrufen. Wenn man einen Loop nutzen möchte, um periodisch die Szene neu zu rendern oder eine Animation implementieren möchte, sollte man die Slot-Funktion `update()` dafür nutzen. Diesen Slot kann man mit einem Timer verbinden, indem man das Timeout-Signal eines Timers mit der `update()` Funktion verbindet. Allerdings bietet es sich an, die `update()` Funktion mit dem Signal `frameSwapped` zu verbinden, da nach jedem gerendertem Frame die Funktion `swapBuffers()` dieses Signal aussendet. So kann man Animationen steuern ohne extra einen Timer zu verwenden. Um eventuelle Benutzereingaben abzufangen, bietet die Basisklasse `QWindow` eigene Funktionen. Um zum Beispiel ein Tastendruck abzufangen, kann man die virtuelle Funktion `keyPressEvent()` reimplementieren und dort abfragen, welche Tasten gedrückt worden sind. Das gleiche geht auch mit einer Eingaben über die Maus. Dafür verwendet man die verschiedenen `mousePressEvent()`, `mouseMoveEvent()` oder `mouseReleasedEvent()` Funktionen. [10] Damit wären alle notwendigen Schritte erledigt, um mit der `QOpenGLWindow` Klasse ein lauffähiges OpenGL Programm zu entwickeln, was für desktop-basierte Anwendungen geeignet ist.

2. QQuickItem
   QQuickItem ist der zweite Ansatz, der in dieser Arbeit vorgestellt wird. Dabei wird das `QtQuick` Modul genutzt, was die Basis für alle in QML geschriebenen GUIs stellt. `QQuickItem` ist die Basisklasse für alle visuellen Elemente in `QtQuick` und ist von der Klasse `QObject` abgeleitet, welche wiederum die Basisklasse für alle Qt Objekte ist. Ein `QtQuickItem` liefert kein sichtbares Erscheinungsbild, stellt aber alle notwendigen Attribute, wie Position, Höhe und Breite. Ein großer Unterschied zu dem ersten Ansatz ist, dass ein `QQuickItem` von dem Programmteil, der für das eigentliche Rendern zuständig ist, getrennt ist. Diese Trennung erfolgt durch zwei Threads: Ein GUI-Thread und ein eigener Render-Thread. Dadurch entsteht ein Mehraufwand als Entwickler, aber es ermöglicht eine andere Herangehensweise und bietet viel mehr Möglichkeiten eine OpenGL basierte Anwendung zu entwickeln. Zum einen hat man die Möglichkeit, die GUI in QML zu entwickeln. Qt Creator bietet hierfür ein eigenes Design-Tool, mit einem „What you see is what you get“ Ansatz. Zum anderen hat man die komplette Kontrolle über den Render-Thread. Zunächst wird die Vorgehensweise mit `QQuickItem` kurz erläutert. Wie bei dem vorherigen Ansatz, erstellt man zuerst eine eigene Klasse. Diese Klasse erbt von `QQuickItem`. Nun gibt es mehrere Möglichkeiten, um das Rendern zu steuern. Normalerweise reimplementiert man die Funktion `QSGNode* updatePaintNode(QSGNode*, UpdatePaintNodeData*)`.

Diese Funktion ermöglicht es direkt mit dem Szenengraph zu interagieren (siehe 5.1.2). Wie zuvor schon beschrieben, erfolgt das Rendern in QML durch den Szenengraph. Hierbei ist es wichtig, dass in der `QSGNode* updatePaintNode(QSGNode*, UpdatePaintNodeData*)` nur Klassen mit dem Präfix "QSG"verwendet werden, da diese Klassen extra dafür vorgesehen sind. Diese Methode sieht aber keinen direkte Verwendung von OpenGL-Code vor, daher wird eine weitere Möglichkeit vorgestellt, die es erlaubt mit der `QQuickItem` Klasse normalen OpenGL-Code zu verwenden. Dazu benötigt man eine separate Render-Klasse, die für das Rendern zuständig ist. Diese Klasse wird separat von dem `QQuickItem` Objekt erstellt, da dieses nur im GUI-Thread lebt und die Render-Klasse in einem eigenen Thread ausgeführt wird. Damit beide Klassen miteinander funktionieren, müssen wir einige Signale und Slots beider Klassen verbinden.

Die benötigte Render-Klasse lässt man von `QObject` und `QOpenGLFunctions` (siehe 5.1.3.1) erben, da hier der OpenGL-Code verwendet werden soll. Dazu werden zwei Slots erstellt, die mit passenden Signalen verbunden werden müssen:

- `init()`
- `paint()`

In diesen Slots findet dann der OpenGL-Code wie gewohnt seinen Platz. In der `init()` Slot-Funktion wird OpenGL initialisiert und in der `paint()` Slot-Funktion findet dann das eigenen Rendern statt. Um nun beide Klassen miteinander zu verbinden, stellt `QQuickItem` folgende Slot-Funktionen:

- `sync()`
- `cleanup()`
- `handleWindowChanged(QQuickWindow *win)]`

Der `cleanup()` Slot dient hierbei als Destruktor und sorgt dafür, dass der Destruktor bei beiden Klassen aufgerufen wird. Durch den `sync()` Slot wird die Verbindung der beiden Klassen hergestellt.

In dem ursprünglichen `QQuickItem` Objekt wird der Slot `handleWindowChanged()` im Konstruktor mit dem `windowChanged` Signal verbunden. In dem Slot selbst wird dann der `sync()` und `cleanup()` Slot mit den Signalen aus dem neuen Window verbunden. Dabei wird `sync()` mit dem Signal `beforeSynchronizing` und `cleanup()` mit `sceneGraphInvalidated` verbunden. In der Implementierung von `sync()` wird eine Instanz von der Renderer-Klasse erstellt und die beiden Signale mit den passenden Slots verbunden. Der `init()` Slot wird mit dem Signal `beforeRendering` und `paint()` mit `beforeRenderPassRecording` aus dem zugehörigen Window, verbunden. Durch all diese Signale kann entschieden werden, wann die OpenGL Szene gerendert wird. In diesem Fall wird das Rendern durch das `beforeRenderPassRecording` ausgelöst und der

OpenGL-Teil wird vor der eigentlichen GUI aus dem `QQuickItem` gerendert. Dadurch liegt die GUI über der OpenGL-Szene. Es besteht auch die Möglichkeit auf das Signal `afterRenderPassRecording` zu reagieren. Dann wäre die GUI hinter, beziehungsweise unter der OpenGL Szene. Dieser Signal- und Slot-Mechanismus erlaubt es zu entscheiden, wann der OpenGL-Teil gerendert werden soll und vor allem kapselt es die GUI von dem eigentlichen Rendern ab. Das verhindert unter anderem auch Verzögerungen und Einfrieren in der GUI. Eventuelle Benutzereingaben durch Buttons oder Ähnliches werden auch in der GUI erstellt und die entsprechende Logik kann auch dort implementiert sein. Die erstellte `QQuickItem` Klasse wird dann in QML als eigenes Objekt verwendet. Dazu muss diese in der `main()` allerdings noch als neuen Typ registrieren. Die Funktion `qmlRegisterType<Type>();` erfüllt diese Aufgabe. Der Namen der Klasse wird unter „Type“ angegeben. Dazu kann noch eine Versionsnummer vergeben werden. In QML selbst lässt sich die eigene Klasse dann als Package importieren und sie kann wie jedes andere Objekt in QML verwendet werden. Dazu gehören auch Funktionen wie eine Animation, Transformation und generelle Attribute wie Position, Größe oder Farbe. [10]

Dieser Ansatz erlaubt es mit etwas mehr Aufwand, den ganzen Funktionsumfang von QML zu nutzen. Es können beliebig viele eigene `QtQuickItem` Klassen erstellt und alle auf dem gleichen Fenster der Anwendung anzeigt werden, die aber alle unabhängig voneinander sind.

3. QQuickFrameBufferObject

   Es wurden nun zwei Ansätze vorgestellt, die relativ verschieden waren. Der letzte Ansatz ähnelt sehr dem `QQuickItem` Ansatz, bietet jedoch noch etwas mehr Möglichkeiten. Hierbei handelt es sich um eine Unterklasse von `QtQuickItem`. Dieser Ansatz erlaubt es uns ebenfalls eigenen OpenGL-Code in QML zu rendern. Dabei wird in ein `OpenGL frame buffer` gerendert und Qt generiert daraus den benötigten Szenengraph.

   Die Vorgehensweise ist ebenfalls ähnlich wie bei dem `QQuickItem`. Es wird wieder eine eigene Klasse erstellt, die von `QQuickFrameBufferObject` erbt. Dabei muss nur eine Funktion reimplementiert werden: `Renderer* createRenderer() const;` Diese Funktion gibt ein neu erstelltes Renderer-Objekt zurück. Die Klasse des Renderers muss ebenfalls erstellt werden. Das Renderer-Objekt muss von der Klasse `QQuickFrameBufferObject::Renderer` erben. Die erstellte Renderer Klasse ist für das Rendern zuständig, daher werden mehrere Funktionen reimplementiert. Zum einen muss es eine Funktion namens `QOpenGLFramebufferObject *createFramebufferObject(const QSize size)` geben, die ein neues `QQuickFrameBufferObject` zurückgibt, was in diesem Fall die

erstellte Subklasse ist. In dieser Funktion gibt die Möglichkeit zu bestimmen, wie das Format des Framebuffers aussehen soll. Dies ist einer der Vorteile gegenüber einfachen `QQuickItem`. Das Format wird durch ein Objekt der Klasse `QOpenGLFramebufferObjectFormat` bestimmt. Dort kann zum Beispiel die Anzahl der Samples pro Pixel oder Depth und Stancil Buffer gesetzt werden. Zum anderen wird noch eine Reimplementierung der Funktion `render()` benötigt. In diese Funktion kommt der ursprünglicher OpenGL-Code. Wichtig hierbei ist wieder, dass die OpenGL-API richtig verwendet wird (siehe 5.1.3.1). Eine Funktion zum Initialisieren von OpenGL ist nicht notwendig, da das in dem Konstruktor der Render-Klasse geschieht. Am Ende der `render()` Funktion wird die Funktion `update()` aufgerufen, damit der `FrameBuffer` noch ein mal rendert. Bei dem Aufruf der `update()` Funktion wird eine weitere Funktion aufgerufen, die reimplementiert werden muss. Dabei handelt es sich um `synchronize(QQuickFramebufferObject *item)`. Da dieser Ansatz, wie auch `QQuickItem`, darauf basiert, dass es zwei Threads gibt, wird diese Funktion genutzt um zwischen den beiden Threads zu kommunizieren. Außerhalb dieser Funktion ist dies nicht möglich. Auch Variablen untereinander auslesen oder beschreiben darf ausschließlich in dieser Funktion stattfinden. Das liegt vor allem daran, dass ein exklusiver Zugriff nicht sichergestellt ist. Während der `synchronize()` Funktion, ist der GUI-Thread pausiert und der Render-Thread kann sicher Variablen lesen oder setzen.

Um die erstellten Klassen zu nutzen, werden wie bei dem `QQuickItem` die Klasse ebenfalls in der `main()` registriert. Dies geschieht wieder mit der Funktion `qmlRegisterType<Type>()`. In QML wird dann nur noch der neu registrierten Typ importiert und er kann wie jedes andere QML Item verwendet werden. [10]

Im Vergleich zu dem Ansatz mit dem `QQuickItem` Modul, muss das Verbinden mit den Signalen und Slots nicht selbst erledigen. Dadurch wird der Code deutlich lesbarer und ist einfacher zu erstellen.

Damit sind alle Ansätze vorgestellt. Es stellt sich aber die Frage, welcher der Ansätze für das Entwickeln der App am besten geeignet ist. In Qt Creator gibt es noch viel mehr Möglichkeiten, normalen OpenGL code zu verwenden. Allerdings sind viele davon veraltet und sollten nicht mehr genutzt werden. Zudem würde der zeitliche Rahmen es nicht zulassen, alle Möglichkeiten hier vorzustellen.

Wie zuvor schon erwähnt, hängt die Entscheidung für einen Ansatz von verschiedenen Faktoren ab. Einerseits sollte klar sein, welche Zielplattform angesprochen werden soll. Dementsprechend wird sich auf einen Widget oder QML basierten Ansatz festgelegt. Andererseits sollte klar sein, ob die verschiedenen Signalen genutzt werden sollen oder ob überhaupt QML benötigt oder benutzt wird.

Für das Entwickeln der App wurde der Ansatz, basierend auf der `QQuickFrameBufferObject` Klasse, verwendet. Das liegt einerseits daran, dass das Ziel ein mobiles Endgerät ist und sich der QML basierte Ansatz dafür eignet. Da sich mit QML in Verbindung mit dem `QtQuick Controls` Modul eine einfache Benutzeroberfläche erstellen lässt, ist dieser Ansatz auch im Bezug auf das erstellen der Benutzeroberfläche geeignet. Die Benutzeroberfläche wird in diesem Fall ein paar Möglichkeiten zur Kontrolle der Applikationen beinhalten, indem Buttons oder Slider nutzt werden. Die genauen Funktionen der Benutzeroberfläche werden in Abschnitt 5.2.3 vorgestellt.

Ein weiterer Entscheidungsgrund ist, dass mit mehreren `QQuickFrameBufferObject` Objekten die verschiedenen Overlays schnell und einfach austauscht werden können. Dazu wird jedes Overlay in einem eigenen `QQuickFrameBufferObject` erstellt. So besteht die Möglichkeit, per Knopfdruck von der analogen zur digitalen Uhr zu wechseln, ohne dass die eigentliche Visualisierung im Hintergrund beeinträchtigt wird.

5.2 ENTWICKLUNG DER APP

Um den Prototyp auch in Bezug auf die Visualisierung mobil zu halten, wird die Visualisierung komplett ausgelagert und findet nicht mehr auf dem Raspberry Pi mit Bildschirm statt. Dafür wird die Visualisierung in einer App realisiert. Wie bereits erwähnt, wird hierfür das Qt Framework genutzt, da die App nicht an eine bestimmte Zielplattform gebunden sein soll. Der erste Schritt besteht darin, den vorhandenen Code genau anzuschauen und zu verstehen, was dort eigentlich passiert. Der folgende Teilabschnitt wird den Programmablauf etwas genauer erklären und beschreiben, welche Teile verändert werden müssen und welche wieder verwendet werden können.

5.2.1 *Vorhandene Codebasis*

Dieser Abschnitt stellt den vorhandenen C++ code vor, der bisher auf dem Raspberry Pi ausgeführt wird und für die Visualisierung zuständig ist. Die Basis stellt die Klasse VisualHearingAid. Von der Basisklasse erben die beiden Klassen `VHAWatch` und `VHADigital`. Daneben gibt es noch die Klassen `VectorCharacter` und `SoundEventSource`. Die Basisklasse verfügt über eine `Init()` Funktion, die am Anfang aufgerufen wird. Diese erstellt zunächst ein `X11 Window` beziehungsweise ein `EglDisplay`, wenn es sich nicht um einen Raspberry Pi der 4. Generation handelt. Anschließend wird noch ein `eglWindowSurface` erstellt, bei dem man Attribute wie die Buffer Size oder die Farbtiefe einstellen kann. Das Fenster und die `eglWindowSurface` werden dann noch mit dem aktuellen `eglContext` verbunden, damit das Fenster zum rendern genutzt werden kann. Daraufhin wird die private Funktion `initOGL()` aufgerufen, die den OpenGL-Teil Initialisiert. Dabei werden ein Fragment- und ein Vertexshader erstellt und kompiliert. Dazu kommt noch eine Textur, die ein `SoundEvent` repräsentiert. Dafür wird ein PNG Bild

von einer Kugel genutzt, die in Blender erstellt worden ist (siehe 5.4). Um das Programm zu starten, wird die public Funktion `loop()` aufgerufen. Diese prüft zunächst, ob es eine Benutzereingabe gibt, die zum Beispiel das Rendern einfrieren kann oder das Programm beendet. Anschließend werden die Daten von dem Martrix Creator empfangen. Dafür wird ein Objekt der Klasse `SoundEventSource` verwendet, das eine Socketverbindung implementiert und auf der `sys/socket.h` Bibliothek basiert. Die Daten werden im JSON-Format empfangen(siehe 4.1). Aus den Daten wird ein Struct namens `SoundEvent` erstellt. Ein SoundEvent ist wie folgt definiert.

Listing 5.2: SoundEvent struct

```
struct SoundEvent {
  union {
    int timeStamp;
    float secondsAgo;
  };
  float x,y,z,E;
  int id;
  float r,g,b,a;
};
```

Das `SoundEvent` speichert einen Zeitstempel und die Sekunden, die vergangen sind, nachdem das Event empfangen worden ist. Dazu werden noch Koordinaten und Energie gespeichert, die die Größe beziehungsweise die Lautstärke bestimmt. Die Farbe eines Events wird durch `RGBA`-Werte dargestellt. Die empfangenen Events werden in einer `deque` gespeichert. Nach dem Empfangen wird die `render()` Funktion aufgerufen, die die `SoundEvents` in einem Vector speichert und basierend auf den Attributen rendert. Die gerenderten `SoundEvents` werden mit der Zeit kleiner und wandern in die Mitte des Bildschirmes, was einen 3D Effekt verursacht, als sei das Ganze ein Zeitstrahl. Abbildung 4.1 zeigt ein Bild der gerenderten Szene. Auf der Abbildung ist auch ein Uhr-Overlay zu sehen. Dieses Overlay entsteht, wenn anstatt der Basisklasse eine der abgeleiteten Klassen verwendet wird. `VHAWatch` und `VHADigital` initialisieren die Basisklasse wie beschrieben, aber beinhaltet selbst noch Code, um ein Uhr-Overlay zu generieren. Bei `VHAWatch` handelt es sich um eine analoge Uhr mit Zifferblatt. `VHADigital` stellt eine digitales Uhr dar. `VHAWatch` nutzt einen weiteren Fragment- und Vertexshader, der die Uhr rendert, nachdem die `render()` Funktion von der Basisklasse aufgerufen worden ist. `VHADigital` benötigt hierbei keinen eigenen Shader. Es wird die Klasse VectorCharacter benutzt, um aus der aktuellen Uhrzeit eine digitale Anzeige zu erstellen. In der `main()` wird dann nur noch ein Object der gewünschten Klasse, also entweder eine analoge oder digitale Uhr oder einfach die Visualisierung ohne Overlay, erstellt und die jeweilige `Init()` und `loop()` Funktion aufgerufen. [16]

Damit ist der vorhanden Code relativ überschaubar. Allerdings lässt er sich nur auf einem Raspberry Pi oder einem Linux System ausführen. Ziel des Prototyps ist es aber, dass das Programm als App auch auf mobilen

Android oder iOS Geräten laufen soll. Der nächste Abschnitt dokumentiert das Entwickeln dieser App.

5.2.2 *Portieren des vorhandenen Codes*

In diesem Teilabschnitt wird die eigentliche Entwicklung der App genauer beschrieben. Die Aufgabe besteht darin, das vorhandene Programm für mobile Geräte lauffähig zu machen. In den vorherigen Abschnitten wurden nun alle Grundlagen zu Qt Creator, dem vorhandenen Code und den verschiedenen Ansätzen zur Entwicklung einer App mit OpenGL-Code, vorgestellt. Dieser Teilabschnitt dokumentiert nun, wie mit der vorhandenen Codebasis und dem ausgewählten Ansatz des `QQuickFrameBufferObject` eine lauffähige Applikation entwickelt wird. Die erste Aufgabe bestand darin, die Teile des vorhandenen Codes anzuschauen, für die Qt eigene Ansätze vorsieht. Das ursprüngliche Programm stellt die Basisklasse `VisualHearingAid`, von der die beiden Unterklassen für die Uhr-Overlays erben. Um den Einstieg zu erleichtern, wird sich vorerst nur auf die Basisklasse konzentriert und alles andere weggelassen. Dazu wird bewusst eine alte Version des Programms als Basis genutzt, welche ohne Texturen auskommt. Die Daten, die das Programm von dem Raspberry Pi empfängt, werden zudem simuliert. Das Ziel ist es zuerst ein lauffähiges Grundprogramm zu erstellen, damit die Vorgehensweise vertraut ist. Die alte Version des Programmes mit simulierten Daten ist um einiges überschaubarer und es muss weder eine Netzwerkverbindung hergestellt werden, noch einen Raspberry Pi benutzt werden. Wie zuvor beschrieben, nutzt der ursprüngliche Code einige Methoden, für die Qt eigene Ansätze hat (siehe 5.2.1). Eine der ersten Methoden, die ausgetauscht werden muss, ist das Erstellen der Fenster. Wie bereits erwähnt, wird dafür die `X11` Bibliothek beziehungsweise ein `EglDisplay` verwendet. In Qt werden diese Ansätze nicht gebraucht, da die Fenster in QML verwaltet und erstellt werden. Die Benutzereingaben für das Beenden oder für das Einfrieren der Visualisierung wird ebenfalls über die `X11` beziehungsweise das `EglDisplay` realisiert. Da das aber nicht zwingend notwendig ist, werden Benutzereingaben erst einmal weggelassen. Die erstellte Klasse, die von `QQuickFrameBufferObject` erbt, ist dann relativ überschaubar, da sie bisher nur als Repräsentant in QML zuständig ist:

Listing 5.3: FrameBufferObject Klasse

```
1
class VisualHearingAid : public QQuickFramebufferObject
{
public:
    explicit VisualHearingAid(QQuickItem* parent = nullptr);
6   Renderer* createRenderer() const override;
};
```

Die Klasse besitzt lediglich einen Konstruktor und die reimplementierte Funktion `createRenderer()`, die, wie in 5.1.3.1 beschrieben,

für das Erstellen der Render-Klasse zuständig ist. In dieser Funktion wird ein `OpenGLFramebufferObject` mit dem Standard Format erstellt und zurückgegeben. Die dazugehörige Render-Klasse, die von `QQuickFrameBufferObject::Renderer` erbt, hat dabei aber um einiges mehr an Funktionen und Variablen. Die Klasse erbt von `QOpenGLFunctions`, wodurch ein extra Object der Klasse `QOpenGLFunctions` nicht benötigt wird, wenn man OpenGL Funktionen aufrufen möchte. Zum Initialisieren wird die Funktion `initializeOpenGLFunctions()` aufgerufen. Dies geschieht in diesem Fall im Konstruktor. Da es sich bei dem ersten Versuch um eine ältere Version ohne Texturen handelt, ist dieser ebenfalls relativ gering. Es werden lediglich ein Fragment- und Vertexshader erstellt, die dem Shader-Objekt hinzugefügt und gelinkt werden. Der Shader-Code ist dabei in Variablen als `Char Array` gespeichert. Anschließend werden noch OpenGL Attribute gesetzt, die sich aus den `SoundEvents` ergeben (siehe 5.2).

Listing 5.4: VisualHearingAidRenderer Konstruktor

```
VisualHearingAidRenderer::VisualHearingAidRenderer()
:m_program(0),
  _width(640),
  _height(480)
{
  initializeOpenGLFunctions();

if (m_program) {
    delete m_program;
    m_program = 0;
}
  m_program = new QOpenGLShaderProgram;
  m_program->addShaderFromSourceCode(QOpenGLShader::Vertex,
      vertexShaderSource);
  m_program->addShaderFromSourceCode(QOpenGLShader::Fragment,
       fragmentShaderSource);
  m_program->link();

  _aSecondsAgo = m_program->attributeLocation("secondsAgo");
  _aDirection = m_program->attributeLocation("direction");
  _aEnergy = m_program->attributeLocation("energy");

  if (_aSecondsAgo < 0 || _aDirection < 0 || _aEnergy < 0)
    throw runtime_error("Unable to get uniform location");
}
```

Die `synchronize()` Funktion ist in diesem Fall noch leer, da es in der früheren Version noch nicht notwendig war, Daten zwischen den Threads auszutauschen. In der `Render()` Funktion kann der existierenden Code direkt kopieren werden und es müssen nur Änderungen vorgenommen werden, die den Shader betreffen.

Listing 5.5: VisualHearingAidRenderer render

```
1
void VisualHearingAidRenderer::render()
  {
  receive();
  initializeOpenGLFunctions();
6
    m_program->bind();
    glClearColor (0.0, 0.0, 0.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);
    glDrawArrays (GL_POINTS, 0, _events.size());
11
    for(int i=0; i< _events.size(); i++)
    {
        _events[i].secondsAgo += 1.0/60.0;
    }
16  m_program->release();
    update();
  }
}
```

Am Anfang der Funktion findet noch ein Aufruf der `receive()` Funktion statt. In dieser werden nur die simulierten Daten, die in einer eigenen Header-Datei vorliegen, in das `vector<SoundEvent> events` Object kopiert. Auf diese Funktion wird hier nicht näher eingegangen, da sie zu einen späteren Zeitpunkt nochmal erklärt wird. In der letzten Zeile wird die `update()` Funktion aufgerufen, die dafür sorgt, dass das `FrameBufferObject` die Szene erneut rendert.

Damit sind alle notwenigen Schritte getan, um die Klasse in QML zu verwenden. In der `main()` wird sie nur noch registriert und es kann ein `VisualHearingAid` Object in QML wie jedes andere Objekt verwenden werden. Abbildung 5.3 zeigt das Ergebnis des ersten Versuchs.

Listing 5.6: VisualHearingAid Typ registrieren

```
int main(int argc, char *argv[])
{
QGuiApplication app(argc, argv);

qmlRegisterType<visualhearingaid>("VisualHearingAid", 1, 0, "
    VisualHearingAid");

QQmlApplicationEngine engine;

engine.load(QUrl(QStringLiteral("qrc:/main.qml")));
10
if (engine.rootObjects().isEmpty())
    return -1;

return app.exec();
15 }
```

Listing 5.7: VisualHearingAid Objekt in QML

```
1 VisualHearingAid          {
2     id : FrameBufferObject
3     anchors.fill: parent
4     }
```

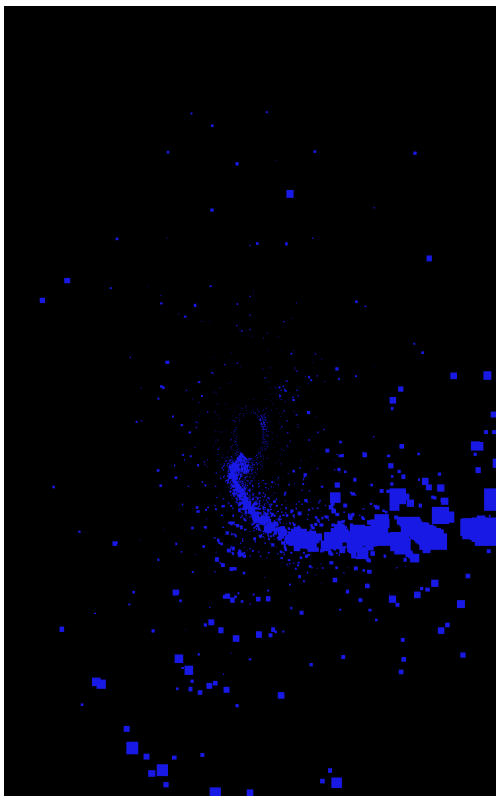

Abbildung 5.3: Screenshot der ersten FrameBufferObject Applikation

Dieser Ansatz wird genutzt, um Erfahrung im Umgang mit der Vorgehensweise von dem `FrameBufferObject` zu sammeln. Da das Ergebnis dem Ziel schon nahe kommt, wird auf dieser Basis der aktuelle Code integriert. Mit dem vorhandenen Grundgerüst, ist der Aufwand relativ überschaubar. Im OpenGL-Teil haben sich die beiden Shader geändert und es kommen die fehlenden Attribute, wie in 5.2 beschrieben, dazu. In der alte Version des Codes wurden blaue Vierecke benutzt, um ein `SoundEvent` zu repräsentieren. Der neue Code setzt hier auf eine Textur einer Kugel (siehe Abbildung 5.4). Das PNG Bild der Kugel wird hierzu eingelesen und eine Textur erstellt. Das Einlesen erfolgt über ein `QImage` Objekt, was anschließend mit `glTexImage2D` an eine erstellte Textur gebunden wird.

Die verschiedenen Uhr-Overlays sind in der App etwas anders Implementiert als bei dem vorhandenen Code. Dort sind sie als Unterklassen der `VisualHearingAid` Klasse implementiert. In der App sind sie allerdings eigenständige Klassen. Das hat den Grund, dass die Overlays als eigene Objekte in QML verwendet werden sollen. Dadurch sind diese nicht mit-

Abbildung 5.4: Bild der Kugel für die Textur [16]

Listing 5.8: Einbinden der Kugel als Textur

```
QImage sphere = QImage(QString("sphere.png")).mirrored();
glGenTextures(1, &_texture);
glBindTexture(GL_TEXTURE_2D, _texture);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, sphere.width(),
    sphere.height(),
                 0, GL_RGBA, GL_UNSIGNED_BYTE,
                sphere.bits());
```

einander verbunden und es besteht die Möglichkeit die Visualisierung im Hintergrund „einfrieren“, während die Uhr weiterläuft. Es besteht auch die Möglichkeit zur Laufzeit zwischen den beiden Overlays zu wechseln. Die Overlay-Klassen haben den gleichen Aufbau wie `VisualHearingAid`. Jedes Overlay hat ihre eigene Render-Klasse und kann in QML wie jedes andere Item genutzt werden. Da bisher nur Testdaten genutzt werden, wird die `receive()` Funktion auf den neuen Stand gebracht. Diese Funktion basiert auf einer Websocket-Verbindung und empfängt die JSON Daten (siehe 4.1). Das Empfangen der Daten findet in der gleichen Klasse statt, die auch die Visualisierung rendert. Wie bereits beschrieben, basiert aber der genutzte Ansatz auf der Trennung zwischen dem Render-Thread und dem GUI-Thread (siehe 5.1.3.1). Die Daten werden also von dem Render-Thread vor jedem Rendern empfangen. Das entspricht natürlich nicht der Aufgabe eines Render-Threads. Um dieses Problem zu lösen, wird das Empfangen in den GUI-Thread ausgelagert, um den Render-Thread zu entlasten und eventuelle Performanceprobleme zu vermeiden.

Die Websocket-Verbindung ist in einer eigenen Klasse `SoundEventSource` Klasse implementiert. Diese Klasse besitzt eine `get()` Methode, die neue `SoundEvents` empfängt, welche dann wiederum in einer `deQueue` gespeichert werden. Das Objekt der Klasse wird von der Render-Klasse in die GUI-Klasse `VisualHearingAid` verschoben. Im Konstruktor wird die `receive()` Funktion mit einem Signal eines Timers verbunden, was die Funktion all 16ms aufruft (Dies entspricht in etwa ein mal pro gerendertem Bild bei 60 Bildern pro Sekunde). Um die empfangenen Daten in dem Rendere-Thread anzuzeigen, wird die `synchronize()` Funktion genutzt. Wie zuvor beschrieben (siehe 5.1.3.1), soll nur dort ein Austauschen der beiden Threads geschehen. Dort kann dann die `deQueue` des Rendere-Threads mit den empfangenen Da-

ten befüllt werden. Die folgenden Code-Abschnitte zeigen den Timer und das Synchronisieren der beiden Threads:

Listing 5.9: Timer zum Empfangen der Daten und Synchronisation

```
VisualHearingAid_FBO::VisualHearingAid_FBO(...)
{
  QTimer *soundTimer = new QTimer(this);
  QObject::connect(soundTimer,SIGNAL(timeout()), this, SLOT(
      recvSoundEvents()));
  soundTimer->start(16);
}
...
void VisualHearingAid::synchronize(QQuickFramebufferObject *
    item)
{
  ...
  VisualHearingAid_FBO* obj = static_cast<
      VisualHearingAid_FBO*>(item);
  ...
  //get new SoundEvents
  vector<SoundEvent> _result = obj->soundEvents();
  _events.insert(_events.end(), _result.begin(), _result.end
      ());
}
```

Damit sind alle Schritte dokumentiert, nötig sind, um aus dem existierendem Code eine mobile Applikation zu erstellen. Abbildung 5.5 zeigt den neuen Systemaufbau. Die Visualisierung ist nun nicht mehr wie zuvor an einen Monitor gebunden (vergleiche Abbildung 4.2). Der nächste Abschnitt wird auf Erweiterungen der Benutzeroberfläche in QML eingehen.

5.2.3 *Erweiterungen der Benutzeroberfläche*

Die Benutzeroberfläche der fertigen Applikation auf Basis des vorhandenen Codes wird noch erweitert. Die vorhandenen Möglichkeit, dass die Visualisierung eingefroren werden kann, wird mit einer `onClick()` Funktion eines durchsichtigen Button realisiert. Dieser Button sitzt auf der Visualisierung und ist in QML implementiert. Damit der Button nicht sichtbar ist, wird das `opacity` Attribut auf 0 gesetzt. Beim Drücken des Buttons wird eine Variable gesetzt, die vor jedem Rendern in der `synchronize()` Funktion geprüft wird. Damit die Visualisierung einfriert, wird der Aufruf der `update()` Funktion ausgesetzt und die Szene wird nicht erneut gerendert. Um die Applikation weiterführend zu steuern, ist ein kleines Menü in QML implementiert. An der linken oberen Ecke ist ein Button, der das Menü öffnet. Das Menü selbst ist erscheint über der Szene, damit man vorgenommene Einstellungen direkt sehen kann. Es besteht unter anderem aus drei Slidern, mit denen man die X-, Y-, und Z-Achse der Visualisierung steuern kann. Intern wird eine Transformation der Szene auf der jeweiligen Achse vorgenommen. Dazu werden noch die Kompassdaten von dem mobilen Gerät, dem Matrix Creator und

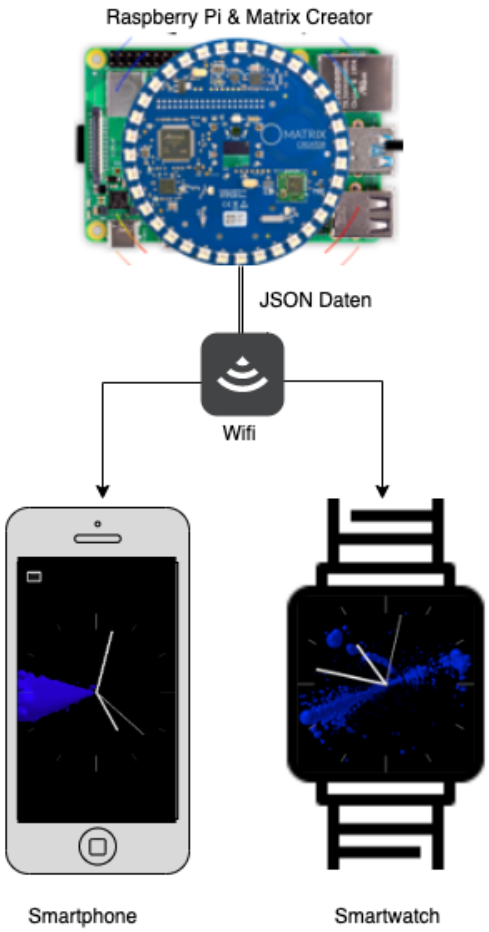

Abbildung 5.5: Systemaufbau des neuen Prototyps mit Smartphone und Smartwatch

die Differenz der Beiden angezeigt. Diese Daten werden benötigt, um die Ausrichtung des mobilen Gerätes und des Matrix Creators anzupassen. Der Abschnitt 6.1 beschreibt das Errechnen der Ausrichtung. Im unteren Teil des Bildschirms befinden sich noch folgende Buttons:

- `Calibrate`: Um die beiden Kompass zu kalibrieren.
- `Switch Overlay`: Um zwischen den verschiedenen Overlays zu wechseln.
- `Help`: Zeigt ein Dialog an, der die Benutzung der App beschreibt.
- `Use Comapss`: Aktiviert und deaktiviert das Anpassen der Visualisierung mit den Kompassdaten.
- `Compass Port`: Stellt den Port des Sockets für den Matrix-Kompass ein.
- `Matrix Port`: Stellt den Port für die ODAS-Daten ein. Für Testdaten kann man die 0 als Port festlegen.

Abbildung 5.6 zeigt links einen Screenshot von diesem Menü und rechts ein Screenshot der fertigen App ohne Menü.

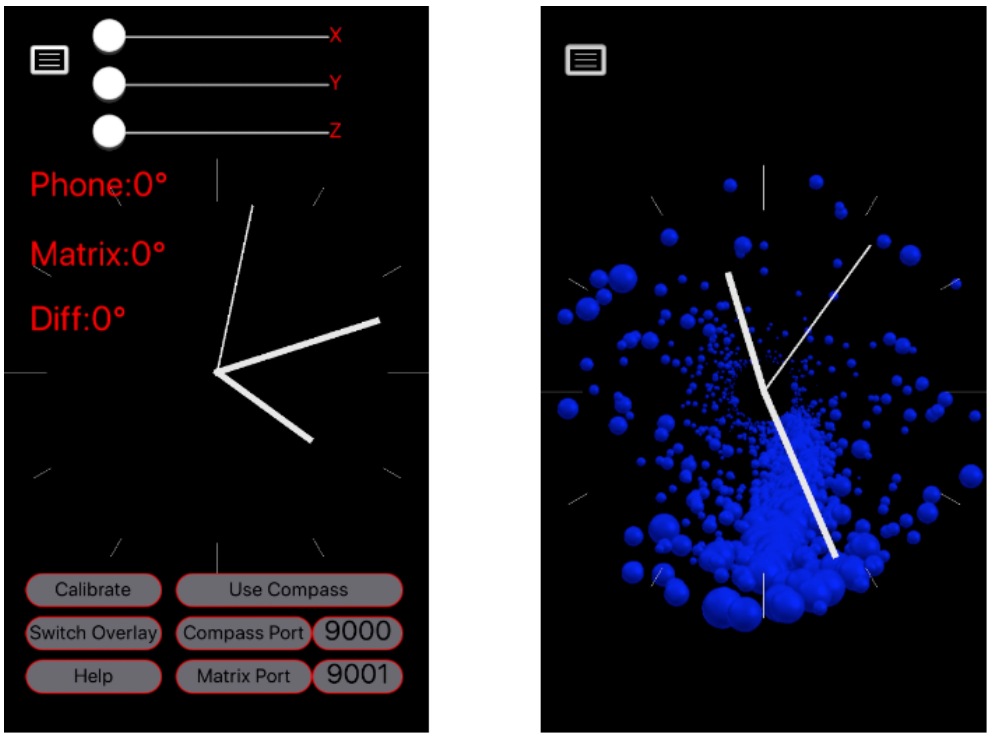

Abbildung 5.6: Benutzeroberfläche in QML (links) und fertige Applikation ohne Menü(rechts)

5.3 GESAMTSYSTEM DES PROTOTYPS

Dieser Abschnitt wird kurz das Zusammensetzen der Komponenten anschneiden. Wie bereits erwähnt, wird das System in einem Hut oder einer Kappe versteckt, so dass der Benutzer das gesamte System bei sich am Körper tragen kann. Damit ist die Anforderung der Mobilität gewährleistet. Abbildung 5.7 zeigt eine grobe Zeichnung dieses Aufbaus. In der Abbildung wurde allerdings die Stromquelle nicht mit eingezeichnet.

Der Benutzer kann diese Kappe nun tragen, ohne dass er sich weiter verkabeln muss. Mit der mobilen Applikation kann sich der Benutzer die Visualisierung auf einem Smartphone oder einer Smartwatch anzeigen lassen. Das gesamte System ist somit ohne großen Aufwand benutzbar und für Außenstehende nicht ohne weiteres zu erkennen. Allerdings ist der Prototyp mit einer Powerbank zu groß, um ihn in einer normalen Kappe zu verstauen. Abbildung 5.8 zeigt den Prototyp neben einer Kappe. Um dieses Problem zu lösen, muss eine größere Kopfbedeckung gewählt oder der Prototyp verkleinern werden. Der Hersteller des Matrix Creators stellt allerdings noch ein anderes Entwicklerboard namens Matrix Voice her. Dieser ist deutlich kleiner und ist mit einem ESP32 ausgestattet. [8] Der Prototyp im Rahmen

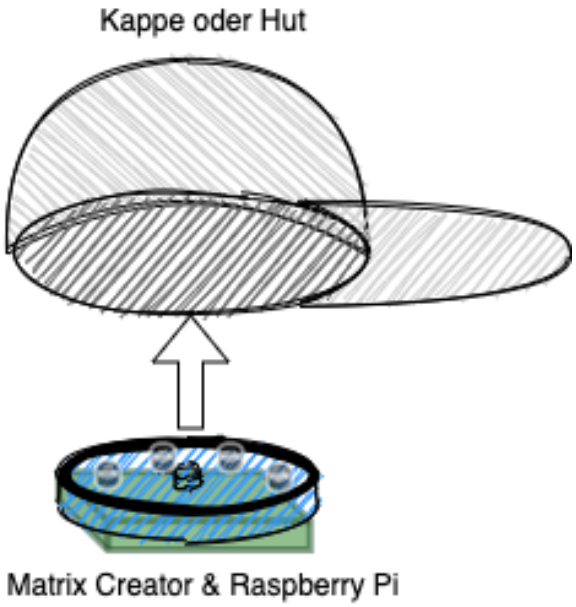

Abbildung 5.7: Zeichnung der Hörhilfe in einer Kappe

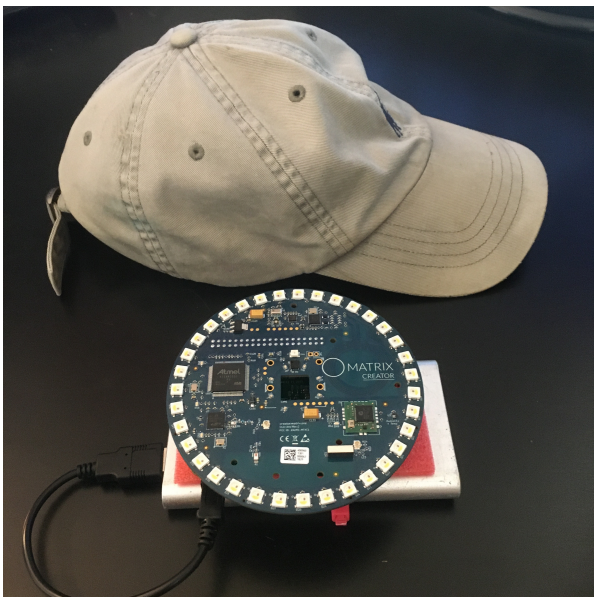

Abbildung 5.8: Hörhilfe mit Powerbank neben einer Kappe

dieser Arbeit ist hiermit nahezu fertiggestellt. Da allerdings die Visualisierung nicht mehr an einen externen Monitor gebunden ist, bedarf es noch weiterer Funktionalitäten, um sicherzustellen, dass der Benutzer auch die richtigen Informationen erhält. Zuvor war die Ausrichtung zum Monitor immer die Gleiche. Nun kann man aber das Handy oder die Hörhilfe, beziehungsweise den Kopf frei bewegen und drehen. Dies führt natürlich zu einer Abweichung, die man ausgleichen muss.

Der nächste Abschnitt wird sich mit diesem Problem beschäftigen und verschiedene Ansätze zur Koordination von dem mobilen Gerät und der Hörhilfe vorstellen. Anschließend wird der Lösungsansatz des Problems dokumentiert und bewertet.

6 KOORDINATENSYSTEME UND ORIENTIERUNG

6.1 AUSRICHTUNG DER VISUALISIERUNGEN

Dieser Abschnitt befasst sich mit den verschiedenen Orientierungsmöglichkeiten an verschiedenen Koordinatensystemen. Ohne eine interne Angleichung der Ausrichtung mit entsprechender Korrektur, ist es dem Benutzer nicht möglich, Informationen bezüglich der Geräuschquellen im Raum zu erhalten. Um eine Richtung anzeigen zu können, muss immer von einem Referenzpunkt ausgegangen werden. Dafür kommen mehrere Möglichkeiten in Frage. Es kann der Benutzer selbst als Referenzpunkt genutzt werden. Dabei wird in jedem Fall davon ausgegangen, dass ein Geräusch links neben dem Benutzer auch links von ihm angezeigt wird. Dabei soll die räumliche Ausrichtung des mobilen Gerätes beachtet werden und die Visualisierung dementsprechend ausgerichtet werden.

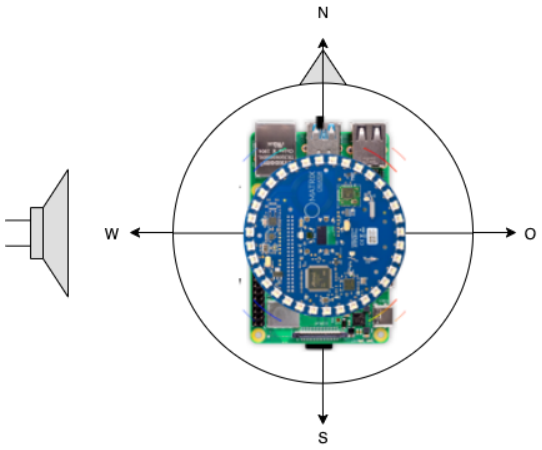

Abbildung 6.1: Benutzer mit VisualHearingAid als Zeichnung

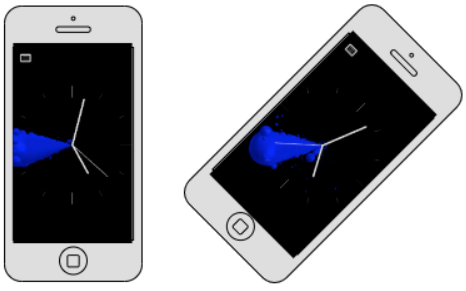

Abbildung 6.2: Beispiel der Visualisierung mit angepasster Ausrichtung

Die Grafik 6.1 ist eine Zeichnung eines Benutzers, der den neuen Prototyp trägt. In diesem Fall auf dem Kopf, aber ohne Kopfbedeckung. Der Benutzer mit dem Matrix Creator wird durch den Kreis mit den Himmelsrichtungen aus der Vogelperspektive dargestellt. Die Spitze am nördlichen Punkt soll die Ausrichtung des Benutzers anzeigen, beziehungsweise seine Blickrichtung darstellen. Von Westen her werden durch den Lautsprecher Geräusche erzeugt. Die Abbildung wird in diesem Abschnitt als Referenz genommen, aus der die Visualisierungen in den anderen Abbildungen entstehen.

Die zweite Grafik 6.2 zeigt zwei Smartphones. Sie zeigen die Visualisierung der Geräusche an, wobei das eine Handy um 45° geneigt ist. Die Visualisierung auf den Handys zeigt in beiden Fällen nach links, auch wenn das rechte Handy geneigt ist. Die Neigung des Handys wird also ermittelt und die Visualisierung dementsprechend gedreht, so dass sie die Geräuschquelle immer an der gleichen Stelle anzeigt. Diese Möglichkeit ist vergleichbar mit einem Kompass, der immer nach Norden zeigt. Die nächste Möglichkeit ist, dass die Hörhilfe als Referenzpunkt dient. Allerdings wird hierbei die Ausrichtung des mobilen Gerätes nicht ausgelesen und die Visualisierung auch nicht angepasst. Die Grafik 6.3 zeigt zwei Ausrichtungen eines Handys, welche die gleiche Visualisierung auf dem Gerät anzeigt. Bei dieser Variante, ist es wichtig, dass der Benutzer die Ausrichtung des mobilen Gerätes mit seiner eigenen Ausrichtung gleichstellt. Entweder, indem er das Smartphone direkt vor sich hält oder die eventuelle Neigung des Handys beachtet, um die richtige Quelle des Geräusches zu erhalten.

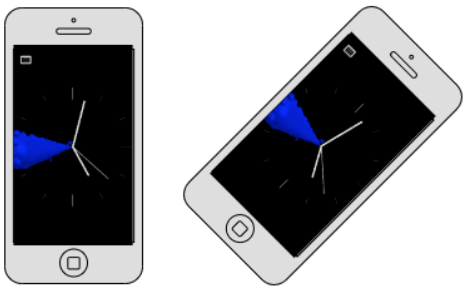

Abbildung 6.3: Beispiel der Visualisierung ohne angepasster Ausrichtung

Als letzter sinnvoller Referenzpunkt kann das Weltkoordinatensystem dienen. Dabei soll die Geräuschquelle in dem Weltkoordinatensystem von der Hörhilfe lokalisiert werden und die Visualisierung soll sich so ausrichten, dass sie auf diesen Punkt zeigt. Der Unterschied zu den beiden vorherigen Möglichkeiten besteht darin, dass es hier notwendig ist, mehr als nur die Ausrichtung eines Gerätes zu bestimmen. Es müssen Informationen über die geographische Lage vorliegen, welche am besten mit GPS ermittelt werden kann. Die geographische Lage muss dazu auf beiden Geräten vorliegen. Das Problem hierbei ist, dass der Matrix Creator kein GPS besitzt. Außerdem ist die Hörhilfe nur in der Lage eine Richtung zu bestimmen und nicht den Ursprung eines Geräusches. Damit wäre es bei einer weiteren Entfernung

zwischen der Hörhilfe und dem Gerät zur Visualisierung nicht möglich, eine genaue Richtung anzugeben. Somit kommt diese Möglichkeit im Rahmen dieser Arbeit nicht in Frage. Daher wird sich auf die ersten beiden Möglichkeiten beschränkt. Im direkten Vergleich fällt auf, dass die erste Möglichkeit für den alltäglichen Einsatz am besten geeignet ist. Die zweite Möglichkeit setzt mehr Eigenaufwand des Nutzers voraus und erschwert die Benutzbarkeit deutlich. Allerdings ist der Entwicklungsaufwand geringer, da man keine Sensordaten benötigt. I

6.2 TECHNISCHE UMSETZUNG

Dieser Abschnitt dient zur Dokumentation der technischen Umsetzung der verschiedenen Koordinationsmöglichkeiten. Die zuvor vorgestellten Möglichkeiten setzen Daten über die Ausrichtung der Hörhilfe und des mobilen Gerätes zur Visualisierung voraus. Für die technische Umsetzung muss sichergestellt werden, dass das mobile Gerät, sowie der Matrix Creator, Sensoren besitzen, die es ermöglichen die Ausrichtung im Raum zu erkennen. Da jedes aktuelle Smartphone oder jede Smartwatch über eine Vielzahl an Sensoren verfügt, stellt kein Problem dar. Der Matrix Creator besitzt ebenfalls einige Sensoren. Neben Luftfeuchtigkeit, Luftdruck und UV-Sensoren, verfügt er auch über eine inertial measurement unit (IMU). [9] Eine IMU besteht aus mehreren Sensoren. Meistens bestehen die Sensoren aus mehreren Beschleunigungssensoren, Gyroskopen und Magnetometern. Jeweils einer dieser Sensoren wird pro Achse genutzt. Die Achsen nennen sich Pitch, Roll und Yaw. Die Achsen lassen sich mit denen eines kartesisches Koordinatensystems vergleichen, wobei Pitch einer Drehung um die Y-Achse, Roll einer Drehung um die X-Achse und Yaw einer Drehung um die Z-Achse entsprechen. Die IMU liefert für jede dieser Achsen einen Wert. In diesem Fall wird allerdings nur der Yaw-Wert benötigt, da dieser die Ausrichtung liefert. Die anderen Werte spielen aber trotzdem eine Rolle, wenn man genaue Werte haben möchte. Der Yaw-Wert liefert nur dann genaue Werte, wenn das Gerät, beziehungsweise der Sensor horizontal auf der XY-Ebene liegt. Sobald das Gerät auf einer der anderen Achsen geneigt ist, wird der Yaw-Wert verfälscht. Daher nutzt man die Pitch- und Roll-Werte, um dem entgegenzuwirken. Dieses Vorgehen nennt sich Tilt Correction. Dabei werden die Pitch- und Roll-Werte ausgelesen und der Yaw-Wert basierend auf den Werten angepasst. [17]

Um auf die Sensoren des Matrix Creators zuzugreifen, wird vom Hersteller eine Bibliothek namens Hardware Abstraction Layer (HAL) zur Verfügung gestellt. [7] Dabei handelt es sich um eine Programmierschnittstelle in C++, die es dem Entwickler erlaubt, auf die Hardware und damit auch auf die Sensoren des Matrix Creators zuzugreifen. Nach der Installation der Bibliothek, können die IMU-Daten mit einem `IMUSensor` Objekt ausgelesen werden. Die Daten werden über ein `IMUData` Objekt repräsentiert, was für jede Achse ein Beschleunigungs-, Gyroskop- und Magnetometerwert hält. Die benötigten Yaw-, Pitch- und Roll-Werte lassen sich dort auslesen. Da

es sich dabei um eine low-level Bibliothek handelt, ist die IMU nicht kalibriert und die Tilt Correction wird auch nicht angewendet. Allerdings sind in dem Repository der Schnittstelle einige Demos enthalten, darunter auch eine Kompass-Demo, die die Tilt Correction und Kalibrierung der Sensoren übernimmt. Um die Kompass-Daten des Matrix Creator auszulesen, wird die Demo erweitert, indem eine Socketverbindung zur Hörhilfe hergestellt wird. Neben den Daten, die ODAS sendet, empfängt die Hörhilfe ebenfalls die Ausrichtung des Matrix Creators. In diesem Fall genügt ein einfacher `int`-Wert, um die Ausrichtung zu übertragen, da nur Zahlen im Bereich von 0 bis 359 benötigt werden. Für das Auslesen der Sensoren des mobilen Gerätes wird eine eigene Klasse erstellt, die von `QCompassFilter` erbt. `QCompassFilter` selbst erbt von `QSensorReading`, welche eine Schnittstelle zu vielen Sensoren bereitstellt. [10] Die Klasse besitzt ein `QCompass` Objekt, das im Konstruktor der Klasse angemeldet wird. Es besteht die Möglichkeit einige Einstellung vorzunehmen, wie zum Beispiel das Überspringen von identischen Werten. Um die Kompassdaten auszulesen, wird eine Funktion genutzt, die mit einem Timer verbunden ist und alle 16ms aufgerufen wird. In dieser Funktion werden auch die Daten vom ODAS empfangen. Das genaue Vorgehen wurde im Abschnitt 5.2.2 erklärt.

Um nun auf Basis der Kompassdaten die Visualisierung anzupassen, werden die Unterschiede beider Geräte errechnet. Da der Kompass des Matrix Creators zum Start nicht die gleichen Werte wie das mobile Gerät liefert, werden beide Geräte zuerst kalibriert. Dafür werden beide Geräte in die gleiche Ausrichtung gebracht und die Kompassdaten ausgelesen. Der ausgelesene Wert wird als Offset gespeichert und bei jedem weiteren Auslesen abgezogen. Dabei muss allerdings beachtet werden, dass sich der Wertebereich von [0,359] zu [0-Offset,359-Offset] ändert. Um diese Verschiebung des Wertebereichs auszugleichen, wird 360 addiert, falls der Wert ins Negative fällt. Damit ist der Wertebereich wieder bei [0,359]. Das ist notwendig, da das Offset beider Geräte nicht identisch ist, was zu zwei verschiedenen Wertebereichen führt. Der Unterschied der Ausrichtung beider Geräte wird dann gespeichert. Die Variable, die den Unterschied speichert, ist mit einem Signal verbunden, was bei einer Änderung ausgesendet wird. Im QML-Teil der Applikation wird auf dieses Signal reagiert und die Visualiserung dementsprechend mit einer Transformation auf der Z-Achse gedreht. Bei dieser Vorgehensweise übernimmt der GUI-Thread das Anpassen und der Render-Thread wird nicht unterbrochen oder unnötig belastet. Wenn nun das mobile Gerät oder der Matrix Creator seine Ausrichtung ändert, wird dies direkt ausgeglichen. Die Abbildung 6.2 zeigt dies beispielhaft. Die andere Variante, die zuvor beschrieben worden ist, kommt ohne Sensordaten aus. Genauer gesagt benötigt sie keinerlei Berechnung der Ausrichtung, da diese Aufgabe auf den Benutzer ausgelagert wird. Abbildung 6.3 veranschaulicht dies beispielhaft. Wie zuvor schon beschrieben, ist diese Variante allerdings nicht zu empfehlen, da die Benutzbarkeit darunter leidet. In Gefahrensituationen oder durch Unachtsamkeit des Benutzers kann diese Variante auch

zu falschen Informationen führen. Aus den gegebenen Gründen wird also nicht näher auf diese Variante eingegangen.

Abschließend lässt sich sagen, dass die erste Variante gut funktionieren könnte. Allerdings besteht das Problem, dass die Sensoren des Matrix Creators relativ ungenaue Werte liefern. Die Ursache dieses Problems wurde bisher nicht gefunden und in dieser Arbeit auch nicht weiter behandelt. Der letzte Abschnitt wird diese Problematik noch einmal kurz aufgreifen bevor ein Fazit getroffen wird.

7 FAZIT

Dieses Kapitel befasst sich mit dem Fazit der Arbeit, indem der Inhalt und vor allem die Entwicklung des Prototypen bewertet wird und diskutiert die mögliche zukünftige Weiterentwicklung der Hörhilfe. Es wird beschrieben, inwiefern die Anforderungen erfüllt worden sind. Abschließend wird der fertige Prototyp zusammengefasst dargestellt.

7.1 BEWERTUNG

Neben dem Entwickeln des Prototyps stellt die Arbeit das Qt Framework im Bezug auf die Verwendung von OpenGL vor. Der Leser erhält hinreichende Informationen und Herangehensweisen für die Entwicklung mit Qt, die einen leichten Einstieg in dieses Framework bieten können. Beachtet man die Größe des Frameworks, ist es nicht leicht, einen Einstieg zu finden. Diese Arbeit stellt neben den Grundlagen zu dem Framework mehrere Ansätze für die Verwendung von existierendem OpenGL Code vor. Dieser Teil der Arbeit kann den Einstieg deutlich erleichtern und ist verständlich genug formuliert, dass man ohne Vorwissen ein Projekt starten kann. Das Erreichen der Anforderungen aus Abschnitt 5 hingegen erweist sich nur teilweise als erfolgreich. Es gibt einige Probleme mit dem Prototyp, die den echten Einsatz erschweren, beziehungsweise sogar verhindern. Zum einen sind die Sensoren des Matrix Creators etwas ungenau. Im Vergleich mit den Sensoren des Smartphones springen die Kompassdaten, die aus dem Matrix Creator ausgelesen werden ständig hin und her, was zu einem Zittern der Visualisierung führt. Eine Glättungsfunktion unterbindet dieses Verhalten nur bedingt. Eine Glättung der Daten führt zu einer Verzögerung der Anpassung der Visualisierung, die bis zu 2 Sekunden betragen kann. Das System sollte allerdings in Echtzeit reagieren. Ein eigenständiger Sensor könnte das Problem lösen, wurde aber nicht im Rahmen dieser Arbeit behandelt. Zum anderen ist der Prototyp zu groß um in einer Kappe versteckt zu werden, da die Kombination aus Matrix Creator, Raspberry Pi und einer Powerbank zu viel Platz verbraucht. Wie im Abschnitt 5.3 bereits erwähnt, gibt es aber die Möglichkeit, den Matrix Voice anstelle des Matrix Creators zu verwenden. Diese ist klein genug, um in eine Kopfbedeckung zu passen und benötigt auch keine Raspberry Pi, da er mit einem ESP32 ausgestattet ist.[8] Abgesehen von den Problemen mit den Sensoren und der Größe ist der Prototyp und vor allem die App einsatzbereit. Das Ausrichten der Visualisierung funktioniert in Echtzeit und gibt dem Benutzer in jeder Position die richtige Information. Die Anforderung der Mobilität ist insofern gegeben, als dass man das System mit sich führen kann. Nur nicht in einer Kopfbedeckung versteckt. Durch das Fehlen der Kopfbedeckung ist die Anforderung der Diskretion

nicht erreicht. Die Koordination wird allerdings bis auf die leicht ungenauen Kompassdaten erreicht. Insgesamt ist durch diese Arbeit der Prototyp weiterentwickelt worden und mit ein paar wenigen Anpassungen, die im letzten Abschnitt aufgegriffen werden, durchaus ein Erfolg.

7.2 ZUKÜNFTIGE ARBEIT

Zum Abschluss werden ein paar Ideen für zukünftige Arbeiten vorgestellt. Da die Idee einer visuellen Hörhilfe von Herr Udo Gebelein und Prof. Dr. Stefan Rapp erst dieses Jahr enstanden ist, dauert es noch lange bis sich aus der Idee ein richtiges Produkt entwickeln wird. Nach dieser Arbeit ist ein weiterer Schritt getan, aber es werden noch einige folgen müssen. Die zuvor beschriebenen Probleme sollten schnell gelöst sein. In ihrem Paper schreiben Herr Udo Gebelein und Prof. Dr. Stefan Rapp schon von weiteren Ideen, wie zum Beispiel Machine Learning für Geräuschklassifizierung oder Weiterentwicklung beziehungsweise Anpassung von ODAS für die Hörhilfe. [3] Das Gerät zur Visualisierung kann auch weiterentwickelt werden, indem zum Beispiel eine Argumented Reality Brille verwendet wird. In dem Paper wird auch von einer Brille mit Mikrofonen im Gestell geschrieben, was man mit dem Argumented Reality Feature verbinden könnte. Es gibt also noch viele Möglichkeiten, um die Hörhilfe weiter zu eintwickeln, so dass in Zukunft ein fertiges und einsatzbereites System entstehen kann.

Teil II

APPENDIX

LITERATUR

[1] Jens Blauert und Jonas Braasch. "Räumliches Hören". In: *Handbuch der Audiotechnik*. Hrsg. von Stefan Weinzierl. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, S. 87–121. ISBN: 978-3-540-34301-1. DOI: 10.1007/978-3-540-34301-1_3. URL: https://doi.org/10.1007/978-3-540-34301-1_3.

[2] Giuseppe D'Angelo. *Integrating OpenGL with QtQuick2*. https://www.kdab.com/integrating-opengl-with-qt-quick-2-applications-part-1/. 2015.

[3] Udo Gebelein und Stefan Rapp. "A Hearing Aid to Visualize the Direction of Sound". In: *Studientexte zur Sprachkommunikation: Elektronische Sprachsignalverarbeitung 2020*. Hrsg. von Ronald Böck, Ingo Siegert und Andreas Wendemuth. TUDpress, Dresden, 2020, S. 69–76. ISBN: 978-3-959081-93-1.

[4] Klaus Genuit und Roland Sottek. "Das menschliche Gehör und Grundlagen der Psychoakustik". In: *Sound-Engineering im Automobilbereich: Methoden zur Messung und Auswertung von Geräuschen und Schwingungen*. Hrsg. von Klaus Genuit. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, S. 39–88. ISBN: 978-3-642-01415-4. DOI: 10.1007/978-3-642-01415-4_2. URL: https://doi.org/10.1007/978-3-642-01415-4_2.

[5] François Grondin und François Michaud. "Lightweight and optimized sound source localization and tracking methods for open and closed microphone array configurations". In: *Robotics and Autonomous Systems* 113 (2019), S. 63–80.

[6] H. Harbauer. "Sprachentwicklung und ihre Störungen". In: *Lehrbuch der speziellen Kinder- und Jugendpsychiatrie*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1980, S. 378–385. ISBN: 978-3-642-96583-8. DOI: 10.1007/978-3-642-96583-8_16. URL: https://doi.org/10.1007/978-3-642-96583-8_16.

[7] *Hardware Abstraction Layer for MATRIX Creator and MATRIX Voice*. https://github.com/matrix-io/matrix-creator-hal. 2020.

[8] *MATRIX Creator*. 2020. URL: https://www.matrix.one/products/.

[9] *Matrix Creator System Architecture*. 2019. URL: https://matrix-io.github.io/matrix-documentation/matrix-creator/resources/system-architecture/.

[10] *Qt Creator Documentation*. 2020. URL: https://doc.qt.io/.

[11] *Raspberry Pi*. 2020. URL: https://www.raspberrypi.org/.

[12] *Räumliches Hören erforschen, Austrian Academy of Science, https://www.oeaw.ac.at/en/detail/event/raeumliches-hoeren-erforschen/*. 22.01.2016. URL: https://www.oeaw.ac.at/en/detail/event/raeumliches-hoeren-erforschen/.

[13] Ashutosh Saxena und Andrew Y. Ng. "Learning sound location from a single microphone". In: *2009 IEEE International Conference on Robotics and Automation* (2009), S. 1737–1742.

[14] BU Seeber. "Binaurales Hören mit Cochlea Implantaten". In: *Fortschritte der Akustik–DAGA'10*. 2010.

[15] *Szenengraph, DGL Wiki*. 2020. URL: https://wiki.delphigl.com/index.php/Szenengraph.

[16] Stefan Rapp Udo Gebelein. *VisualHearingAid*. https://code.fbi.h-da.de/s.rapp/VisualHearingAid/. 2020.

[17] S. Yosi, N. J. Agung und S. Unang. "Tilt and heading measurement using sensor fusion from inertial measurement unit". In: *2015 International Conference on Control, Electronics, Renewable Energy and Communications (ICCEREC)*. 2015, S. 193–197.

[18] *cocktail-partys und hörgeräte: biophysik des gehörs*. 2002. URL: http://medi.uni-oldenburg.de/download/docs/paper/kollmeier_2002_biophysik_gehoer.pdf.