ulm university universität ulm

Ulm University| 89069 Ulm | Germany

Faculty of Engineering and Computer Science

institute of media informatics

Moteye

Gaze-Tracking in an Automotive Context

Project Ubiquitous Computing at Ulm University

Project-Members and authors:
Andreas Köll, Ömer Şahin, Tamino Hartmann
andreas.koell@uni-ulm.de, oemer-1.sahin@uni-ulm.de, tamino.hartmann@uni-ulm.de

Reviewer:
Prof. Dr.-Ing. Michael Weber

Supervisor:
Prof. Dr. Enrico Rukzio, David Dobbelstein

Year:
2014 – 2015

Print: PDF-LaTeX 2_ε

Abstract

The project Moteye implements a gaze based prototype in an automotive context in order to interact with a car's cockpit components while keeping the driver's hands on the steering wheel. This is done using multiple eye trackers for recognizing the user's gaze and a single joystick-like interface placed on the steering wheel. The concept developed is called Gaze Context Mapping, where a physical or virtual cockpit component is selected via gaze and the joystick's input mapped to control it. The system avoids the Midas touch problem by utilizing the eye trackers only as an intent pointer, not as the user input device that actually triggers actions. To show how our proposed system works we implemented the components left side mirror, radio, and hands free call for practical use. The prototype was built on top of an existing driving simulator.

Acknowledgments

First and foremost we would like to thank David Dobbelstein and Professor Enrico Rukzio, our project supervisors, for their input and considerations to our work. Further thanks are in order to Professor Michael Weber for his help and input concerning eye tracking error filtering and to e.solutions GmbH in Ulm for some domain knowledge in car cockpit systems. Besides we thank Leo Hnatek for his support on Arduino issues. We would also like to acknowledge and thank all the various people that offered helpful feedback and critic in the project meetings on a regular basis.

Contents

1

Introduction

Moteye is a ubiquitous computing project of master students at the University of Ulm. The project's task is to develop a system in the automotive context using one or multiple eye trackers. Our team, consisting of Andreas Köll, Ömer Şahin, and Tamino Hartmann, is accompanied by David Dobbelstein and Prof. Dr. Enrico Rukzio. Furthermore the project was discussed in cooperation with e.Solutions GmbH in Ulm to integrate current practical know-how into our approach.

The main theme of this project is gaze tracking in an automotive context. Our aim is to develop a prototype of a gaze based human computer interaction concept for vehicles.

To better understand the scope of this project it is important to differentiate between eye tracking and gaze tracking. For the purpose of this project we will define and differentiate the two terms as follows, based on previous work [1]. The classic **eye tracking** measures the relative position of the eye to the head without knowledge of the

surroundings. **Gaze tracking** on the other hand is the process of tracking and recording gaze movements of a person in an area. The **gaze** determines a person's current line of sight or point of fixation. The difference is that eye tracking only gives us information on the eye movement while gaze tracking allows us to know what the user is looking at.

1.1 Motivation

Everyone who has ever driven a car knows how important it is to adjust the mirrors as well as the seat before driving. These are the most essential components which have to be customized and many of today's cars provide quite comfortable controls to do that. But if you have bought a new car or rent one, you might not be familiar with its controls. This makes it difficult or hard to use the controls. Sometimes it needs practice over a period of time to be familiar with such controls.

This starts with essential adjustments mentioned above and goes to navigating the complex interfaces of the entertainment systems. It is noticeable how the amount of electronic features have now become one of the more defining features of new vehicles[1]. However this multitude of auxiliary systems has increased the adoption time required for learning to handle all the options a vehicle offers by quite a bit. The authors know of quite a few instances where drivers aren't even aware of some of the features a vehicle offers even though they would greatly enjoy them.

So how to stem this tide of input and output modality? A look at what is currently newly available for developers as input devices offers up some interesting options. Apart from gesture detection as input, eye trackers have currently become available at a low enough price point that easy prototyping is doable. This gives us new perspectives and makes it very exciting to develop new human computer interaction interfaces in automotive context, where these play a crucial role for the satisfaction of customers and thus for the success of a product and therefore its manufacturer.

[1] See for example: `http://www.wired.com/2011/04/the-growing-role-of-software-in-our-cars/`, retrieved on 10.03.2015

Input modalities in cars are miscellaneous from button inputs over scrolling wheels up to touchscreens. To be able to categorize these modalities we consider the tasks categorization of Tönnis et al. [2] as an initial point. They divide the interactive tasks in a car into three classes which we call input types: the **primary**, **secondary**, and **tertiary input types**. The primary input types are the controls used to control the vehicle itself – gas, brake, clutch, steering wheel. The secondary input types are mandatory controls such as control of turning signals, headlights, or mirror adjustment. The tertiary input types are the controls used to control entertainment and infotainment components such as the radio or a navigation device. The primary input types are irrelevant for our project and the secondary input types are only partly important as opposed to the tertiary input types which are in our main focus. Our project tries to simplify and, more importantly, unify driver input for the input devices most used after the primary driving input devices.

While abstracting away the functionality and location of component related controls of these inputs and thus unifying them we can also decrease electrical wiring complexity of the vehicle by removing knobs and switches which are then unnecessary. In this case abstraction means the control of all components with a central control interface in a central location in the cockpit instead of at each component location with dedicated controls. Vehicle manufactures can save quite a bit of money due to the decrease in parts required and the decrease in assembly complexity. This also has the positive side effect of decreasing impact points in the case of a crash.

The idea of the abstraction step mentioned above also aims to decrease the mental work load of the driver by placing the one input device still required in a central location, increasing the amount of time the driver can stay focused on driving. This increases the ubiquitous aspect of interacting with the vehicle – instead of controlling single systems, the driver must simply look at them to interact with them via the single input device. Apart from improving the perceived response time, we hope this will further allow a perceived standardization of input modalities across multiple vehicle brands even if they each have a unique interior. This in turn should make familiarization with new vehicle brands increasingly simple, lowering the hurdle for attaining familiarity with new vehicles. This could be a nice bonus for drivers that drive rental vehicles on a frequent basis.

1.2 Organization

This document is structured by starting with a look into related work in gaze and eye tracker research in chapter 2 as well as how this technology has been used in automotive scenarios so far.

Next we present our project in chapter 3. Our project goals and features are categorized in must-, should-, and could-haves which were inferred from the use cases, also included in the chapter.

Chapter 4 goes into detail with which hardware and software we developed the prototype. Here we also go into detail of how we structured our work in the allotted time frame given to the project.

The details to the implementation of the prototype can be found in chapter 5. We touch on the details of procuring our hardware and go into a bit more detail on how our multiple code segments are structured. This chapter closes with a note on our development environment.

As with any project there is always more that can be done on it. We expand on this in chapter 6. We also briefly introduce the alternative concepts that we were considering for our project.

Last we conclude our project by summing it up in chapter 7 with all the relevant steps and the results we worked up.

2

Gaze Tracking

Based on related work we will discuss the basics of eye and gaze tracking in this chapter, as well as its benefits and problems. Research most relevant to us concerning the automotive field is presented in section 2.4.

2.1 Eye Tracking

Eye Tracking is not a new approach to human computer interaction or a new technology in any way. Eye tracking has been incrementally developed over a long period of time until it reached its current state as we look back at the history of it [3].

In the early 1900s there were eye trackers using a contact lens where the gaze direction was shown by a piece of protruding wire. Not until 1980 was eye tracking used as a tool for investigating human computer interaction. In 2000 eye-controlled computer interfaces

were primarily used by people with disabilities to generate speech and to use computers. 10 years later the process of collecting and analyzing data with the use of eye tracking has become much more automatic, precise, and objective. Currently there is a huge interest in implementing eye tracking technology in computers, games, cars, operator controls, medical diagnostics equipment, and much more. But what does it mean to track one's eyes?

Tracking a user's eyes allows collecting and processing data on his eye movements and gaze points. This information is mainly used in user interface interaction with disabled people [4]. Eye and gaze data from eye trackers is received in real time and used as a way of selection and input for support computers. But an emerging field for eye trackers has been in marketing. They are used where it is important to check that product placement or websites work as intended for directing a user's attention. The eyes also give cues about the user's physical state such as fatigue and attentiveness [5].

There are several different ways to track eye movement and the gaze point via different technologies. They can be categorized into **relative** and **absolute** tracking. Relative eye tracking usually measures eye orientation and movements using a head mounted device. This could be done using electrooculography measuring the eyes' resting potential using electrodes placed near the eye [6, 7]. Another approach is using a contact lens which can be tracked in a electromagnetic field frame, causing discomfort but allowing high accuracy [7]. A third approach is using infrared light. Here the eye tracker emits infrared light to the user which is reflected from the users pupil and cornea. This reflection is recognized by the infrared cameras of the eye tracker in order to calculate the eyes' angles and gaze points.

This infrared approach can also be used in absolute eye tracking. Here the eye tracker is positioned remotely allowing the gaze point to be calculated in absolute coordinates. This approach allows an accuracy of about 0.5 degrees [8]. Recently several low-cost devices providing absolute eye tracking have emerged on the market allowing them to be utilized outside of specialized applications. Even the use of multiple eye trackers in an environment concurrently is now feasible cost-wise which was not the case just a few years ago. For information in a nutshell about how eye tracking works see [9].

2.2 Benefits/Advantages

In [10] Silbert and Jacob show with their two experiments that their technique for object selection with gaze tracking is faster than selection with a mouse. The first experiment was to select a circle from a three by four grid of circles as quickly as possible. The results showed that the performance of selecting a series of circle targets with eye gaze selection is 428 ms faster than with a mouse. The second experiments was to select a letter from a grid of letters. Each letter was placed in a circle ordered in a grid like in the first experiment. The subjects were told which letter they should select. The results here again showed that selection with gaze tracking was significantly faster than selection with a mouse by 338 ms.

Interaction by gaze reduces stress for hand and arm because these are used less – as opposed to direct touch interactions for example. It is also a form of remote control for gaze-based applications. Furthermore there is also a hygienic advantage as the interaction via eye tracking is (semi-)contactless. This might be important in paramedical sciences [5].

But what kinds of applications make sense for gaze-based interaction? There are applications where for example large displays or multiple monitor setups are used. It is difficult to interact with an application on a huge display when you make use of a mouse or direct touch input. For a mouse you need a very big area to slide it over or you scale up mouse sensitivity. Higher mouse sensitivity might be difficult to control when small objects should be selected – if the objects are scaled up to adapt the increase in interaction area is mitigated. Besides, interaction via mouse always needs a mouse pointer which must be visible at most times. With gaze interaction you can jump from one object to the other very quickly as Silbert and Jacob showed in [10].

Applications with gaze-based interaction interfaces in vehicles make sense because the driver should always leave his hands on the steering wheel during a ride. Later in section 2.4 we present some related examples for gaze-based interaction applications in automotive contexts. For a more in depth look at the advantages of using eye trackers in a vehicle see also section 3.1.

2.3 Problems

The first problem which occurs in systems with eye tracking is the calibration process. All eye trackers need a one-time calibration process before they can be used per setup. This is caused by the fact that every user has individual eye positions because of the head or body size, the distance to the eye trackers, and the individual range of eye or head movement. Additionally we have technical limitations such as the accuracy of eye trackers - today at $\pm$ 0.5 $^\circ$ - and human factors which appear because of the nature of human eye movements and contains factors such as saccades, eye fixations, and glimpses [5] [11]. These human factors depend on the anatomy of the eye and are therefore individual to every human. Hence this is a vicious circle because the low accuracy of the gaze leads to inaccurate calibration process and vice versa. In [5] Drewes suggests an idea to hide the calibration process from the user by presenting an introduction screen which is used to automatically calibrate the eye tracker in the background.

A big problem can be unconscious gazes – when the user quickly glances at something without intending an interaction. Differentiating between these glances and conscious interactive gazes can be non trivial.

An interesting problem that eye tracking has in general is that "*[there] is no natural way to indicate when to engage the input device[...]*"[11, p. 15]. This problem is known as the "Midas Touch Problem", which describes an unintentionally initiated action or selection when eyes are used to control or select user interface elements [12]. A variety of solutions for this exist, but finding a solution that works for any given interaction environment is not always easy. For example, utilizing a blink as a trigger forces the user to actively control their blinking and increases the mental load [13]. A collection of solutions to the Midas touch problem [14] is thus partially relevant to our work.

On the matter of vehicle systems there are a few more problems that appear. One problem that we foresee with the current eye tracking technology we will be using is that it is infrared based. A vehicle spends a considerable amount of time while driving under direct or indirect sunlight, which means that we have an external infrared light source

that disrupts a clean tracking of the eyes – the sun. While for our purpose this poses a lesser problem as we will work in a controlled environment as the project is not initially meant for real world usage, it is nonetheless to be kept in mind if further work is to be done. There is a range of options to counter this problem. There exist known instances of vehicles that have infrared reflective glass[1]. It is viable that such glass might suffice to enable infrared based eye trackers to work while driving. Alternatively other technologies for gaze tracking can be used, such as camera based eye trackers.

2.4 Automotive Eye Tracking

Eye tracking in the automotive sector is a highly topical issue, as for example in the *Advanced Driver Assistance with Tobii Eye Tracking* [15]. The automotive context is most relevant to our project. With the cost of eye trackers reducing there have emerged multiple research papers in the recent years. The following sections cover this related work.

2.4.1 Making Use of Drivers' Glances onto the Screen for Explicit Gaze-Based Interaction

In [16] Kern et al. introduce their approach of gaze-based interaction in combination with a button on the steering wheel. Gaze recognition requires means for providing feedback, so a feedback cursor is inaccurate and therefore highlighting of user interface elements is used as a feedback mechanism. We will also use this because gaze tracking does not provide any form of physical or auditive feedback. We chose visual feedback because we think auditive feedback might be too slow due to long audio playback times of component names or command confirmations in combination with gaze-based object selections. This could be annoying or distracting for the driver.

To avoid the back and forth between multiple items in a system without threshold Kern et al. use a spatial and temporal threshold before switching to the next item. The visual

[1] See for example: `http://www.volkswagen.co.uk/technology/glossary/infrared-reflective-glass` – accessed 2014-09-06.

feedback is only moved if 5 gaze points are registered at a new area. An interaction sequence is often interrupted as the driver should regularly look back at the street, so the last item is kept highlighted so that the driver can select it without looking back onto the screen. We will be using a similar approach in our project because we would not like to distract the driver by forcing him to always continuously look into a direction away from the road to be able to interact with the provided interface.

The **dwell time** part of Kern et al.'s approach requires the user to look at an item for a defined time period (about 150-250 ms) for selecting it. The button-pressed part of the approach is more feasible in a driving condition because the driver is not forced to look at an object longer than necessary. The whole approach of Kern et al. consisting of gaze tracking in combination with a button on the steering wheel makes interactions possible without compromising safety while the hands remain on the steering wheel. Nevertheless this solution is a little slower and more distracting then touch screens for untrained persons [16].

The main difference between our concept and their work is that we are not only going to replace the "touch" input on touchscreen interaction but generally almost every input modality in a vehicle cockpit with only one hardware button and a reasonable function mapping. These input modalities includes for example: pressing buttons, rotating knobs, rotating scrolling wheels, and adjusting directions.

2.4.2 Gaze-based Interaction on Multiple Displays in an Automotive Environment

In [17] Poitschke et al. present a concept in an automotive environment based on gaze interaction where multiple displays are used to select several user interface elements. Rapid advances in the field of in-car information systems (both driver assistance and driver information) contribute to an increasing mental workload of drivers originating from tertiary tasks [17]. The idea of Poitschke et al. is that scanning the environment is part of the natural orientation and task planning process of everyone. So eye movements and especially fixations can be used for selecting visible objects in the user's surroundings.

There are two main problems in the concept. One of them is the **Midas Touch** problem as a big challenge. The other is that input action must be interruptible for urgent traffic situations. Advantages are that the selection process can be accomplished very quickly and is easy to understand even for untrained users. Furthermore the driver can keep his hands on the steering wheel which is fundamental for optimal driving safety [17]. To realize the concept an additional channel for confirming an action is needed (e.g. speech input, hardware button).

Poitschke et al. used confirmation and control with barrel keys on the steering wheel, which can be pressed and also rotated. As mentioned in the previous section in our project we are planning on allowing mapping of almost every input modality in a vehicle cockpit to our single hardware button, not just one.

The results of Poitschke et al. are that there is a high cognitive load for gaze interaction which lead to high reaction times. The driver shows lower performance in keeping the vehicle centered on the lane. Additionally the road is not easily visible during gaze interaction and the concept is not considered much better than touch interaction [17]. Trained users perform better with gaze interaction compared to touch interaction. Untrained users tend to "stare" at interaction objects.

2.4.3 Driver Inattention Detection Based on Eye Gaze-Road Event Correlation

In [18] Fletcher and Zelinsky present their driver assistance system which tracks user gaze data while driving. The aim of the system is to correlate the driver's eye gaze with road events to determine his observations, e.g. to detect missed road events and warn him. It is unfortunately not possible to determine that road events are seen for certain by the driver due to the inability to detect whether the user has processed an event. The system detects that the driver looks at certain elements but it does not know if the driver has consciously detected it [18].

2.4.4 Infotainment Devices Control by Eye Gaze and Gesture Recognition Fusion

In [19] Nawaz et al. show a concept for controlling consumer devices by fusion of eye gaze and gesture recognition methodologies. Consumer devices are devices such as MP3 players or other daily life appliances. Selecting elements is done by focusing on them with eyes for a *dwell time* period and by confirming it via a head shake or nod. This approach has a good detection rate but also long delays due to the need to wait for the dwell time.

2.4.5 Using Eye-tracking to Support Interaction with Layered 3D Interfaces on Stereoscopic Displays

Alt et al. [20] investigate the concept of using eye trackers for 3D user interfaces. Therefor they compare methods for calculating the 3D gaze point in space as well as 2D and 3D performance. In their study they use layered screens where the user could select the layer using gaze. They found out that eye tracking with current technology on stereoscopic displays is possible with similar performance as on standard 2D screens. They also argue that for instance in an automotive scenario warnings, navigation cues, status, traffic information, and radio could be moved onto these different layers in a car heads-up-display for example.

2.4.6 Driver Distraction

In this section we consider related work to driver distraction factors which play a crucial role in automotive context, especially in combination with gaze tracking, where the driver has to look at the component he wants to interact with.

Horberry et al. [21] examined the effects of distraction upon driving performance for drivers in a study. Therefor they used two in-vehicle distracter tasks. The first one was operating the vehicle entertainment system and the second one conducting a simulated hands free mobile phone conversation. The drivers in their study had to drive in a

simple and in a complex road environment and were divided into three age groups. They recorded the driver performance and reactions to roadway hazards and the subjective workload under two different circumstances. The conclusion of their work is that control of the entertainment system has the greatest negative impact on driver performance, such as driving speed, and readiness to react to sudden hazards because of visual distraction. On the other hand they found that the hands free mobile phone conversation has less negative effect on safety then performing usual tasks (for example switching radio channel) whilst driving. So a visual distraction could end in a traffic crash and should be prevented.

Stutts et al. [22] analyzed in their study the role of driver distraction in traffic crashes. They identified eleven specific sources of distraction among distracted drivers and ranked them. At the top of the list is the distraction caused by outside persons, objects or events. As we focus mainly on tertiary input methods in car cockpits, these events are less relevant for us than the others. In second place, with 11.4 % of the cases, was adjusting the radio, cassette, or CD. Adjusting vehicle/climate controls was ranked at sixth place with 2.8 % of the cases and using/dialing cell phone was ranked at eighth place with 1.5 % of the cases. Thus they showed that adjusting the radio, cassette, or CD is one of the main causes for distraction-caused accidents. Using a phone while driving here seems to be less of a distraction to the driver.

But this can be disproved by Burns et al.'s work [23]. They investigated in their study how dangerous it is to drive and at the same time use a mobile phone, once with a hands free conversation and once with a hand-held device. Furthermore they related these driver performances with those of driving with alcohol impairment. Their results show that using a hand-held mobile phone significantly declines driver performance in matters of speed control and response time to hazards. Hands free was better than hand-held but nevertheless it detracts driving performance and thus driver attention more than driving with a legal blood alcohol level (UK legal limit of 80mg/100ml) [23].

3

System Concept

In this chapter we first describe the core idea of our human computer interaction concept of gaze tracking in section 3.1. Afterwards we define the realized concept based on this idea, which we call **Gaze Context Mapping** (GCM) in section 3.2. The GCM represents the interaction form between the driver and the components in the vehicle. Based on this concept we define our main project goals in section 3.3 which are divided into three sections: Must-Haves, Should-Haves, and Could-Haves. Furthermore we describe an exemplary approach in section 3.4 for automating gaze-based component controls in automotive context which we have implemented in our system: the Automatic Mirror Adjustment.

Furthermore we will take a look at the use cases for the Moteye system in section 3.5. We consider use cases in this context to be possible goals to be achieved with our system. First we will take an abstract look at the general use cases that form the basis

for any concrete application oriented use cases. Then we will list the clearly defined use cases we considered for our prototype. Based on these we will highlight the use cases we chose to give precedence for our prototype in section 3.6 with a short explanation as to the reasoning behind the decision.

3.1 Concept Idea

A car's cockpit today consists of many components with many more controls to be used. The user must learn which control provides which functionality to be able to interact with these components quickly while driving. When doing this interaction he has to find or check the required control with his eyes. Then he can refocus his gaze back onto the street. His hand leaves the steering wheel and moves to the button, eventually feeling its position, then select or adjust it (see figure 3.1). Finally he reverts his hand back to the steering wheel.

Figure 3.1: Hand off the wheel for physical interaction.[1]

This use case may only take a couple of seconds – however this is a lot of time to be distracted in a fast moving vehicle, covering hundreds of meters on a motorway in the worst case. The second fact is that one of his hands leaves the steering wheel, reducing safety as the reaction time increases as the driver is required to put his hand back onto the wheel for full control in the case that a fast reaction is required. Another fact is that the driver has to search through all the controls first to find the correct one if the driver is not familiar with the layout.

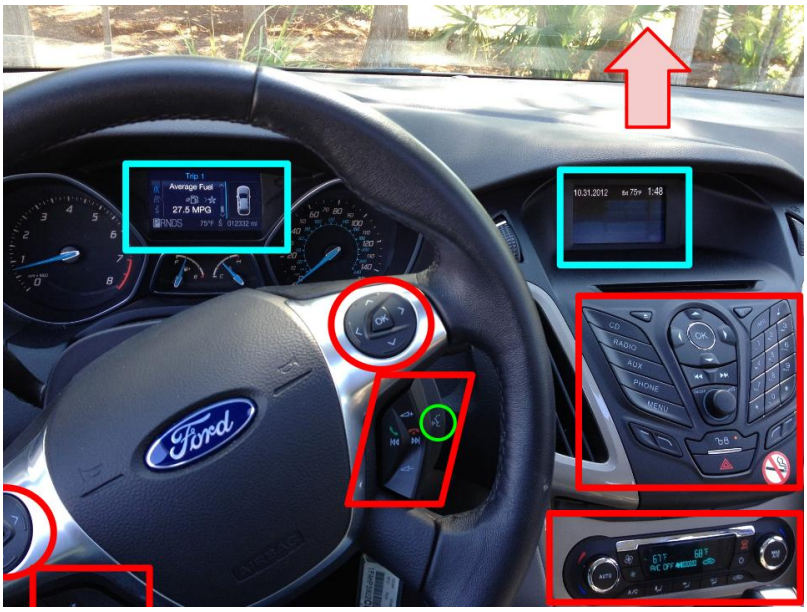

Figure 3.2: Markup of control elements and their locations in a car.[2]

Using these controls always needs a physical interaction such as for example pressing a button, scrolling a wheel, or turning a switch over several places in the cockpit, such as the middle console (see figure 3.2), tachometer, head-up-display, or even outside the vehicle.

These causes are some of our reasons for our concept of Gaze Context Mapping. The purpose of GCM is to provide a new, gaze-based interaction interface for the driver which

[1] Image source: `http://g.fastcompany.net/multisite_files/fastcompany/poster/2013/03/3007085-poster-1280-your-car-giant-smartphone-wheels.jpg`, retrieved on 18.02.2015

[2] Image source: `http://blog.andrewroback.com/wp-content/uploads/2012/11/ford_focus_markup.jpg`, retrieved on 19.02.2015

enables them to select components in a car with gazes and control all of them via one central hardware button directly on the steering wheel. We believe that GCM provides a better human computer interaction model than plain hardware interaction by combining gaze tracking with abstracted components interaction. We elaborate this in the section of Benefits and Challenges.

3.2 Gaze Context Mapping

The idea of GCM is simple: use physical components of a car cockpit as targets and map the user's gaze onto them. This provides the possibility to select these components with a short glance (see figure 3.3). In order to actively interact with them a single control button is placed on the steering wheel allowing adjustments to this component without taking one's hands off the wheel.

Figure 3.3: The idea of GCM.[3]

[3] Image source: `http://www.hdcarwallpapers.com/view/audi_r8_v10_interior-1440x900.html`, retrieved on 19.02.2015

Benefits and Challenges

One benefit is the reduction of controls in a vehicle as well as the adaption time for learning these. Besides finding any of the auxiliary controls during driving might be stressful or distractive. As we described in section 2.4.6 driver distraction is a general problem which must be prevented as best as possible. We handle this problem with a very simple approach.

GCM needs only very short glances ($\sim$100 ms) for component selection so that the driver can primarily keep his eyes on the road and visual distraction is greatly reduced. A completely visual distraction-free environment in a car cockpit can not be created with GCM but we believe that it reduces the time a driver takes to look off the street at a component and also the time of controlling such a component.

In the case of our proposal the driver can keep his hands on the steering wheel while interacting with components in the cockpit, which is also an important safety factor. Also GCM eventually reduces costs for manufacturers (less production/testing costs) as well as consumers (less waste costs) by removing electrical physical switches and improving crash safety that way.

Therefore a couple of challenges exist for realizing GCM. A challenge is to find suitable components in the car cockpit which work well with this concept and map them intuitively to our controls. Also the cockpit is relatively large and our eye trackers have a limited working angle which requires us to find optimal positions and use multiple of them to complete cover all interaction areas. Another challenge is to implement this concept in a way it does not increase mental load on the user or distract him and thus keep driving safety to a maximum. The **Midas Touch problem** is another challenge for selection of components which we counter with two different **selection modes** for components with one default mode for comparison:

Eye-Select Simple component selection type that only selects the component when the user is currently looking at it directly. If the user is not looking at a component it is immediately deselected. This mode is primarily for comparison purposes while development, however it is a viable

alternative when using the system when the vehicle is not moving. This mode does not counter the Midas touch problem as the eye is actively used to select the components (although no interaction is executed just by looking at the component).

Last-Select The last-select expands on the eye-select mode by bringing more automation into the interaction. It keeps the component last looked at selected, keeping it controllable via the joystick button even if the driver returns their gaze to the road. To select a different component the driver only needs to look at the new one. Thus the driver cannot only keep his hands on the steering wheel but also his eyes on the road. This selection mode especially makes sense for components which are often used over a longer period of time as it keeps the driver from having to look at the respective component continuously each time they want to interact with it, as opposed to the eye-select mode.

Button-Select This selection mode expands on last-select by requiring a confirming button press via the joystick before selecting a component. Thus we distinguish between glancing at components and targeted component fixations with the intention to control a component.

3.3 Project Goals

This section describes features and requirements to the system in order to work properly and be beneficial to a user interacting with it. For this project, the feature it must have are required for it to be considered a success. The features listed under the ‘Should-Haves’ and ‘Could-Haves’ sections are features we will additionally try to fulfill but which are not strictly required for the project.

General Goals

The goal of this project is to have a prototype system that proves the feasibility of our project idea. This means that the prototype will not be directly integrable into an existing vehicle; instead it will run in a lab setting with only the required hardware – the eye trackers and one or more computers for running the system with the possibility of attaching working component mock-ups. If time permits we would like to try integrating the system into the existing driving simulator. No real-world tests are planned, including any feasibility tests for actually integrating such a system into a real vehicle.

As the system is intended to run as a closed system, it will not offer up any privacy or security features. This greatly reduces the code complexity and allows us to keep the system generally valid – no adaption to any single vehicle brand must be performed. At the same time it should not require any sensitive information from its users anyway, so this should not be a problem.

What we will work on is to make the system as responsive as possible to allow us to verify our theory that such a system decreases driver inattention to the road and decreases mental load. While the system might not be completely real time, we aim to come as close as we can within the scope of the allotted time for this project. To truly make the system real time would also require a more complex development environment as it would require real time hardware, which we do not have readily available.

Another point that we would like to work towards is to make the system capable of increased modularity. This means for example that new components could be quickly and effortlessly added to the system to allow further work without having to rework the core implementation. Rapid iteration would thus also be easily available without increased amounts of unnecessary work.

System Structure

The system will consist of three main parts. Foremost are the eye trackers. We will make use of multiple eye trackers because every eye tracker has a defined interaction area where components can be placed in. Components beyond this area are not sensed by

an eye tracker. Thus it makes sense to use multiple eye trackers on different positions in the cockpit to make the many different components selectable. The eye trackers are placed in any configuration in front of a user so that they cover the work area in which the user can interact via our system. Input is initiated via the singular control unit on the steering wheel, which is our second part of the system. The two input modalities are connected in the system core, our third and most crucial component. Here the gaze information is combined with control inputs to detect when a user wants to initiate an action. This action is then passed on to any number of components required to successfully execute it. This could range from abstract actions such as controlling the entertainment volume to physical actions such as adjusting the side mirrors.

Vision

Our vision is to prove that the utilization of eye tracking in an automotive context is a viable alternative to current existing solutions for controlling mainly tertiary systems such as voice input. We hope to show that our system can increase driver attention to the road without decreasing the interaction possibilities, thus allowing a more natural and ubiquitous integration of auxiliary systems.

Scope Definition

This section highlights our definition of the project scope categorized by feature or capability into priority from required to nice-to-have.

Must-Haves

The following list describes the general features the system must fulfill to count as successfully implemented.

Cockpit Tracking

Multiple eye trackers are required to track the user's eyes at whichever position in the car cockpit he may be looking.

Merge Controls

Control elements of components are to be merged into a single control at the steering wheel.

Gaze Differentiation

Multiple components are to be differentiated via gaze.

Component Interaction

The user must have a minimum of one component to interact with. This should be the mirror component.

Vehicle Component Abstraction

Environment must be computer comprehensible and workable.

Should-Haves

This list describes the more advanced features that go beyond the basic functionality. These features should be implemented as they add to the system scope but are not strictly required for the system to work.

New Component Interaction

Multiple components must be supported and working. We proposed to implement a radio and hands free calling component.

Improve Usability

Lower system usage hurdles such as calibrating eye trackers and configuring the system using a guided wizard.

Could-Haves

These listed features significantly broaden the scope of the project but require the previous features as groundwork. While they would be nice to have, the scope is probably beyond and above what we can offer in term of time and resources.

Automatic Mirror Adjustment

Extend the mirror adjustment that it can use the user's eye positions to continuously and automatically adjust the mirror correctly.

Virtual Gaze Targets

Allow virtual components to work with the system, a requirement for example for interacting with digital components or objects not part of the static vehicle fixture.

External Gaze Target Component

Implement a component that can use external gaze targets for interaction, for example a component that can read road signs and offer user interaction with them.

3.4 Automatic Mirror Adjustment

An additional feature of our concept is the Automatic Mirror Adjustment (AMA) mentioned in the ‘Could-Haves’ section. As mirrors usually are adjusted only once before starting one’s drive, they may not be ideally suited for every possible situation and one may miss important angles. But also while driving a better view may sometimes be required when overtaking or parking.

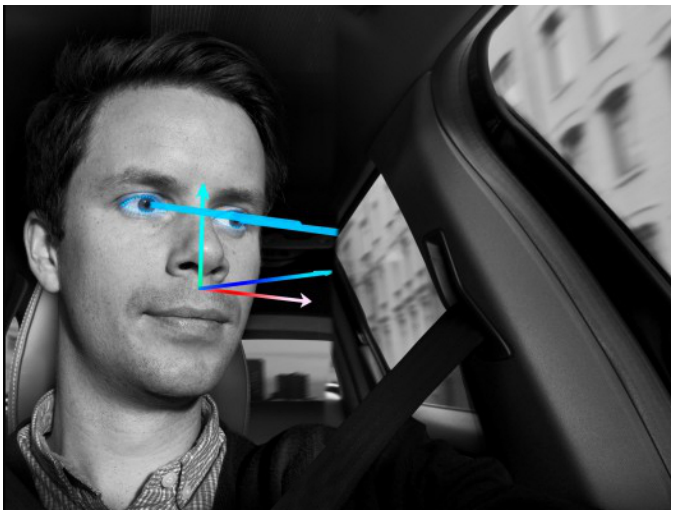

Figure 3.4: Exemplary representation of automatic gaze tracking.[4]

The idea of AMA is to track the user’s head and eyes position in order to calculate the best possible mirror angle for him (see figure 3.4). So the mirror angle follows the user’s head movement at all times until he stops it manually by pressing the joystick button once.

3.5 Use Cases

Generally, there are three actors in GCM. The **user**, usually the driver of the car; the **system** running in the background evaluating the user's gaze and interactions; and the car's **components** to be controlled.

3.5.1 Basic Use Cases

Basic use cases define access to the system and common operations. These include preference and maintenance tasks required for the system to function.

Note that one possible category of use cases is missing: this category contains all use cases required for the setup of the system. This includes the installation of the system – either into a vehicle or the prototype setup – and the registration of the components, including setting up the information connections between all the system parts. We do not list those on purpose as we focus on proving the system can work.

User Setup And Calibration

Each user's eyes are different and the user may have glasses or contact lenses. Thus to provide accurate tracking a calibration process is required the first time the user uses the system. Hereby the user sits in the car and has to follow a calibration dialog.

Selecting a User Profile

A User known to the system must select his profile to allow the system to load the corresponding calibration data. To reduce overhead during repeated use, the profile should be persistently savable and reusable across sessions.

Selecting a Component

In this basic use case the user looks at a component which is registered by the system. The component should somehow give feedback of the successful selection. Initially this will be via system debug options for developer purposes.

[4] Image source: `http://www.gizmag.com/volvo-automated-driver-monitoring/31257/pictures#3`, retrieved on 05.02.2015

Adjusting a Component

When having a component selected the user may operate a steering wheel control to manipulate the component. This should provide some form of feedback – simply executing the command may result in feedback such as changing a visible or audible property. Otherwise a virtual feedback mechanism should be triggered in a none distracting way for the driver.

3.5.2 Specific Use Cases

The following specific use cases describe interaction with different car components using GCM. These were collected by looking at existing vehicles and our knowledge of existing in-vehicle interface possibilities.

Air Conditioning Control

Ventilation, climate, and heating can be controlled relatively easily. Looking at a vent will allow the system to adjust its orientation if the vent can be electronically controlled. This would however fully utilize our control element, disallowing direct climate control. The alternative is to map airspeed to the up and down directions and using the left and right directions to adjust the temperature. Based on anecdotal evidence we believe that the alternative would be the more used option of the two.

Seat Control

Adjusting the seat could be possible too, but difficulties arise where to gaze at to activate the control. The gaze target may have to be a mock-up within the field of view of the driver. This feature might not be all that interesting as future vehicles will most likely adjust the seat automatically. Furthermore the seat should be adjusted before beginning a drive, so any positive effects of using our system are mitigated anyway.

Seat Heating

For those cars that have seat heating, adjusting the temperature of the heating could be implemented by looking at an icon or notification light for it. As the controls for seat heating are often placed relatively far away from the vehicle control functions, this feature could decrease interaction times and reduction of focus on the road

significantly. As a mocked gaze target is required here this could also serve as a feedback display at the same time showing the heating level.

Mirror Adjustment

A feature that would be well received could be adjusting the side and rear view mirrors via gaze control. This could possibly work even better than the usual method via extra control pad as the user has to look at the mirrors anyway to adjust them. This is however also a task that should usually be done before beginning a drive, so our system may not be overly relevant for this task.

Window control

Opening a window might also be doable via gaze. Looking at a window will allow the user to adjust the window opening via the up and down control. Difficulties arise from the fact that the target is the window – the user will regularly glance in its direction, possibly hindering clean target detection. Especially when utilizing virtual or external gaze targets as proposed in section 3.3 this would lead to targets where the system cannot differentiate between them.

Window Defrost

As the rear window usually cannot be opened, the gaze selection and control can instead be used to activate the defrosting heating in the window. It is however not very comfortable to turn the head around fully to look at it. Therefore a better solution might be to add the defroster functionality to the rear view mirror. This would however again possibly lead to a double mapping if the rear view mirror can also be adjusted via our system.

Control Convertible

Starting the opening and closing of a convertible car could also be implemented via gaze control. As most vehicles must stand still to move the roof, the advantage of having a hands-free approach would be limited however.

Sliding Roof

If the car has a solid roof with a sliding window in it, controlling the position of said window can also be done with our system. The sliding roof is unlikely to serve as a valid target for external or virtual targets, allowing this function to be easily mapped.

Windscreen Wiper

Activating the windscreen wiper is not quite a suitable task for the gaze control due to the multiple speed settings and the overlap with the driver view of the road. A representative icon would be required to be looked at. Again the mapping of overlapping targets could prove to be very difficult.

Light Systems

An easy use case for our system would also be controlling the interior lights via it. Looking at a light and pressing the control would switch its state. Controlling the external lights might also be an option, but lacks the direct interaction as the driver can not look at the lights "through" the vehicle. Here too an icon or status light might be used instead, comparable to the seat adjustment and windscreen wiper control.

Entertainment

A comparatively major distraction factor is control of the radio by the driver while driving. The two most important controls can be easily mapped to our system: volume to the up and down controls, and either switching tracks or stations to the left and right controls, respectively for playing local audio or radio stations. The driver would then only have to physically interact with the radio to turn it on and off, although depending on the control element a switch for this might be doable.

Hands-free System

If the car has a hands free system for calling, control of volume would be easy to implement by looking at the phone, comparable to the entertainment use case. Alternatively a phone's contacts may be browsed using up and down controls. Starting, accepting and closing a call could be mapped to left and right for instance.

Cruise Control

Controlling the cruise control could also be done via gaze. As gaze target the speedometer could be used; or alternatively just the front windscreen, allowing the driver to remain fully focused on the road. However this would be a usage of the system that touches the primary vehicle interfaces of driving. This should only be done if the safety of the vehicle is unlikely to be compromised if the system misinterprets a gaze.

Automatic Mirror Adjustment

Extending section 3.5.2 a recurring automatic mirror adjustment could be implemented. Hereby the mirrors are adjusted for an optimal field of view depending on the drivers size, head position and looking direction. Notably this uses our system as a service, not as an interaction device as it would simply read the user's head position and continuously adjust the mirrors accordingly.

3.6 Targeted Use Cases

Due to the time scope of the project and the limited size of the workforce for implementing it, we have decided to focus on a very limited number of use cases for the initial prototype. We remain hopeful however that our system will prove easy enough to expand so that further expanding work can easily build on our work.

Our initial implementation will be adjusting the side view mirrors as seen in section 3.5.2. Our primary reason for this is that connecting a mirror to the system offers significant feedback and should prove the feasibility of attaching non-software components to the system. Furthermore the hardware required for this is already readily available at the university, so practicality also played a role. Finally this will enable us to continue on to the automatic mirror adjustment should we have enough time as described in section 3.5.2.

Apart from that we plan to prove the modularity by adding a subset of the entertainment system to Moteye, specifically the radio control – section 3.5.2. Volume and switching the station or tracks should be relatively easy to implement and allow interaction with a non-visual interface.

Building on that we also intend to implement the hands free calling component as seen in section 3.5.2 which connects to a smartphone and allows the easy usage of volume and declining or accepting a call. As this might require an Android application to work it might require more time than initially planned – we hope to base the functionality on existing connectivity work using Bluetooth or Wifi.

4

Project Foundation

This chapter defines and reasons about the hardware and software technologies that were used to implement the Moteye project. We also show our project plan to which we tried to adhere over the course of the project.

4.1 Technologies

This section describes all the requirements to physical and virtual resources that are required in order to develop the project. Hereby we differentiate between the hardware and software components.

4.1.1 Hardware

First as Moteye is an in-car system there is a need for a physical representation of this environment. As the demand for a real car or parts of it exceed the project limits we use a driving simulator for tests and evaluation. The driving simulator, similar to that one in figure 4.1, is available to us and consists of three monitors, a seat, steering wheel, pedals, gear lever, and has force feedback support.

Figure 4.1: Picture of our driving simulator.

Eye Tracker

The central hardware component of Moteye are the eye trackers. Currently the eye tracker developers dominating the market are The Eye Tribe and Tobii. Therefore we took three different eye tracker models into consideration. Each of the eye trackers implements an absolute position approach using infrared light reflections as described in section 2.1. Coordinates delivered from this tracking method span a 2D plane which

usually is mapped to a monitor area. Figure 4.2 shows the low-cost Eye Tribe Tracker which is declared by *theeyetribe.com* [1] as:

> *"[...]the world's smallest eye tracker, the first to use USB 3.0 and the only alternative below the magical price point of $100."*

Figure 4.2: The Eye Tribe Tracker.[2]

Figure 4.3 shows the Tobii EyeX Controller which is a low cost end consumer targeted eye tracking device. It is distributed by Steelseries as Sentry which advertises the device as gaming equipment.

Figure 4.3: Picture of the Tobii EyeX Controller.[3]

In figure 4.4 we see the Tobii Rex which is a product aimed at developers of embedded solutions. Thus it is more expensive and only available through Tobii itself.

A comparison of these three eye trackers is seen in table 4.1.

[1] `https://theeyetribe.com/products/`, retrieved on 03.03.2015
[2] Image source: `http://www.eyeproof.net/p/img/bg/box.jpg`, retrieved on 13.02.2015
[3] Image source: `http://images.gizmag.com/hero/tobii-eyex-games.jpg`, retrieved on 13.02.2015

Figure 4.4: Picture of the Tobii Rex.[4]

Feature	The Eye Tribe Tracker	Tobii EyeX Controller	Tobii Rex
Price	75 Euro	139 Euro	595 Euro
Screen Size	24"	24"	27"
Sampling Rate	30 & 60Hz		30Hz
Accuracy	0.5–1 °	0.5–1 °	0.5–1 °
Operating Range	45–75cm	45–80cm	40–90cm
Cameras	unknown	1	2
Emitters	2	3	2
Dimensions	20x1.9x1.9cm	32x2x1.5cm	18.4x2.6x2cm
Weight	70g	91g	
Connection	USB 3.0	USB 3.0	USB 2.0

Table 4.1: Comparison of the specifications of the various eye trackers that we considered for our project.

Control Button

In order to implement GCM we also need a way to control our gaze-selected components. This is to be done using a control on the steering wheel. There are multiple options to do this. We need a way to make adjustments, possibly on two axes and an accepting button. In order to reduce the amount of controls a single control element would work best combining both. A common practice in todays cars such as Audi's is a rotary knob with a pushbutton on top. However there are more possibilities listed below:

- rotary knob with push button (1 axis)
- Joystick with pushbutton (2+ axes)
- directional pad (2 axes)
- directional pad with center button (2 axes)

[4] Image source: `http://www.tobii.com/ImageVaultFiles/id_2723/cf_2/st_edited/QT7O7XXeHlmpknBl9J15.jpg`, retrieved on 13.02.2015

- roller which can be pushed in (1 axis)

Ideal for us would be the directional pad or a joystick as we could use both axes directly: adjust volume and skip tracks for a radio and left/right top/down mirror adjustments.

Joystick with Push-Functionality

One possibility for the joystick are the products from sparkfun which we make use of. They provide a simple, easy to integrate joystick with a breakout board. The control of this joystick can be implemented over an Arduino.

Figure 4.5: Analog thumb joystick on breakout board[5].

Left Side Mirror

To approximate a real use case we need real hardware modules from the automotive context. For the mirror adjustment use case we organized a real left side mirror of a car with a 12 V servomotor which is able to rotate both in horizontal and vertical direction.

Arduino Uno & Motor Shield R3

To control the gaze-selected elements we make use of the joystick. But this joystick can only provide analog values of the inherent axis and has no standard USB port to simply

[5]Image source: `http://www.hobbytronics.co.uk/image/cache/data/sparkfun/thumb-joystick-breakout-board-2-500x500.jpg`, retrieved on 11.12.2014

Figure 4.6: A real left side mirror of a car used for the prototype.

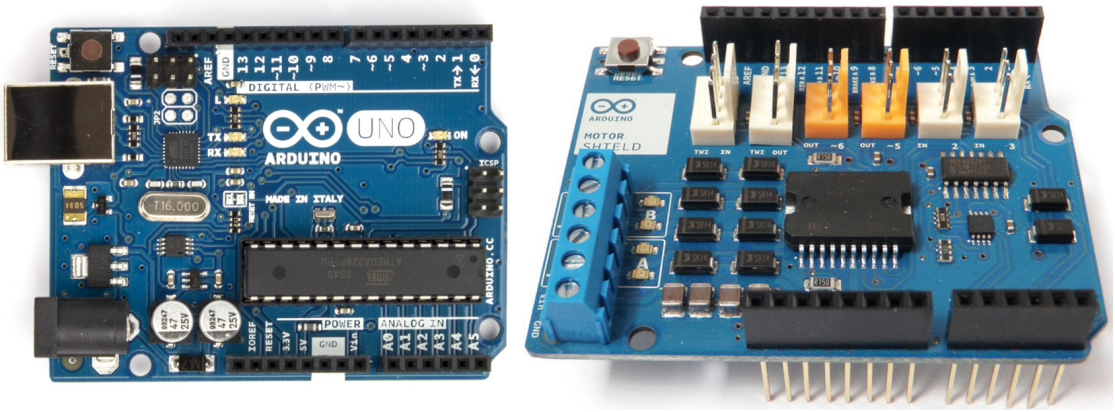

Figure 4.7: Arduino Uno R3[6](left) and Motor Shield R3[7](right) we used for the mirror and joystick control.

connect it to the computer. Therefore we make use of a micro-controller, the Arduino Uno, which is directly connected to the computer via USB. Furthermore we also need the Arduino, more precisely a motor shield, to control the servomotor in the mirror. The Arduino Uno itself provides only pins to supply 5 V output, however 12 V is necessary to

[7]Image source: `http://arduino.cc/en/uploads/Main/ArduinoUno_R3_Front.jpg`, retrieved on 06.02.2015

[7]Image source: `http://cdn.instructables.com/FES/1UE5/H05NSWPJ/FES1UE5H05NSWPJ.LARGE.jpg`, retrieved on 06.02.2015

drive the servomotor of the mirror. The screw-terminals of the motor shield produce 9 V output at maximum, this is nevertheless enough to drive the servomotor, but with some tolerable deficit on rotation time.

4.1.2 Software

TheEyeTribe SDK [24] is developed for the TheEyeTribe Tracker and allows an abstracted development using multiple languages and direct access via a JSON Tracker API. These are some of its features:

- C++, C#, and Java APIs
- Communication based on JSON
- Multiple trackers supported (8)
- Mouse gaze emulation for development
- Windows, Linux, Android

The EyeX Engine is the newest high level framework developed by Tobii for their trackers. It currently has the following features related to our possible use case:

- Easy high level APIs using C/C++, Unity, .Net
- Only one tracker support
- Hardware configuration
- Coordinate mapping
- Data filtering
- User profiles and calibration
- Windows support

The Gaze SDK is another, older existing framework by Tobii. It features the following:

- Low level C/C++, .Net APIs
- Allows dynamic calibration
- Multiple trackers supported (7)
- Windows, Linux, Android

4.1.3 Assessment

For our concept we chose the Tobii Rex eye tracker because this one has the biggest virtual screen size and therefore covers a larger visual area to work with. This is useful for placing many components in the same virtual scope. Furthermore the operating range is bigger than it is for the other two eye trackers. Thus the eye tracker is more flexible concerning its placement. Moreover the dimensions of the Tobii Rex eye tracker are the smallest ones which is quite utile because placing it does not require much room and it is easier to mount it on the driving simulator elements. In addition the Tobii Rex eye tracker has two cameras to sense IR-reflections which may be more reliable than just one camera. For more information about how Tobii eye trackers work see [9].

The EyeTribeSDK doesn't allow us to go as much in depth of configuring the eye tracker as for instance the GazeSDK does. The EyeX Engine is good for desktop application usage and allows developers to create such application on a higher abstraction level. However it is still in relatively early development at the time this project is being done, which limits functionality. The Gaze SDK is a good framework for non desktop systems and "full" control over eye trackers. However more work is required to be done to get comparable features. The great benefit of the Gaze SDK is that we are not limited by features of the higher level engine. This is why we would work with this framework.

As a control mechanism we went for the already mentioned analog thumb joystick on the breakout board with push button functionality which provides us with plenty of input options that can be mapped to almost every components functionalities.

The Arduino is simple and well suited for controlling the joystick and our basic use case of controlling the mirror.

4.2 Project Plan

The project plan is layout out for phase two of the project, the implementation, which is to be completed within one term. It shows when which milestones were due and describes what we intended to do within them.

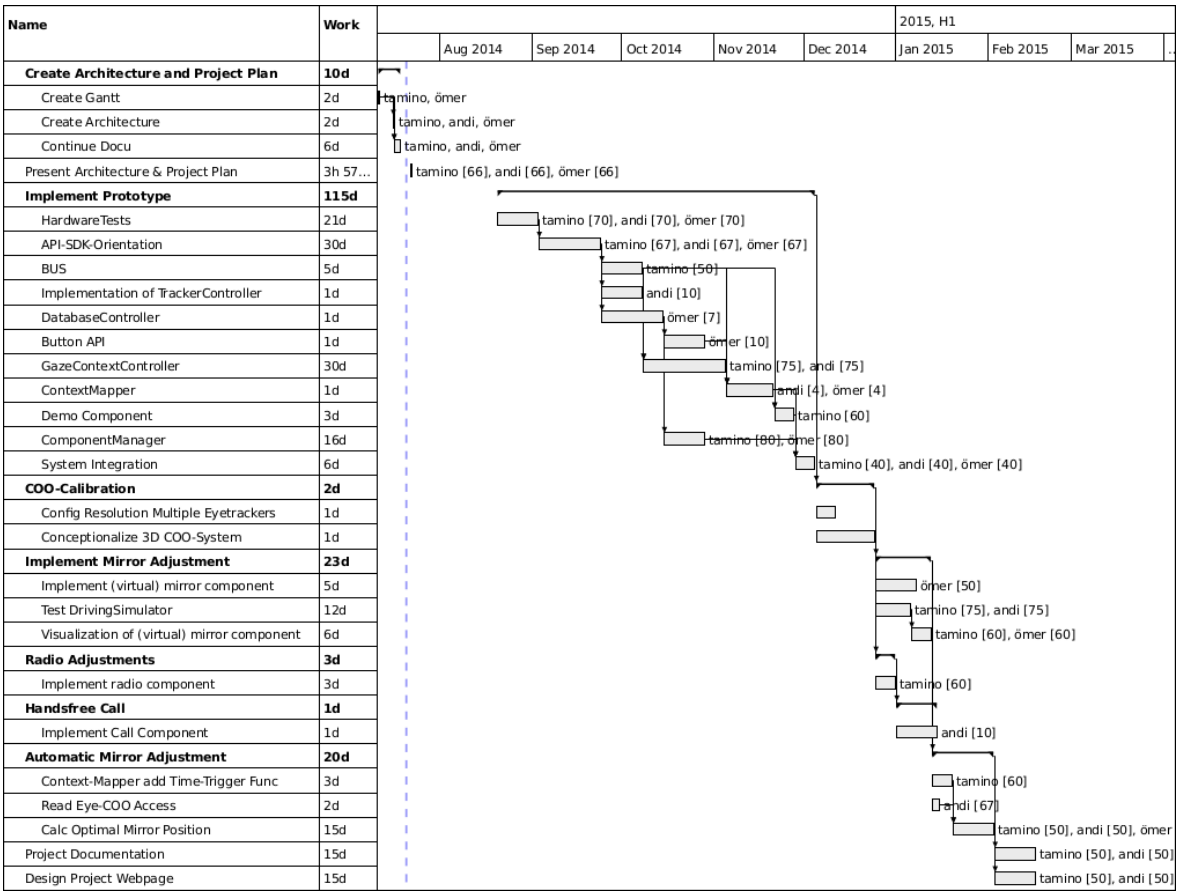

Figure 4.8: Our project plan in advance.

Figure 4.8 shows our proposed project plan for this project. The time span fits pretty good into our available time including the proposed could-haves. In case of larger stumbling blocks, we had be ability to cut away could-haves as required to remain within the allotted time. If the implementation would have taken significantly longer than planned we could also start cutting away should-haves, although we would have tried to get them done in any way possible.

Note that the work column is not very accurate due to some missing and vague entries. This mainly comes from the fact that we could not accurately estimate our work as the project introduces us to quite a few elements that were new to us personally.

Milestone 1: GCM Prototype Implementation

The first milestone was our biggest one. At its end, a working prototype was to be completed. It also includes time to get used to the hardware such as the eye trackers and to familiarize ourselves with their SDKs. Tasks in this milestone could be developed in parallel, as modules and interfaces are well defined. At the end of the development of all the internal modules a system integration was done concluding the milestone of a system working with basic functionality. Concretely the following tasks were completed:

- Channel communication
- Configurator
- Eye tracker control
- Eye tracker data processing
- Simple demo component
- Event logic
- Database
- Virtual button

Milestone 2: Mirror Adjustment

The next step was to add actual components to be controlled. This required us to find a way of mapping components in the 3D space to our 2D tracking planes. Multiple trackers were required here which were supposed to work with the existing prototype. The mirror component itself and the physical joystick button had to be realized using a real mirror and electrical motors. This required us to read into Arduino usage and build a working control system using the Arduino, motor shield, and joystick boards as well as the mirror.

- Multiple eye tracker support
- Confidence logic for trackers
- Debugging GUIs for trackers
- Ardunio sketch
- Java-Arduino control
- Mirror and joystick value tuning

- Realize component mapping

Milestone 3: Multiple Components

In this milestone more components had to be created fulfilling our should-have criteria. The project tasks of *Radio Adjustment* and *Handsfree Call* were thus implemented here. They were meant to prove the system beyond the initial must-have in terms of its capabilities and flexibility. As these components use existing interfaces they were easily integrated into Moteye and only their actual functionality required time to realize. Also in order to focus on usability of the system the wizard was implemented here guiding a user through the system and calibration setup.

- Implement and test filters to improve eye tracker data
- Hands free call component as Android application
- Radio component
- Configuration wizard component

Milestone 4: Automatic Mirror Adjustment

This milestone is our could-have and was planned to be tailored or extended to fit our progress. It focused on our AMA concept and should allow mirror control by head movement. Furthermore our prototype was installed on top of the driving simulator to serve as a more realistic testing environment. This milestone was meant to polish the system as much as possible for the final demo presentation.

- Automatic mirror control
- Extension of event processing
- Tests, debugging and fixes
- Building a setup for integration into driving simulator
- Refining the distributed component system to be used with the driving simulator

Milestone 5: Documentation

The project documentation time allotted at the end is meant for the general completion of the documentation, not the primary writing of it. We planned on continuously extending the documentation as events happen to give an accurate representation of the project. The project website and video are however done in a block at the end with the goal of simply representing the achieved project scope.

5

Implementation

Moteye consists of several hardware elements and software modules. The following sections take a closer look at the inner workings that make up the hardware and software. Also encountered challenges and our solutions to them are explained.

5.1 Hardware

The hardware part of Moteye includes the eye trackers, left side mirror, the joystick button, and the driving simulator. Mirror and joystick are powered by an Arduino board.

In order to test our prototype we built a mounting installation holding the mirror, joystick, and eye trackers in place (see figure 5.1 and figure 5.2). Therefore we measured out a real car's dimensions and placed all hardware parts in a plausible way (see figure 5.4). The eye trackers had to be positioned to allow tracking of the left side mirror and the

center console, thus one was placed where the driver's door would be and a second one near the center console. Also we propose to add a third tracker above or behind the steering wheel to cover future use cases tracking components outside or in a HUD.

The Arduino board, concretely an Arduino Uno with a Motor Shield R3 is mounted and placed in our *Moteye magic box* (see figure 5.3). The motor shield is used to power the mirror via an external 12V power supply. Also the joystick button is connected to the Arduino. Via USB the Arduino is connected to a computer running a Java program parsing joystick input and directing commands to the mirror via the serial port.

Figure 5.1: Picture of the Moteye hardware setup with eye tracker as well as the left side mirror and radio component.

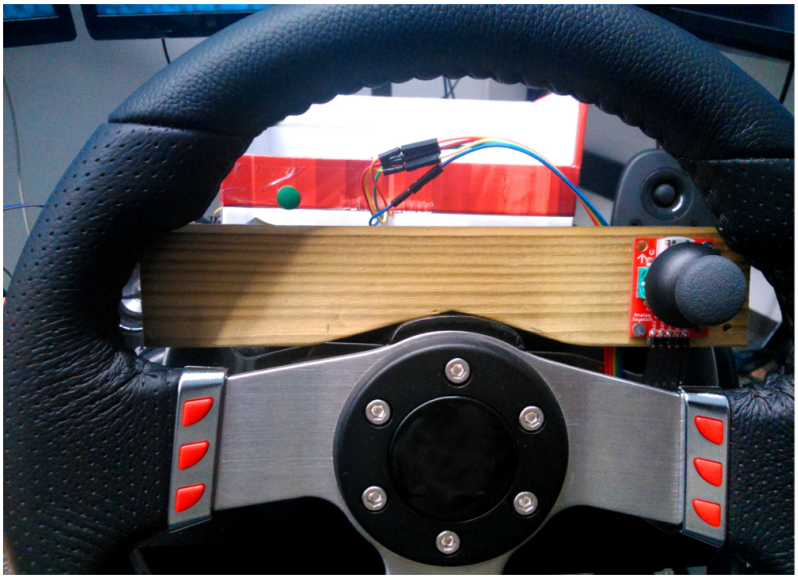

Figure 5.2: Picture of the Moteye joystick button on the steering wheel fixed with a piece of wood.

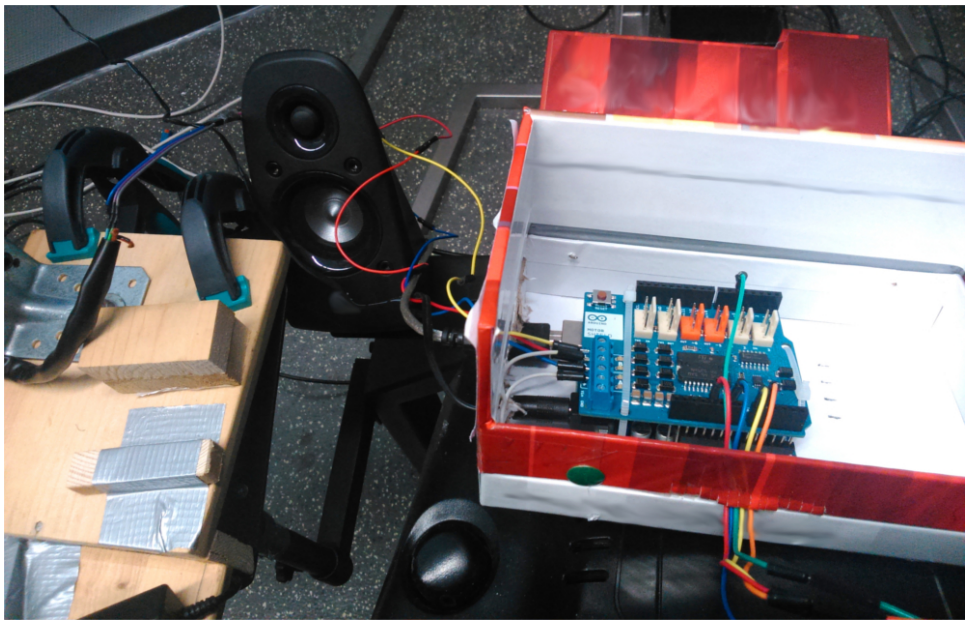

Figure 5.3: Picture of the Moteye magic box with Arduino and the motorshield in it.

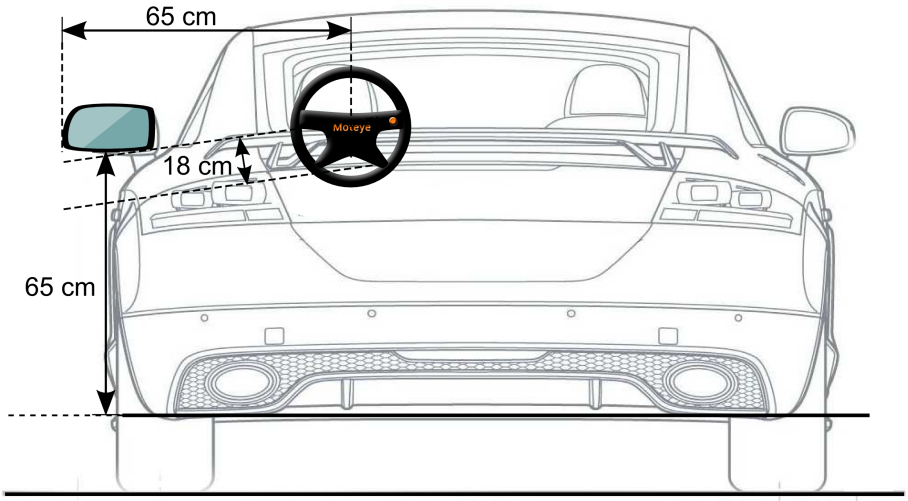

Figure 5.4: Draft of car, mirror, and steering wheel distances.[1]

5.2 Software Overview

Moteye uses a distributed system architecture of nine applications. An overview of these is given in figure 5.5. Here on the left side the modules are shown which create input to the system, namely the eye trackers as well as the joystick control. Gaze data is received and preprocessed by the Trackteye module written in C# and then handed over to the Java part of this module. There it is processed again and finally sent to the Coreteye module. Joystick input comes in via the serial port of the Arduino, is parsed, and then also directed to Coreteye. Coreteye fuses this data runs logic and dispatches events to the components. The components react to incoming events. All this is described in the upcoming sections in detail.

[1] Based on image from: `http://cartype.com/pics/9180/full/audi_tt-rs_coupe_f-r_12.jpg`, retrieved on 11.03.2015

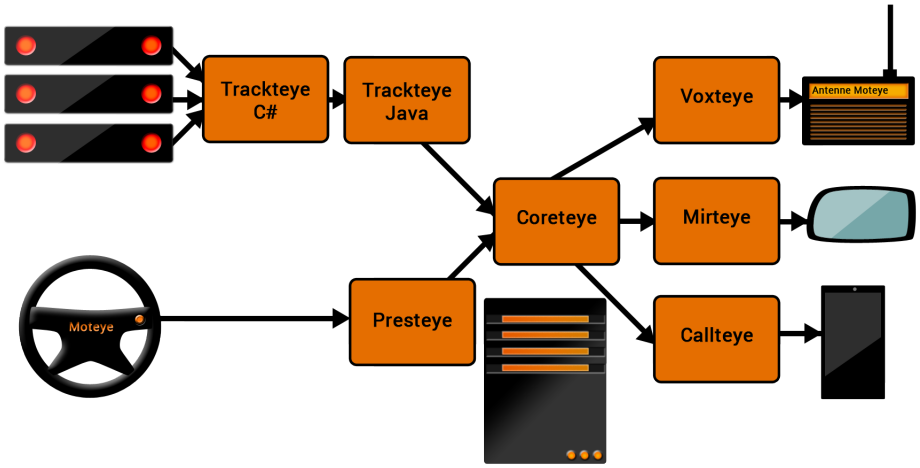

Figure 5.5: Overview of the Moteye Architecture

5.3 Trackteye

The C# part of Trackteye controls the eye trackers. The system supports up to seven eye trackers per controlling computer which can be placed anywhere in the car cockpit. This is a hard limit given by Tobii, however the system is flexible enough to allow multiple computers controlling the trackers and therefore can be extended at will. Eye trackers are controlled via the Tobii Gaze SDK. This is a low level SDK allowing multiple trackers to be connected. Also we receive raw tracking data. For each connected eye tracker at the USB controller an instance of the Tracker class is constructed. Each Tracker runs in a thread in order to allow parallel processing. On program start the eye tracker setup is run and a debugging GUI started, as seen in figure 5.6. Here a listener is constructed which receives GazeData events from the physical eye tracker in an independent thread. Then the eye trackers are connected and the tracking started. When GazeData is received it is serialized in a string and then send via TCP socket to the Java part. The TCP socket is connected at the first attempt of sending a message. When the setup is completed the Tracker instance registers itself by sending its ID.

In Java the main class is the GazeContextController (GCC). On construction it opens a listening port for Trackers to connect to and tries connecting to Coreteye (see also section 5.4) to send out data after processing. For each tracker connected an instance

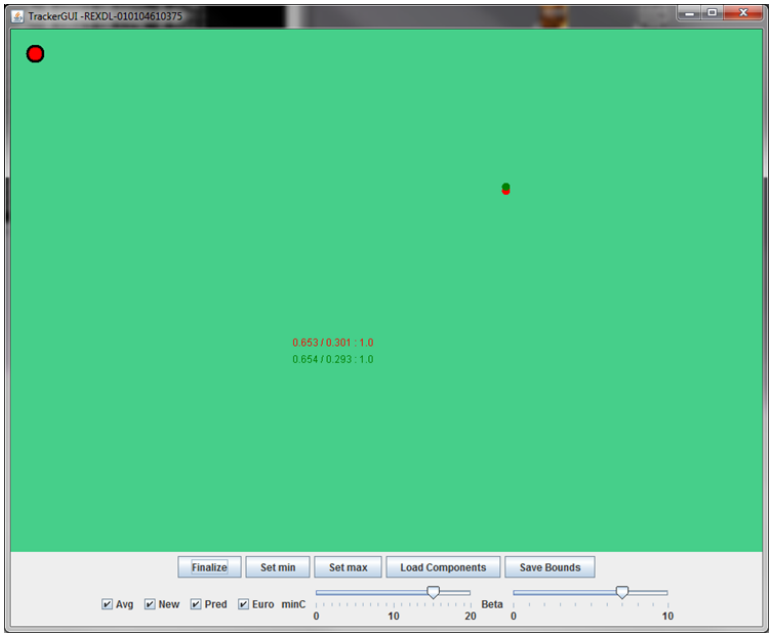

Figure 5.6: GUI for Trackteye debugging showing gaze data, coordinates, filters, and options to calibrate the tracker.

of a Tracker class is constructed. A list of all connected trackers is stored in the GCC. See also figure 5.7 for a class diagram of Trackteye to follow along.

On construction a Tracker opens the debug GUI and registers itself on the TCP connection to listen for messages. There are two types of messages received: either the identification (ID) or the serialized GazeData mentioned before. The ID is used as identification of this very instance. The serialized GazeData is parsed to a GazeData object and added to a list holding the last n frames of GazeData, an implementation of a sliding window buffer. The tracking state of an GazeData frame is hereby mapped to a confidence value where both eyes tracked means full confidence and no eyes tracked none.

When a connection to Coreteye is given, the GCC repeatedly queries all connected Trackers in order to find out which one is being looked at and how confident its data is. Several filters were implemented to process GazeData. A simple moving average window, the 1€ filter [25], a prediction based on the moving average window, and unfiltered. Each Tracker is queried for its filtered data, then this data is normalized from

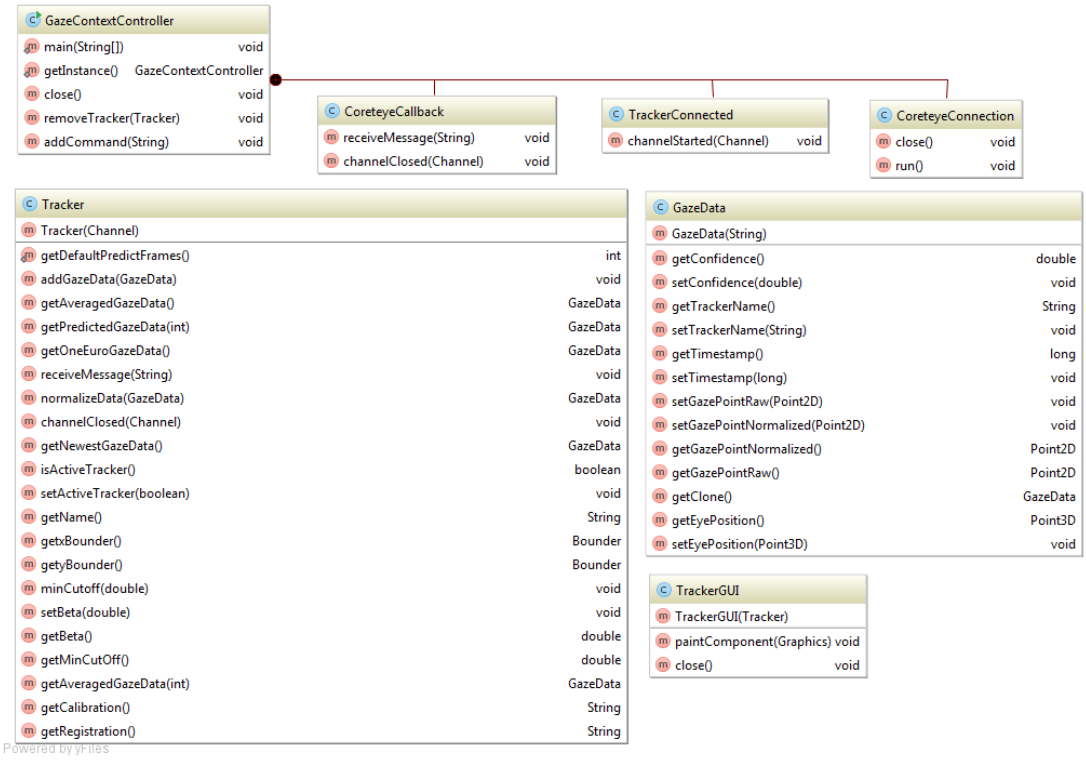

Figure 5.7: Class diagram of Trackteye

the 2D tracker coordinate system to our normalized system. Also based on how far from the calibrated center a gaze point is the confidence is penalized. The filtered GazeData is then compared between all Trackers and, if it is sufficiently confident, sent as best data. Its originating eye tracker is designated as active. In the next step it is checked whether or not this GazeData was a fixation or comes from a saccade. This is done by looking at the average data of the last x frames and comparing euclidean distances. In our case the last x frames is defined by a dwell time larger than 100 ms as defined by Salvucci [26]. Eventually the GazeData is written out to Coreteye.

Implementing Trackteye we encountered multiple challenges and problems. These are presented with out solutions in the following list.

Pattern Overlap

As the trackers are infrared based using multiple of them may result in their patterns to overlap and thus be inaccurate. This can only be solved by proper positioning to minimize this effect by occluding them from each other.

Unfiltered Data

Using the Gaze SDK we get unfiltered data with many frames missing information or with the eye tracker not confident enough about the coordinates. Here we implemented and tested multiple filters. Using unfiltered data directly, doing a moving window average over the past 200ms, using the 1€ filter, and linear prediction of the next few frames based on previous frames. The 1€ filter happened to work a little worse than the moving average which showed the best results in our case. This may however still be further improved by adjusting parameters or the window sizes.

Determining Active Tracker

As gaze data may be received by multiple trackers at the same time, we had to find a way to distinguish which one the user currently looks at and set that one to being active. Here we used the number of eyes tracked and the gaze position within the calibrated tracking plane to build a confidence mean. This again is calculated per tracker for each frame and the one with the best confidence is defined as the active eye tracker.

Parallax Effect

Before implementation we believed that we would have to account for a parallax

effect due to the mapping of 3D component coordinates to the eye tracker's 2D gaze tracking plane. However it turned out that the field of view of the trackers is too narrow and too restricted in our case to cause significant parallax effects, thus we did not have to solve this problem.

Tracker Standby

Whenever an eye tracker looses eye contact for a few seconds it goes into standby or energy saving mode, only casually checking for reinitialized eye contact. Recovering from this mode takes up to 2 seconds for the eye tracker we used. Thus quick glances may be missed increasing interaction times considerably. A solution would be a different firmware keeping the tracker in full operation at all times. This was however beyond the scope of our project.

5.4 Coreteye

Coreteye is the central module for the Moteye system. It handles and processes event fusion of trackers and the input joystick, manages the user profiles, communication to the components, selection modes, and overall configuration. To make developing and controlling it easier we implemented a GUI for the basic operations. This allows us to rapidly change configurations on the fly if required and control user profiles independent of Wizteye. See figure 5.8 for a screenshot of the GUI.

Now follows a brief overview of the functionality of Coreteye. For an easier overview and better understanding of the structure of the Coreteye code, see the class diagram in figure 5.9. Because Coreteye offers a range of features and functionality we will begin with its core purpose before expanding on the extra code around it.

Core Functionality

The core functionality of Coreteye is the data fusion of the eye tracker events with the button events, the processing of both together, and the fission of the generated component event from these. Upon starting Coreteye neither Trackteye, Presteye, or any Component

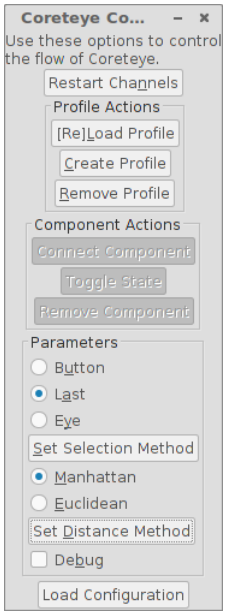

Figure 5.8: Coreteye graphical user interface. Not shown are the modal dialogs that are used to input information. The component actions are only available when a user profile has been loaded, not shown here.

can have established their communications as Coreteye provides the communication hub for all these programs. Thus Coreteye initially opens the communication channels for all these to connect to. The connections are not stateless: certain events happen upon successful connections in the case of TrackerPost and ComponentPost. When Trackteye connects it receives stored eye tracker configuration data if previously saved. Trackteye on the other hand can send four different messages to Coreteye:

Data The data type message offers up the current gaze position of the user including the active eye tracker.

Command This message is used to tell Coreteye to store the eye tracker extrema in the current user profile so that it can be retrieved upon later reconnects.

Position The position data is the current eye position of the user. It is immediately passed on to any connected mirror component for it to use in the case of automatic mirror adjustment.

Register This message tells Coreteye which eye trackers are currently available. This information is used for example upon user profile creation.

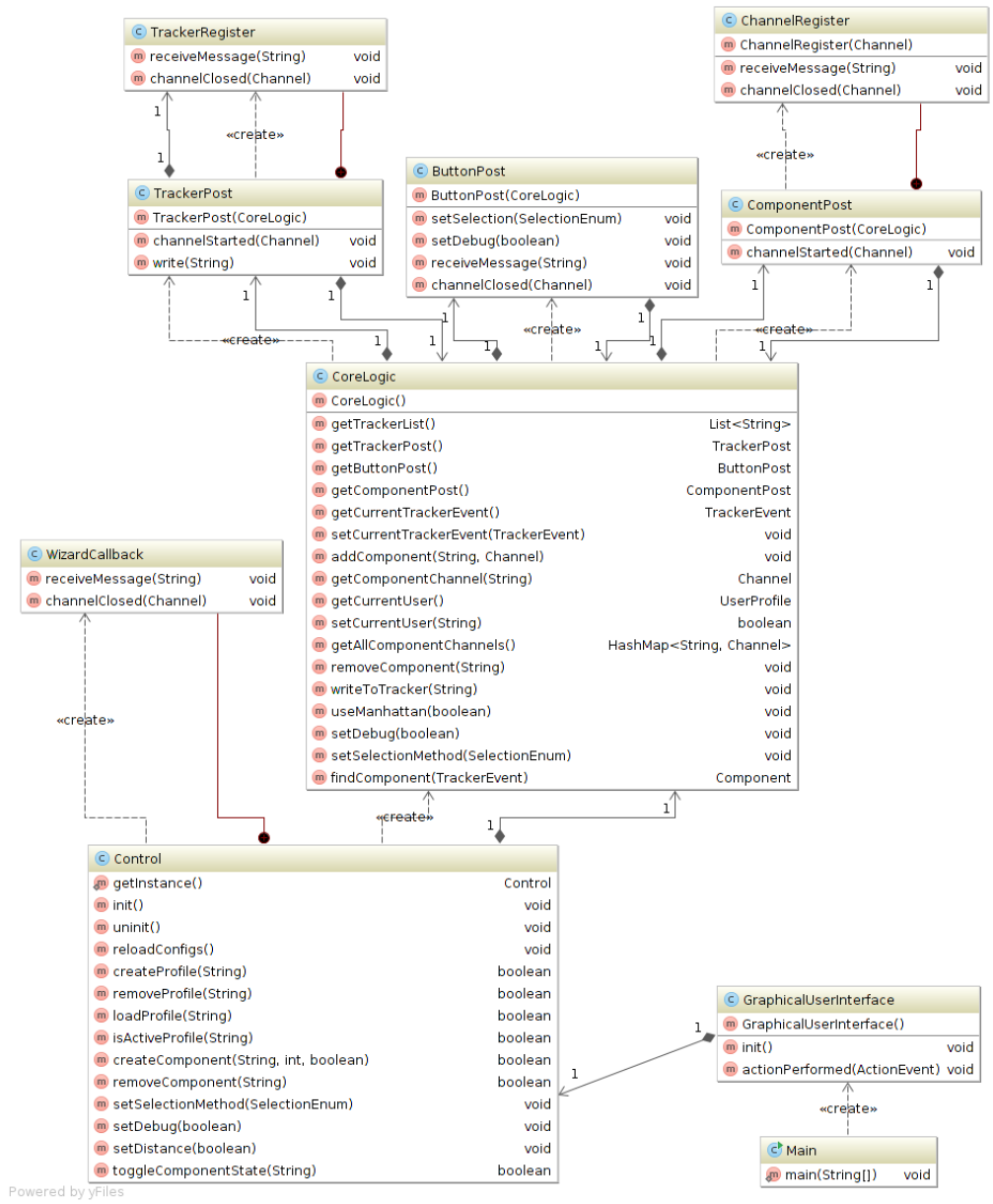

Figure 5.9: Class diagram of Coreteye. Note that the Control class is a wrapper for CoreLogic that is used from the GUI. CoreLogic itself contains the main logic. The Tracker-, Button-, and ComponentPost classes implement the communication receivers to abstract these away from CoreLogic.

For ComponentPost Coreteye receives the name of the component upon successful connection, which it stores so that it can later send events to the correct component on the correct channel. Presteye is the only stateless connection: the system will run even without button events, although no component will of course then be ever triggered.

Now that all communication partners are ready the system can fully run. Depending on the selection method chosen CoreLogic will recalculate the currently active component. The selection method can be changed at runtime via the GUI or alternatively in the Coreteye configuration file at startup. For the button selection method this will happen only when the user is looking at a new component and pressing the center button at the same time. The last selection and eye selection will recalculate on every gaze point update, with the only difference between the two that when no component is being looked at currently, eye selection will remove the current component and set it to null.

The current component is selected via a Voronoi diagram[2] method that is applied on a per eye tracker granularity. This means that as soon as a component is placed on an eye tracker just looking anywhere at the tracking plane of said tracker will count as looking at that component. At the same time placing multiple components will smoothly subdivide the area so that each component has the maximum available area that can be looked at to select it. This method was chosen to hide accuracy problems of the gaze data due to tracking noise. In turn this means that the only way not to select a component is not to look at an eye tracker where a component has been placed. We have also implemented two different methods of distance calculation for the Voronoi diagram, configurable via the GUI at runtime or the configuration file: euclidean and Manhattan based distance metrics.

Upon a button press and subsequent button event Coreteye thus gets the current component if available and sends the button event to it. The exception to this is if the Wizteye component is connected: in this case all button events are sent to it as it has priority over the other components. Apart from a button event Coreteye will also send selected events. These notify components that they are the currently active components.

[2] A Voronoi diagram is a diagram that allocates every point in a plane to its closest neighbor of a specific subset of points. In our case the subset is the location of the components and the points that are allocated are the gaze data points.

An event is sent upon setting a component to selected and when it is deselected. The event includes the old selected component and newly selected component although this information is not currently being used.

Further Functionality

Coreteye offers up extra functionality beyond the core data fusion and event fission required for the main part of the system to run. Most of this functionality can be found in the Control class which wraps the CoreLogic class for easier implementation of commands. Commands can be specified either through a command channel for external programs, utilized by us for Wizteye, or via the graphical user interface which also handles Coreteye startup through a Main class.

Moteye allows the usage of user profiles to separate different eye tracker configurations, available component sets, and users. These user profiles can be created, managed, and removed via Control. This also includes adding and removing components from an active user profile.

Note that configuration options such as the selection method and the distance method are profile independent and not stored with them.

5.5 Compteye

Components represent physical or virtual car components like the mirror, radio, air conditioning etc. Moteye allows to place an arbitrary amount of components in the car. However it isn't feasible to have too many components in the field of a single eye as accuracy might become a problem. A component is required to implement a simple interface and instantiate a listener. For more details see section 5.6.2.

Wizteye is a configuration wizard component which allows loading and creation of user profiles. Also in the process of creating a profile, eye trackers are calibrated and components registered. Note that Wizteye is modal: all input events are redirected to it while it is actively running. This is sensible because Wizteye is used to setup the

system for a user, thus no components are ready to be used when Wizteye is running anyway. Left/right moves through the dialogs and up/down allows scrolling through options. Confirmations are done via button press. See figure 5.10 for the start screen as example.

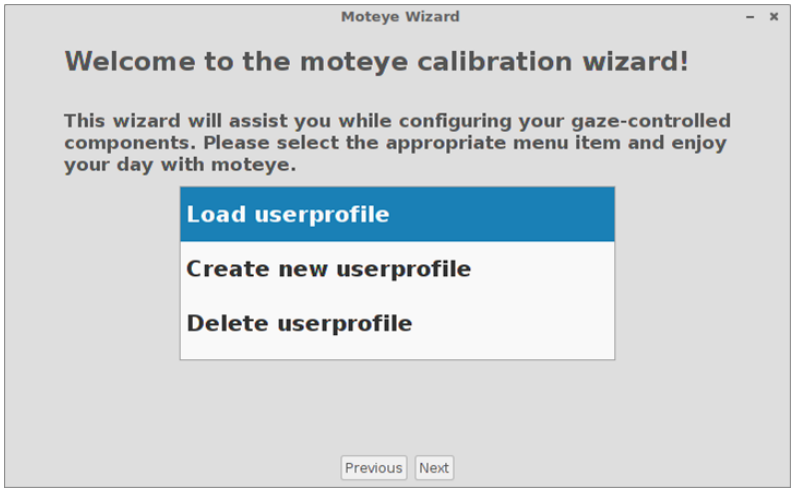

Figure 5.10: Wizteye configuration wizard component.

Voxteye is a simple virtual radio type component. It features multiple web radio channels which can be switched through using left and right button controls as well as mute/unmute the current channel using up/down. Originally it was planned to control volume via up/down, however this is not readily possible via Java. If up/down had worked for volume control we could have used the center button to switch the radio on/off. A screenshot of Voxteye can be seen in figure 5.11.

Callteye is a simple hands free call component implementation running on Android Devices which allows the user to scroll through their contact list using up/down and calling someone using left/right controls. While the call is in progress the volume can be changed with up/down instead. Note that due to restrictions of Android, ending a real call via Moteye does not work. We implemented the functionality in a way that allows the user to select whether to allow real calls or fake ringtones and whether to use the real address book or a fake one. A screenshot of Callteye running on a Nexus 5 device can be seen in figure 5.12.

Figure 5.11: Voxteye radio component.

Figure 5.12: Callteye hands free call component.

Mirteye is the mirror component controlling the physical left side mirror hooked to our Arduino board. It can be adjusted using left/right/up/down button events. Therefore these button events are directly forwarded via serial port to the Arduino board. There the output pins of the mirror motors are set on power accordingly until the joystick released triggering a released event which stops all power to the mirror. Mirteye includes all Arduino related code thus also the joystick. Here the byte stream coming from the board is parsed consisting of a triple of current X and Y positions and the button press state. This information is translated into the various button events and sent to Coreteye.

Furthermore the Mirteye component features our automatic mirror adjustment feature. This can be activated when selecting the mirror and pressing the center button. In this mode, Mirteye evaluates the current user's eye position and adjusts the mirror automatically. For our prototype this is done via linear mapping only (for further considerations on this, see section 6.1). Here the eye trackers calibrated X/Y bounds are mapped onto the mirror's X/Y rotation directly allowing the mirror to follow the user's head.

Figure 5.13: Mirteye mirror component.

Additionally to these there were two debug components implemented: Presteye as seen in figure 5.14 which is a virtual button UI and a SimpleComponent only printing its received input events.

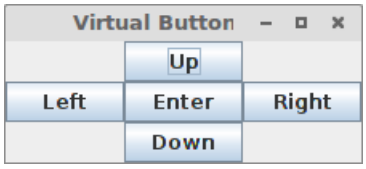

Figure 5.14: Presteye simple button UI component.

5.6 Depteye

This section describes background mechanics that were implemented to glue the distributed system together. All of the various Java packages were placed into the Depteye master package to allow a clear abstraction and ease of use for all code built on it.

5.6.1 Communicator

Due to the distributed nature of the Moteye project we required a communication architecture for passing messages around. Since Moteye in its current form is simply a prototype it was clear from the beginning that we would not utilize a vehicle data bus. Therefore we turned to using TCP/IP based communication as that offered up the highest flexibility for the prototype. As Java does not offer up real asynchronous socket communication we implemented a basic form of it ourselves.

Communication is based around a channel concept. One communication partner creates a channel for the other partner to connect to. The basic communication runs in internal threads, thus decoupling the receiving of messages from the thread of the program to avoid blocking I/O. A channel can be given a callback in the form of an interface that is called when a message is received in which the program can then react to it. Writing a message to the channel is done via a simple method call with a message. All messages in our case are String based as we prefer utilizing JSON for data transfers where simple String words did not suffice.

Creating and connecting to a Channel is handled via the ChannelFactory class. To further increase the flexibility of the Communicator we also implemented a few features that decrease the complexity of managing multiple channels. One example that is used

on multiple occasions in Moteye is the capability of registering a FactoryCallback that is called whenever a Channel is connected to on the server side so that each connected channel can be specifically handled.

5.6.2 Component

The Component package abstracts away the Component side of the channel based communication with Coreteye. This allows each component to receive and handle any kind of event coming from Coreteye in a simple, uniform way. This package is the primary code required to develop further components for the Moteye system. Currently there are ButtonEvents informing about the joystick state, PositionEvents informing about the current eye position of the user, and SelectedEvents informing about whether or not a component is being selected by gaze or button input. The class diagram for the ComponentEventReceiver can be seen in figure 5.15.

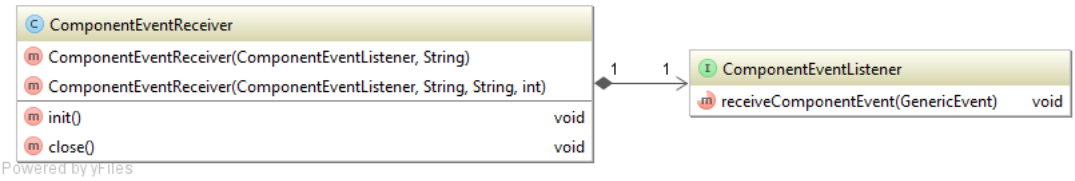

Figure 5.15: ComponentEvent Interface.

For a program to work as a component it must simply instantiate the ComponentEventReceiver with a callback interface which will be called on events and a String name which is the component name given to Coreteye. Nothing more is required for this to work. Note that sending messages from a component back to Coreteye is currently not possible as this function is not required within our scope.

5.6.3 Configurator

The Configurator allows easy configuration management of Moteye using JSON files. coreteye.json defines configuration values concerning Coreteye:

address Specifies the address where the Coreteye application is running.

ports Specifies the ports to be used for these modules to connect to Coreteye.

algorithm Use "manhatten" or "euclidean" distance measurement to evaluate which component was looked at.

trackerEventTimeoutSeconds
Delay in seconds for which a incoming event from the eye trackers is considered valid for use in selection logic.

debug The debug flag disables the requirement of having Trackteye connected for creating and loading user profiles.

allowRemoteControl
Set to "false" if the wizard should not be allowed to make changes to configuration and database.

selection Which selection mode is to be used: "eye", "last", or "button".

confidence_limit
In case TrackerEvents reach Coreteye with a low confidence, set an upper limit (exclusive) here to ignore them.

The following listing is an example of the contents of coreteye.json:

Listing 5.1: coreteye.json

```
0  {
1    "address": "localhost",
2    "ports": {
3      "presteye": "55662",
4      "trackteye": "55660",
5      "components": "55663",
6      "wizard":"55664"
7    },
8    "algorithm": "manhattan",
9    "trackerEventTimeoutSeconds": "2",
10   "debug": "false",
11   "allowRemoteControl":"true",
12   "selection": "last",
13   "confidence_limit":"0.1"
14 }
```

Similarly the trackteye.json file is defined, setting configuration for Trackteye:

address Specifies the address where the Trackteye application is running.

ports Specifies the port to be used for eye trackers to to connect to Trackteye.

gazeDataOption

Which filter for the tracking data is to be used, see gazeDataOptionValues for values.

gazeDataOptionValues

A list of valid options to be used as gazeDataOption.

dataPushInterval

Time interval in milliseconds on which Trackteye is supposed to push gaze data to Coreteye.

frameCacheSize

Number of latest gaze data frames to keep for calculations.

movingWindowInterval

Time interval in milliseconds defining valid gaze data frames to be considered for the average filter.

dwellTimeInterval

Time interval in milliseconds of valid gaze data frames to be considered for calculating if a fixation occurred.

leftTrackingMargin

Size of the left margin of the calibrated tracking plane (0.0–1.0) which causes confidence penalty.

rightTrackingMargin

Size of the right margin of the calibrated tracking plane: (0.0–1.0) which causes confidence penalty.

The following listing is an example of the contents of trackteye.json:

Listing 5.2: trackteye.json

```
0 {
1   "address": "127.0.0.1",
2   "port": "55661",
3   "gazeDataOption": "AVERAGE",
4   "gazeDataOptionValues": "AVERAGE, PREDICT, LATEST, EURO",
5   "dataPushInterval": "100",
6   "frameCacheSize": "100",
```

```
7   "movingWindowInterval": "200",
8   "dwellTimeInterval": "100",
9   "leftTrackingMargin": "0.05",
10  "rightTrackingMargin": "0.95"
11 }
```

The db.json file defines database access:

driver The database driver to be used.

address Specifies the address where the database is running.

ports Specifies the port of the database.

name Name of the database.

user User for the database.

password Password for the database.

The following listing is an example of the contents of db.json:

Listing 5.3: db.json

```
0 {
1   "driver":"com.mysql.jdbc.Driver",
2   "address":"localhost",
3   "port":"3306",
4   "name":"moteyedb",
5   "user":"root",
6   "password":""
7 }
```

The mirror.json file defines startup configuration for the mirror.

defaultX Time in milliseconds which the mirror is to rotate to the right at startup.

defaultY Time in milliseconds which the mirror is to rotate downwards at startup.

The following listing is an example of the contents of mirror.json:

Listing 5.4: mirror.json

```
0 {
1   "defaultX":3500,
2   "defaultY":1000
3 }
```

5.6.4 Database

As the Moteye prototype offers persistent storage for user profiles a database was required. We opted for a SQL database. This package contains the abstraction code for access to said database. Primary database is MySQL, but MariaDB was also used successfully. For each user profile tracker calibrations are stored independently. Components and eye trackers are stored uniquely.

5.6.5 Log

This package includes our two logging mechanisms for the project. They offer up a standardized way to log system output and error messages, enforcing a more usable usage of system.out.

5.6.6 Objects

The objects package stores all objects that we utilize across multiple packages. In our case this consists of the various event objects and the user profile objects. See figure 5.16 for the class diagram of this package.

5.6.7 Util

The Util package (short for utility) contains miscellaneous code that is used in multiple other packages. The two functional code classes Bounder and Helper are supplemented by command data in the form of Java enumerations and data objects.

5.6.8 Wizard

The Wizard package provides a simple to use Java wizard implementation after we tried to use an existing framework that proved to be too complex, too error prone, and too obfuscated for our needs. Our wizard implementation provides a simple to use,

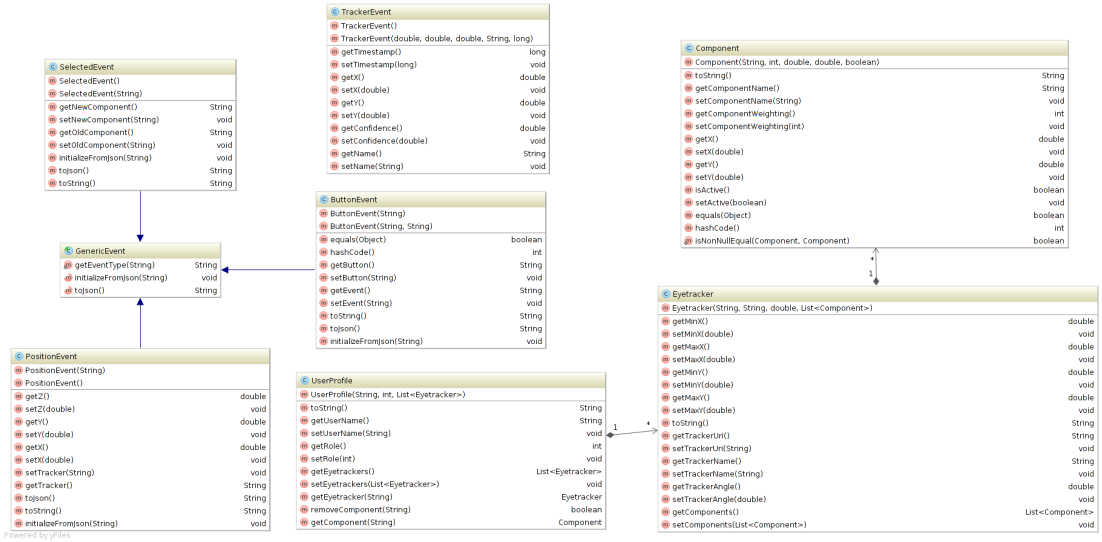

Figure 5.16: Class diagram of the Depteye Objects package.

extremely flexible framework for navigating through a dialog structure, both via mouse or Presteye commands. It is easily capable of traversing dialogs in either a linear fashion or a nonlinear fashion, as required. It also accepts dynamic content which was vital for our needs but the previously tried framework failed to provide. Single steps in the wizard are abstracted in a way that allows the programmer to concentrate on the content of each step without having to work with the nasty bits of Java Swing.

5.7 Development Environment and Setup

For development the following software environment was used:

- Microsoft Visual Studio 2013 for developing C# using the Tobii GazeSdk DotNetApi 2.0.3.182.
- Intellij IDEA Ultimate 14 & Eclipse Kepler for Java modules.
- Java Development Toolkit & Runtime Environment 8.
- Arduino 1.0.6
- MySQL Server 5.6
- GIT for version control

In order to get the system running in full operation on the driving simulator setup we used it is suggested to do the following sequence of actions. They should result in a fully running and working Moteye system, ready for usage. Of course these need to be adapted if the base setup of the environment differentiate.

- Start the driving simulator computer and connect it to the internet.
- Make sure the driving simulator's coreteye.json and db.json configurations reflect correct addresses.
- Start Voxteye (radio.jar) on the driving simulator and place it where wanted.
- Start OpenDS on the driving simulator loading the village scenario.
- Start The computer that is to run the moteye logical programs.
- Make sure the driving simulator's coreteye.json, db.json and trackteye.json configurations reflect correct addresses.
- Start the MySQL database and if it is the first time, create a database "moteyedb" and load the moteyedb.sql file.
- Start Coreteye Java module (Main)
- Start Trackteye Java module (GazeContextcontroller)
- Start Trackteye C# module (Trackteye.exe)
- Start Mirteye and wait for it to rotate to the default position.
- Start Wizteye on the driving simulator if the wizard is to be used, alternatively load/create a profile in Coreteye.

6

Future Work

This chapter describes what we envision as possible future work for Moteye. Primarily we will focus on the system we ended up implementing, but our initial alternatives offer up some interesting possibilities too, so they will be mentioned as well.

6.1 System Further Work

Possible further work on and with our system can be divided into two categories: work to further improve the current implementation of our concept and work to evaluate the feasibility of it.

Improvements and Changes

While our system proves the concept as a workable alternative to existing solutions for controlling secondary or tertiary systems in an automotive context, there is always room for improvement.

Primarily further work on the automatic mirror adjustment component would be rewarding. The current implementation uses a comparatively simple relative mapping without static reference points. While this works initially it does not guarantee a satisfying degree of accuracy for the future. To increase the general usefulness and accuracy of this mechanism we propose utilizing a mirror that can be finely controlled with a stepper motor. However currently the software is not ready for an accurate calculation and placing of the mirror in respect to the user's eye position. This would have to be implemented accordingly.

A more theoretical improvement would be to rework the software architecture of Coreteye. For this we propose decoupling the selection modes from the fusion and fission code, allowing new modes to be easily implemented without having to work around existing modes as would be required now. This cold be done by writing an interface for the selection modes which Coreteye will call the correct instance of for any selected selection method. Difficulties could arise here due to the problems in cleanly separating the code.

We mentioned in section 2.4.1 that we believe auditive feedback to be too slow compared to the visual feedback that we've implemented for eye tracking in an automotive context. However as the feedback is something the component has to implement (as a reaction to the selection event, see section 5.4), it would be a simple matter of changing a component's behavior to enable any required feedback. Therefore a comparison between various feedback modes could also be an interesting aspect of an extension to our system.

However with the current eye tracker setup receiving good data from them is not always possible due to hardware limitations of the eye trackers. Therefore we suggest replacing the eye trackers with a camera based tracking system which also tracks the users head and allows for a larger tracking volume. Theoretically this would only require writing

a new Trackteye implementation as the tracking data is abstracted for the rest of the system anyway. Either when using the existing setup or when utilizing alternative eye trackers, the actual tracking algorithms can be further improved too.

Finally it would be interesting to test the system in a real vehicle in a safe environment. Depending on the scope of integration some software work might be required. But for an initial test how the system functions in its intended environment the current prototype should suffice. Difficulties might arise from the placement of the eye trackers and the problem of user occlusion. Furthermore the wiring could be somewhat difficult due to USB length concerns and quality of signal.

Extensions

Moteye was built from the ground up to be as flexible as a prototype can be. Therefore it is very easy to extend the system functions by attaching and implementing new components.

One of the more better suited possible components would be integration of Moteye with a heads up display. Since the projection area for HUDs is never the interaction surface, using gaze and the joystick to control the HUD interface could remove a further extra interaction mechanism. Many HUDs already offer integration with the vehicle's entertainment system. Therefore a HUD component would also allow control of any entertainment system that is connected to it, again decreasing the complexity of second order interactions.

Apart from the HUD a variety of further components can be considered. The vehicle's climate control is one further possibility. The system could be used to control temperature and strength of the air flow. Cruise control could also be integrated somewhat, especially in future vehicles with sophisticated autonomous driving systems. Control of interior lights, seat adjustment, and sliding roof controls are further possible components.

For all of these possible extensions the connection to the Moteye system is the easy part; actually implementing the drivers for the hardware is the larger hurdle.

An extension to the feature set of Coreteye would be to allow components to broadcast messages to all other connected components. Callteye could use this feature for example to mute all other audio when receiving or starting a call. This feature would require a specification of the available broadcast actions and require new implementation at the Compteye and Coreteye level. However it would allow a more dynamic and more intelligent usage of the system, removing extra steps from the user interaction. In short we propose adding automation capabilities to the component interactions.

Evaluation

Moteye offers up a wide range of possible variables that can be modified and evaluated. From a human computer interaction aspect the usability and user distraction are primarily interesting. We believe that Moteye decreases distraction while driving when using secondary or tertiary inputs in a vehicle that are connected to Moteye. Usability of Moteye might be not as high as hoped due to the previously mentioned problems with the eye tracking hardware in the current state of the project. However even so the system is currently usable albeit not at its peak yet. Thus an evaluation whether improved tracking hardware increases the system usability could be a future work.

A more detailed evaluation could also evaluate system settings. Primarily this includes deciding which of the selection methods is the better one of the three currently implemented. Furthermore a range of values can be modified internally to play with the current tracking algorithms and general system response time. This goes from tracking filters to the filters within Coreteye all the way to the components.

Of course single components could also be evaluated in comparison to the current interactions state used with them. It might be interesting to directly compare interaction with for example the radio via physical buttons as is standard nowadays and via Moteye to discern whether one is directly better from a user experience point of view.

6.2 Alternative Concepts

These concepts are alternatives that came up in cooperation with our partner e.Solutions GmbH besides our final concept. These are future work in the sense that they expand on the topic of utilizing eye trackers in an automotive context. The concepts are listed here for the sake of completion.

Attention Assistance

This concept is based on a serious game context with eye trackers in an automotive environment. The basic principle is to use an eye tracker to keep the driver's attention focused on the road by having him play a very simple game in the HUD.

The proposed game consisted of looking at various displayed elements on the windscreen or HUD within a certain time frame. By spreading these elements out over the course of a drive, the game could also measure driver reaction times to sudden changes in his visual field of view. By utilizing scores or other motivating factors the game could possibly decrease driver fatigue, especially on long or unsophisticated drives, such as on highways.

Copilot

The copilot concept is based on the fact that drivers occasionally miss signs or other important road events. By using eye trackers to detect where the driver has looked and a system that is aware of its surroundings, this concept proposes an assistant for the driver. The goal is to be able to alert the driver to missed signs and important events, for example a person on a street crossing, if the driver has not seen the event already. It could also include objects in the rear or side view mirrors. Such a system might also be used to train drivers and to assist experienced drivers in being fully aware of their surroundings at all times.

7

Conclusion

Moteye is a prototype implementing our conceptual idea of Gaze Context Mapping using eye trackers in an automotive context. Therefore we introduced eye tracking as an emerging interaction mechanism beneficial for human computer interaction as well as the automotive field. We also showed what problems might arise from it. Based on this related work we proposed our concept including use cases and goals that were to be achieved in this project. Then, to be able to implement the concept, we had to evaluate which technologies were required and set up a schedule for our implementation term. The implementation itself was then documented in order to be comprehensible for any future work based on it. Then future work itself was suggested extending and evaluating the system as well as alternative concepts that were considered but dropped.

Concluding this, we have shown that our concept was feasible to realize and offers a new way of interacting with physical and virtual car cockpit components. Although

this is a prototype we have had mixed personal experiences about the concept's real world applicability. We found it positive that mapping multiple car components to a single control was possible and easy to use. Also the gaze tracking and selection of these components worked sufficiently well. However when you are not used to selecting something with one's eyes it feels strange. Thus a training phase may be required for anyone actually using such a system. Generally we liked the last-select mode best as pressing a button is an additional hurdle in interaction, even though this mode has disadvantages concerning the Midas touch problem. All this would have to be evaluated to draw significant conclusions however.

List of Figures

Bibliography

[1] ZHU, Zhiwei ; JI, Qiang: Eye and gaze tracking for interactive graphic display. In: *Machine Vision and Applications* 15 (2004), Nr. 3, 139-148. http://dx.doi.org/10.1007/s00138-004-0139-4. – DOI 10.1007/s00138–004–0139–4. – ISSN 0932–8092

[2] TONNIS, Marcus ; BROY, Verena ; KLINKER, Gudrun: A Survey of Challenges Related to the Design of 3D User Interfaces for Car Drivers. In: *Proceedings of the IEEE Conference on Virtual Reality*. Washington, DC, USA : IEEE Computer Society, 2006 (VR '06). – ISBN 1–4244–0224–7, 134–

[3] TOBII TECHNOLOGY GMBH (Hrsg.): *The history of eye-tracking technology*. http://www.tobii.com/en/what-is-eye-tracking/the-history-of-eye-tracking-technology/. Version: Mar 2015

[4] JACOB, Robert J.: What you look at is what you get: eye movement-based interaction techniques. In: *Proceedings of the SIGCHI conference on Human factors in computing systems* ACM, 1990, S. 11–18

[5] DREWES, Heiko: *Eye gaze tracking for human computer interaction*, lmu, Diss., 2010. http://edoc.ub.uni-muenchen.de/11591/1/Drewes_Heiko.pdf

[6] BULLING, Andreas ; ROGGEN, Daniel ; TRÖSTER, Gerhard: It's in your eyes: towards context-awareness and mobile HCI using wearable EOG goggles. In: *Proceedings of the 10th international conference on Ubiquitous computing* ACM, 2008, S. 84–93

[7] DUCHOWSKI, Andrew: *Eye tracking methodology: Theory and practice*. Bd. 373. Springer, 2007

[8] BULLING, Andreas ; GELLERSEN, Hans: Toward mobile eye-based human-computer interaction. In: *Pervasive Computing, IEEE* 9 (2010), Nr. 4, S. 8–12

[9] TOBII TECHNOLOGY AB: Tobii Eye Tracking – An introduction to eye tracking and Tobii Eye Trackers. Version: Jan 2010. http://www.tobii.com/Global/Analysis/

Training/WhitePapers/Tobii_EyeTracking_Introduction_WhitePaper.pdf. 2010. – White Paper

[10] SIBERT, Linda E. ; JACOB, Robert J. K.: Evaluation of Eye Gaze Interaction. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA : ACM, 2000 (CHI '00). – ISBN 1–58113–216–6, 281–288

[11] JACOB, Robert J.: Eye movement-based human-computer interaction techniques: Toward non-command interfaces. In: *Advances in human-computer interaction* 4 (1993), S. 151–190

[12] ISTANCE, Howell ; BATES, Richard ; HYRSKYKARI, Aulikki ; VICKERS, Stephen: Snap Clutch, a Moded Approach to Solving the Midas Touch Problem. In: *Proceedings of the 2008 Symposium on Eye Tracking Research & Applications*. New York, NY, USA : ACM, 2008 (ETRA '08). – ISBN 978–1–59593–982–1, 221–228

[13] GLENSTRUP, Arne J. ; ENGELL-NIELSEN, Theo: *5.4 Main Problems of Eye-Gaze Interfacing*. http://www.diku.dk/~panic/eyegaze/node27.html. Version: Sep 2014

[14] VELICHKOVSKY, Boris ; SPRENGER, Andreas ; UNEMA, Pieter: Towards gaze-mediated interaction: Collecting solutions of the "Midas touch problem". In: *Human-Computer Interaction INTERACT'97* Springer, 1997, S. 509–516

[15] TOBII TECHNOLOGY GMBH (Hrsg.): *Advanced Driver Assistance with Tobii Eye Tracking*. http://www.tobii.com/en/products-services/advanced-driver-assistance/. Version: Mar 2015

[16] KERN, Dagmar ; MAHR, Angela ; CASTRONOVO, Sandro ; SCHMIDT, Albrecht ; MÜLLER, Christian: Making Use of Drivers' Glances Onto the Screen for Explicit Gaze-based Interaction. In: *Proceedings of the 2Nd International Conference on Automotive User Interfaces and Interactive Vehicular Applications*. New York, NY, USA : ACM, 2010 (AutomotiveUI '10). – ISBN 978–1–4503–0437–5, 110–116

[17] POITSCHKE, T. ; LAQUAI, F. ; STAMBOLIEV, S. ; RIGOLL, G.: Gaze-based interaction on multiple displays in an automotive environment. In: *Systems, Man, and Cybernetics (SMC), 2011 IEEE International Conference on*, 2011, S. 543–548

[18] FLETCHER, Luke ; ZELINSKY, Alexander: Driver Inattention Detection Based on Eye Gaze-Road Event Correlation. In: *Int. J. Rob. Res.* 28 (2009), Juni, Nr. 6, 774–801. http://dx.doi.org/10.1177/0278364908099459. – DOI 10.1177/0278364908099459. – ISSN 0278–3649

[19] NAWAZ, T. ; MIAN, M. ; HABIB, H.: Infotainment Devices Control by Eye Gaze and Gesture Recognition Fusion. In: *IEEE Trans. on Consum. Electron.* 54 (2008), Mai, Nr. 2, 277–282. http://dx.doi.org/10.1109/TCE.2008.4560086. – DOI 10.1109/TCE.2008.4560086. – ISSN 0098–3063

[20] ALT, Florian ; SCHNEEGASS, Stefan ; AUDA, Jonas ; RZAYEV, Rufat ; BROY, Nora: Using Eye-tracking to Support Interaction with Layered 3D Interfaces on Stereoscopic Displays. In: *Proceedings of the 19th International Conference on Intelligent User Interfaces*. New York, NY, USA : ACM, 2014 (IUI '14). – ISBN 978–1–4503–2184–6, 267–272

[21] HORBERRY, Tim ; ANDERSON, Janet ; REGAN, Michael A. ; TRIGGS, Thomas J. ; BROWN, John: Driver distraction: the effects of concurrent in-vehicle tasks, road environment complexity and age on driving performance. In: *Accident Analysis & Prevention* 38 (2006), Nr. 1, S. 185–191

[22] STUTTS, Jane C. ; ASSOCIATION, American A. u. a.: *The role of driver distraction in traffic crashes*. AAA Foundation for Traffic Safety Washington, DC, 2001

[23] BURNS, PC ; PARKES, A ; BURTON, S ; SMITH, RK ; BURCH, D: How dangerous is driving with a mobile phone. In: *Benchmarking the impairment to alcohol* 56 (2002)

[24] *TheEyeTribe Developer Guide*. http://dev.theeyetribe.com/dev/. Version: Mar 2015

[25] CASIEZ, Géry ; ROUSSEL, Nicolas ; VOGEL, Daniel: 1 € filter: a simple speed-based low-pass filter for noisy input in interactive systems. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* ACM, 2012, S. 2527–2530

[26] SALVUCCI, Dario D. ; GOLDBERG, Joseph H.: Identifying Fixations and Saccades in Eye-tracking Protocols. In: *Proceedings of the 2000 Symposium on Eye Tracking Research & Applications*. New York, NY, USA : ACM, 2000 (ETRA '00). – ISBN 1–58113–280–8, 71–78

Glossary

dwell time The amount of time an eye dwells on a gaze target. 10, 12

eye tracking Eye tracking is the process of tracking and recording the three dimensional rotation of one or both eyes of a user. 1

gaze Gaze determines a person's current line of sight or point of fixation [1]. 2

gaze tracking Gaze tracking is the process of tracking and recording gaze movements of a person over an area. 2

GCM Gaze Context Mapping. 15, 18, 19, 25, 26, 34

Midas Touch Unintentionally initiated action or selection when eyes are used to control or select UI-elements [12]. 11, 19

Moteye Our group name during this project. Stems from a blending of the words automotive, tracking, and eye. 1

saccade A saccade is a rapid eye movement between fixation points. 8