Eye Tracking Simulation Framework

Tobias Widén

Degree project in
Computer Science
Second cycle
Stockholm, Sweden 2013
Eye Tracking Simulation Framework

Ramverk för eye tracking-simulering

TOBIAS WIDÉN
twiden@kth.se

Master’s thesis in Computer Science (30 credits)
at the School of Computer Science and Communication,
Royal Institute of Technology, 2013-03-17

Supervisor: Josephine Sullivan
Examiner: Stefan Carlsson
Project commissioned by: Georgios Dimitriadis at Tobii Technology

TRITA xxx yyyy-nn
Abstract

Eye Tracking Simulation Framework

To facilitate eye tracking algorithm development that would benefit from simulation we created an eye tracking simulation software library and Application Programming Interface with reusable components.

The main part of this library consists of a simulation core able to accept different geometrical models for the eyes, depiction mechanisms, illumination sources and eye tracking algorithms.

The library has support for data generation using these models as partial input. It is also capable of performing eye tracking, including eye model calibration using the generated data and algorithms as input.

We describe a selection of methodological techniques and principles for software engineering and exemplify how these can be used to improve functionality and usability of software systems.

Finally, we discuss future uses and improvements which could further increase the tool’s usability and engineering qualities.

Sammanfattning

Ramverk för eye tracking-simulering

För att underlätta utveckling av eye tracking-algoritmer har vi skapat ett simuleringssverktyg och programmeringsgränssnitt med återanvändbara komponenter.

Verktyget består i huvudsak av en kärna, vilken utifrån geometriska modeller för ögon och ljuskällor, mekanismer för optisk projektion och algoritmer genererar eye tracking-data från virtuella scener samt kalibrering av ögonmodeller med algoritmer som tolkar genererat data.

Vi beskriver ett urval av formella metoder och principer för mjukvaruutveckling och exemplifierar hur dessa kan användas för att förbättra funktionalitet och användbarhet av mjukvara.

Slutligen diskuterar vi framtida användningsområden och förbättringar som ytterligare kan öka verktygets användbarhet och ingenjörsmässighet.
# Contents

1 Introduction ................................................................. 1
   1.1 Basic Definitions .................................................. 1
   1.2 Background .......................................................... 1
   1.3 Typical Scenario .................................................... 2
   1.4 Problem Statement .................................................. 3
   1.5 Objectives ............................................................. 4
       1.5.1 Simulation Model .............................................. 4
       1.5.2 Data Generation .............................................. 4
       1.5.3 Calibration ................................................... 5
   1.6 Scope ......................................................................... 5

2 Theory ........................................................................... 6
   2.1 Eye Tracking Simulation .............................................. 6
   2.2 Application Programming Interface ............................... 6
   2.3 Techniques and Principles for Software Design ............... 7
       2.3.1 Encapsulation .................................................... 7
       2.3.2 Coupling .......................................................... 7
       2.3.3 Structural Patterns .......................................... 8
       2.3.4 Behavioral Patterns ......................................... 9
       2.3.5 Inheritance ...................................................... 10
       2.3.6 Principle of Least Astonishment ........................... 11
   2.4 API Qualities ............................................................ 11
       2.4.1 Good Naming .................................................... 11
       2.4.2 Difficult to Misuse ............................................ 12
       2.4.3 Minimally Complete ......................................... 12
       2.4.4 Open For Extension ......................................... 12
       2.4.5 Documentation ................................................ 13

3 Method ........................................................................... 14
   3.1 Defining Requirements ............................................... 14
   3.2 System Design .......................................................... 14
   3.3 Implementation ........................................................ 15
   3.4 Testing .................................................................... 15
# Requirements

4.1 Input and Output ........................................ 17
4.2 User Friendly ........................................ 17
4.3 High Coding Standards .............................. 18
4.4 Open for Extension ................................. 18
4.5 Robust and Tested ................................. 19

# System Design

5.1 Application Workflow ............................... 20
5.2 Major Components ................................. 20
5.3 Dependencies ....................................... 23
5.4 Extendibility ....................................... 23

# Implementation and Results

6.1 Programming Environment ...................... 25
   6.1.1 C++ Programming Language ................. 25
   6.1.2 BOOST ........................................ 25
6.2 The Optimization Problem ..................... 26
   6.2.1 Levenberg Merquardt Algorithm ............ 27
   6.2.2 Value Function Parameters ............... 27
   6.2.3 Finding a Good Start Guess ............... 28
   6.2.4 Value Function Output ..................... 29
6.3 Rotating Objects in Space ..................... 31
   6.3.1 Rotation Matrix .................................. 32
   6.3.2 Axis-angle Representation ............... 33
6.4 Tests and Documentation ....................... 35

# Conclusions

7.1 Usability ........................................ 36
7.2 Functionality ..................................... 37
7.3 Robust and Tested ............................... 37
   7.3.1 Dependencies .................................. 38
7.4 Open For Extension ............................ 38
7.5 Future Work ..................................... 39
   7.5.1 Gaze Tracking .................................. 39
   7.5.2 Experimental Eye Models .................... 39
   7.5.3 Extended Sensor Models ...................... 40
   7.5.4 Language Mappings ........................... 41
   7.5.5 Text-Based API ................................ 41
   7.5.6 Improved Display of Results .............. 42
   7.5.7 Multithreading ................................ 42
7.6 Final Words ..................................... 43

# Bibliography

44
Appendices

A Data Generation Example

B Data Interpretation Example

C Simulation Output

List of Figures

1.1 Tobii IS-2 Eye Tracker Without Housing .................................. 2
1.2 Tobii Glasses ............................................................................. 2

2.1 Visualization of cyclic and acyclic dependencies ....................... 8
2.2 Examples of Adapter and Facade patterns ............................... 10

5.1 Application Workflow .............................................................. 21
5.2 Distilled Class Diagram ........................................................... 22

6.1 Internal Reflection ................................................................. 29
6.2 Illustration of $e_1(\hat{v})$ ......................................................... 30
6.3 Illustration of $e_2(\hat{v})$ .......................................................... 31
6.4 Reflection in a Spherical Surface ............................................ 32

7.1 Suggested Improved Application Workflow ......................... 40
Chapter 1

Introduction

This chapter introduces relevant terminology that will be used throughout the thesis and explains the basic theory behind eye tracking. The concept of Pupil Center and Corneal Reflection (PCCR) eye tracking is described and we relate this to the goals of the project.

1.1 Basic Definitions

Eye Tracking
Determining the position and geometrical properties of the subject’s eye(s).

Gaze Tracking
Determining the direction of gaze or the three dimensional point that the subject is fixated on.

Glint
The corneal reflection of illuminators captured by the eye trackers’ sensors.

Users and Clients
In this report programmers that write code using the Application Programming Interface (API) are termed users and applications that depend on the API are termed clients.

1.2 Background

Tobii is a vendor of both eye tracking hardware and software: technology that makes it possible for a computer to know where the user is looking. The idea came from a research project at the Royal Institute of Technology in Stockholm and Tobii was founded in 2001 [1].
Eye tracking is used in psychology and vision research, user experience testing and market research. Another application is Augmentated and Alternative Communication (AAC) solutions for speech and computer access.

1.3 Typical Scenario

An eye tracker is a device containing illuminators and light sensors (fig. 1.1). The sensors capture illuminator reflections in the subject’s cornea and can through image processing algorithms determine the eye position and direction of gaze. The system can be a stand-alone unit or integrated into the user’s monitor. It can also be mounted on a pair of specially designed glasses (fig. 1.2) to determine where the user is looking.

![Figure 1.1: Tobii IS-2 Eye Tracker Without Housing](image)

![Figure 1.2: Tobii Glasses](image)

The simplest gaze tracker uses active illumination and a camera to perform so-called PCCR eye tracking. In summary, you do the following: Make sure that
the subject’s head stays stationary (e.g. using a chin rest). Then fit a function $f : \mathbb{R}^2 \rightarrow \mathbb{R}^2$ which estimates the 2D gaze point on the computer screen from the pupil center in the image.

The common choice for $f$ is a 2D polynomial often of the 1st degree. This is an extremely simplified model and takes no account of the composition of the eye or camera optics. Thus this gaze tracker is likely to suffer from systematic errors. The size and shape (vector field) of these errors are probably dependent on the non-modeled optical system parameters (e.g. lens distortion) as well as the subject’s eye shape and size.

1.4 Problem Statement

Creating models and algorithms for Near Infrared Light (NIR) image based eye/gaze tracking involves the development of geometrical models for the hardware used as well as the tracked human eyes. These models will of course not match reality perfectly. Thus, with increasing system complexity and performance requirements, it becomes increasingly hard to find and/or separate the sources of different errors in the system output, especially when only large pieces of the system can be tested at a time.

One way to mitigate this lack of insight would be to create a simulation environment where the entire eye tracking system may be modeled along with the subject (the person having their eyes tracked). In such an environment it becomes possible to introduce one or more known model discrepancies or necessary simplifications and analyze their impact on the final result. In this case one knows that all data inputs not related to the introduced error are unchanged.

If we need to improve overall performance of the gaze mapping function, maybe by choosing some $f$ other than the originally proposed 1st degree polynomial, it is important to know what to correct for, i.e. where the largest errors come from. If we are able to construct a simulation of:

- Point light source
- Spherical eye with a refractive surface
- Pupil placed behind the surface of the eye
- Camera that projects all the 3D objects onto a 2D surface

We could generate image data like pupil center and illuminator reflection to use while calibrating. Assuming a good approximation in this simplified world we would go ahead and add ever more complex models to generate data from, until we find a magnitude of error that is meaningful to target with a different $f$. These models could be a pinhole camera with radial distortion, or an ellipsoid eye with configurable principal axes.
1.5 Objectives

To facilitate eye tracking algorithm development that would benefit from simulation we created an eye tracking simulation software library and API. The main part of this library consists of a simulation core able to accept different geometrical models for the eyes, depiction mechanisms, illumination sources and eye tracking algorithms. The library has support for data generation using these models as partial input. It is also capable of performing eye tracking, including eye model calibration using the generated data and algorithms as input.

1.5.1 Simulation Model

The first step towards performing simulations is to describe the world and the relevant conditions in it. The simulation model consists of mathematical representations of physical objects, operations on these objects and ways to interact with and perform operations on that model. The physical components are eyes, cameras and illuminators. All of these entities have some properties in common, such as having a position in space and a certain orientation. Other properties are specific to different types of objects. For example, eyes have the properties radius, pupil size and cornea thickness associated with them whereas a camera model would have sensor size and focal length.

Operations on these objects are also part of the simulation model. Eyes need to be able to reflect light and cameras to capture light given a set of property values. This functionality is implemented on the specific object model but they need a common support structure. They need to agree on some basic facts about how light behaves when it passes from a medium to another or hits a surface. So before defining any operations, the model needs basic scaffolding that defines mathematical and physical operations. We need to find or implement such libraries before carrying on to perform actual simulations.

Another important aspect of the simulation model is that it should be a tool in itself for performing simulations of any type. The simulation core does not contain any simulation logic in itself. Rather, it is a function that takes a model and some eye tracking algorithms as input and applies these algorithms to the model and extracts relevant data.

1.5.2 Data Generation

Given the necessary tools to describe simulation models and the simulator process we can implement eye tracking algorithms. The algorithms generate images represented as glints and the extracted data are coordinates on the sensor plane.

We extract two types of images from the model. The first type is the illuminator reflection in the subject’s cornea. The second is the projection of the subject’s pupillary border on the sensor plane. These two types of data and the eye trackers geometrical setup are the input for PCCR eye tracking algorithms.
1.6 Scope

The thesis work concerns the software architecture for a simulation framework, not performing and reviewing simulations or mathematical models for eyes, cameras or illuminators. Our focus is to design the high-level structure of a software system and all the practices used to realize that structure.

The framework should only provide a set of basic model components, sufficiently complex to demonstrate data generation and model calibration. These objects can be used as building blocks and templates for modeling more advanced objects later.

The task of calibrating the rotation of the subject’s eye, the direction of gaze, is left outside of this project because of time restrictions. However the possibility to add this functionality at a later date affected decisions about the frameworks inner structure.

1.5.3 Calibration

From the generated data we simulate eye tracking and perform eye model calibration. Using the generated data we use the same geometrical model but different algorithms as input to the simulation process. These algorithms perform eye parameter calibration to fit the image data. We implement algorithms to calibrate the eye’s position and radius.
Chapter 2

Theory

In this chapter we explain what an Application Programming Interface (API) is and why Tobii wanted to create an API for generating eye tracking data and eye tracking simulation. We describe general techniques and principles for software engineering that are applicable for API design and qualities of a good API from a user perspective.

2.1 Eye Tracking Simulation

The idea of applying computer simulation to eye tracking is not new [7, 23]. Böhme et al. [7] describe an open-source software framework implemented in MATLAB that performs data generation and eye model calibration. They demonstrate the utility of simulation frameworks by comparing the performance of two algorithms for remote eye tracking.

Tobii also has experience of eye tracking simulations using MATLAB. They use a tool that was originally written to solve a specific problematic case through simulation, but later extended as the wide range of application for such a tool was recognized. This caused the initially small program to grow, as it was adapted to meet new demands.

The tool now has a broader use than it was designed for and has become hard to maintain and modify after being tweaked and fixed repeatedly without a clearly stated goal or set of design principles.

The current state of this tool gives us a level of functional maturity to target. But theoretically we could deliver better value to the users by repackaging this functionality as an API following modern software development techniques and principles. This is the aim of this project.

2.2 Application Programming Interface

An Application Programming Interface (API) defines reusable code that solves a specific problem. Often there are several applications in a problem domain that
2.3. TECHNIQUES AND PRINCIPLES FOR SOFTWARE DESIGN

2.3.1 Encapsulation

Encapsulation is a technique for minimizing dependencies among software components by defining strict external interfaces. The external interface is not only a contract between that particular component and its clients, but between the designer of the component and other designers [17].

Good API design is about providing a logical interface towards clients. However, the code behind an API can be rough and inefficient at first. Further complexity can be added later, without changing the interface or logical design [22, 24]. A module is encapsulated if clients can only access the module via its defined external interface [24].

2.3.2 Coupling

The notion of coupling describes the level of dependency between the components of a system. A loosely coupled system is one in which each of its components has little knowledge of other components and ideally no knowledge about their implementations.

Components that are tightly coupled are hard to reuse in isolation, since they depend on each other. Tight coupling leads to systems where components cannot
be changed or removed without changing other classes. The system becomes hard to learn, port, and maintain [10].

Important cases of tight coupling are cyclic dependencies (fig. 2.1). Objects with cyclic dependencies cannot be tested in isolation and it is difficult to tell if a bug is in one or the other. Since it becomes necessary to understand both components in order to understand either one, they are in terms of encapsulation one single component [14].

![Diagram of cyclic and acyclic dependencies]

*(a) Acyclic Hierarchy  (b) Cyclic Dependency*

*Figure 2.1: Visualization of cyclic and acyclic dependencies*

The goal is a loosely coupled system and an acyclic hierarchy of logically related components [22].

### 2.3.3 Structural Patterns

Structural patterns describe how entities are composed to form larger structures such as class or object hierarchies [10]. The patterns describe a convention for solving specific recurring structural problems in software design.

Structural patterns can be divided into class patterns and object patterns [10]. Class patterns deal with relationships between classes and object patterns deal with relationships between class instances. Relationships between classes are defined by the programmer prior to compiling the source code and are static at runtime. Object
patterns change dynamically during the program execution. Almost all patterns use inheritance to some extent [10].

The Adapter pattern converts the interface of a class into another interface and decouple the interface from implementation so that the two can vary independently. For example we can imagine an implementation of a system that depends on the driver of some hardware device. If the device vendor releases a new version of the driver with a different interface, the implementation of every client has to be changed. The adapter class mimics the interface of a driver so that clients can have a dependency to the adapter interface instead (fig. 2.2). Only the adapter implementation has to be changed if the driver interface changes. The system becomes easier to upgrade.

A type that covers another type can also be called a wrapper. Wrappers are usually not used for data transfer, like adapters, but for hiding data structures with possibly unstable interfaces inside more stable ones.

The Facade pattern displays a simplified front-end to a complex software hierarchy, usually providing convenience-methods [10, 22]. A concrete example would be a data storage facade that provides a single method that retrieves data from many data providers. Behind the facade there can be any number of independent data sources providing search results (fig. 2.2). Data sources can be added or removed behind the facade without changing any code in the clients. Approaches like this can preserve integrity of existing components while adding new functionality [9].

Patterns, such as Adapter and Facade [10] decrease the level of coupling by encapsulating and manage dependencies [10].

A very important pattern in the field of testing is the Dependency Injection pattern, where a dependency object is passed into a class (injected) upon creation instead of having the class create and store the object itself [22]. Dependency injection improves encapsulation and simplifies testing by bringing up a working instance of an isolated component [22].

When standard patterns are applied to an API, people who know the patterns gain insight into the framework faster [10], and from a given pattern a lot of design decisions follow automatically [10].

### 2.3.4 Behavioral Patterns

Classes are not only the structure of a program; they also control its behavior. Behavioral patterns are ways of working with algorithms and distribution of responsibility in software [10].

The Strategy pattern encapsulates an algorithm in an object and manages dependencies between the behavior of different components by defining a family of behaviors or algorithms. This increases the level of modularity and encapsulation and provides a way to configure a class with one of many behaviors [10], by injecting the behavior as a dependency of that object.

Defining strategies is particularly relevant when having many related classes that differ only in their behavior, such as different types of simulations on the same
CHAPTER 2. THEORY

2.3.5 Inheritance

Inheritance is a powerful way of composing objects and allowing implementations of data abstraction to be related hierarchically [9, 15]. It is also a useful mechanism for reusing code [24] and to let users of the API extend it by subclassing the built-in types [22].

However, allowing subclassing by clients can cause problems and hard to trace bugs if the clients override functionality in error-prone ways. The default implementation could be changing some inner state of the API but the overriding method in the client code may not, causing the whole application to exhibit incorrect behavior [22].

Unfortunately, by letting client applications override methods this way, the advantages of encapsulation that one associates with object-oriented languages are severely weakened [15, 24].

Hence, inheritance should be used with great care. The design of an API should only allow overriding if it is explicitly intended for this to be possible. Since classes with no virtual methods tend to be more robust [22], it is strongly recommended to never subclass only for convenience or to force an inheritance relationship when not appropriate [19, 22].

There are also performance issues to be aware of. In C++ for example, virtual
2.4 API QUALITIES

methods are slower [6, 22]. In this case there is a tradeoff between performance loss and the advantages of using inheritance.

2.3.6 Principle of Least Astonishment

A program should be easily read and understood by other programmers and especially the author [3]. The design should minimize the learning curve [9, 10] and match the user’s experience and expectations, borrowing heavily from similar programs with which the user is likely to be familiar [22]. In the programming language C++ for example, it is a good idea to try to mimic patterns of the Standard Template Library (STL) [4, 12, 22]. By doing so the API designer communicates crucial information to experienced developers and they will quickly catch on [22]. This reduces unpleasant surprises and means that the user can guess how to do things without reading the documentation [2].

However, the principle of least astonishment should not be interpreted as conservatism [21]. Sometimes it is motivated to introduce new concepts and styles. This may increase the threshold for a user’s first few interactions with an interface, but poor design choices will make the interface needlessly painful forever [21].

2.4 API Qualities

For code to be reused, programmers must find it appealing. That is, programmers must be willing to merge the API with their own code rather than use code from other sources, including writing their own [9]. Their inclination to do this is determined by the quality of the code itself, as well as any available examples and documentation [9].

2.4.1 Good Naming

As an implication of the principle of least astonishment the names of methods and parameters should intuitively match their respective behaviors [12, 16, 22] so that experienced programmers immediately recognize familiar patterns and naming conventions.

Throughout the API the same name should be used for the same concept, and different names for different concepts [2, 3, 4]. There should not be too many different things to keep in mind at the same time. One should strive for economy of concepts, and predictability [4].

Good naming drives good design [6] but also requires knowing the audience [4]. The user of an API will make certain assumptions based on previous knowledge. The API designer validates these assumptions by choosing good names [4, 12].
2.4.2 Difficult to Misuse

When the programming language Perl was created the motto was to make easy things easy and hard things possible [4]. This means that the programming environment should protect the user, without being restrictive. The basic and most common tasks should be exposed and obvious, but more arcane operations should still be allowed [2]. This is known as progressive disclosure [2].

The key is to prevent the user from invoking such methods by mistake. These methods can for example be stored in a separate namespace or have longer and more technical names that signal to the user that this is an operation that should be used with consideration [18].

Exceptions let clients separate error handling from normal control flow [22]. APIs should throw exceptions instead of returning insignificant values, such as NULL or random data. This causes clients to write check code [22], which is unnecessary and possibly not correctly implemented. Programming textbooks in general show many examples of triggering exceptions but few of how to handle them [17].

When there is a problem, the program should fail fast [5] with an error message that clearly describes the type of error and the last known application state [17, 19, 22, 12, 16].

Meyers [18] suggests that the most important thing about an API is that it should be difficult to misuse, because such systems are both more usable and more likely to be used correctly [18].

2.4.3 Minimally Complete

An API should be as small as possible, but no smaller [22]. It is better to have a system omit certain features and keep the design coherent than to involve many good but independent ideas [4].

It is easier to make additions and changes to a minimally complete API than a large complex one [6, 22]. A good rule of thumb is When in doubt, leave it out [4, 5, 6] and to wait until several users have independently asked for a feature before implementing it [4].

Minimally complete also means to only provide one way to perform a single task [22]. An overworked and clever interface contributes to making the code less readable and maintainable, and increases the need for documentation [4].

On the other hand, providing too little functionality causes the clients to write lots of code to perform simple tasks. Forcing clients to write boilerplate code discourages code reuse [22] and increases the risk of bugs spreading across applications.

2.4.4 Open For Extension

Over time an API can be extended in two ways: Programmers that maintain the API will add functionality in future releases and users will write subclasses to define custom behavior [4].
2.4. API QUALITIES

Encapsulation and loose coupling promote extension. Studies have found that large software systems with clearly decoupled components can be four times easier to modify than programs that are not [13].

2.4.5 Documentation

High quality documentation is important for all types of software, but it is particularly important for APIs. Frameworks often have a steep learning curve that users must overcome before they become productive [10].

Adding code examples and improving documentation text can be a way to improve an API without making changes to the implementation [9].

It is more important to document the design and purpose of code than its mechanics [3]. Documenting specific implementation details can reduce encapsulation, causing problems later. If users write code based on some part of the API implementation (e.g. specific algorithms), this cannot be changed later without breaking backwards compatibility [5].
Chapter 3

Method

Software development methodology is motivated by the expectation that, as in other engineering disciplines, performing appropriate problem analysis can contribute to the reliability and robustness of a design. Formal methods are best described as a series of steps each of which is designed to structure, plan, and control the process of developing an information system.

3.1 Defining Requirements

Good design is worth little if the API does not give users the features they need. The workflow should include collecting reliable user feedback and testing, in order to discover problems as early as possible.

Traditionally, many software developers would start by designing the core functionality and then move on to the programming interface [4, 22]. This is in many ways the wrong order because it tends to make the interface reflect the structure of the underlying code, rather than a logical abstraction of the problem [4].

The process of defining requirements is iterative. As conditions change the design should be reassessed [22].

3.2 System Design

The system design process involves capturing and taking into account the constraints that the requirements impose on the architecture. It results in a high-level object model for primary objects and processes. The high-level objects are organized in a hierarchy and with such granularity that major control flows can be identified and described.

As mentioned earlier, requirements inevitably change and therefore the design is likely to be revised. It is hard to get a system design right on the first try because as the design evolves, the designers themselves will be able to view the problem from new perspectives. By basing the design on actual concepts of the problem domain it is more likely to become general and robust [22].
3.3 Implementation

When an API is intended to replace an old one, understanding every aspect of the old implementation is crucial. Otherwise there is a great risk of substituting old design flaws for new ones [4].

It is much more preferable to have a straightforward API and a complicated implementation than the other way around [4].

Once an API is released the interface usually shouldn’t change, since backwards compatibility must be maintained for older clients [4, 20], but the implementation is likely to change with future bug fixes, workarounds and optimizations [4, 22].

3.4 Testing

For reusable components, which may be used in many different applications, the potential consequences of incorrect behavior are even more serious than for application-specific components [17]. Studies have shown that correcting defects later in the development cycle is more expensive than addressing them earlier [22].

How a component will be tested should be considered early on during its development [16, 22] and automated tests should be written in parallel with, or even before, writing implementation code [16]. Like any other requirement, test cases may change and trigger refactorization of the code and architecture [22].

To facilitate testing it is good practice to use the dependency injection pattern in conjunction with mock, fake and stub objects [22]. Mocks, fakes and stubs are placeholders for the component’s dependencies. They can be used to record data during the test and verify that the component performs the correct method calls on its dependencies [22]. For example sending the correct SQL string to a database or calling an event handler.

Structured testing verifies that the program behaves according to expectations and intentions. If that behavior is changed inadvertently, a failing test will warn the programmer. This makes programmers more confident, because the tests will catch changes in behavior that they could not anticipate themselves [8, 16, 22].

Unit testing is a programming practice in which each unit of code is tested in isolation with controlled input. Tests are grouped in test suites, which are executed automatically. Tests should exercise general behavior as well as edge cases and error handling [4, 22].

The test suite becomes a part of the software documentation that maintenance programmers can study to gain basic understanding of how units of code work [22].

Sometimes it may be necessary to reuse code that has no test coverage at all [8, 22]. This code could become a burden to all programmers that come in contact with it [8]. For this reason Feathers coined the term legacy code, which he defined as code without tests [8, 22]. This code is hard to change or even understand because it lacks unit tests. He also notes that it is possible for a programmer to produce legacy code, simply by not writing comprehensible tests from the outset [8].
Good unit tests encourages the improvement of old code because programmers feel confident that compatibility is maintained by not changing behavior [22].
Chapter 4

Requirements

The following requirements have been raised in discussions with the group of engineers who are meant to use the tool and are in addition to the objectives outlined in the introductory chapter of this document.

4.1 Input and Output

The simulation output is different for each data generation and eye tracking algorithm. The glint data generation algorithm should output the 3-dimensional point where light is reflected in the eye surface, normal vector of the eye surface at that point and the directions of incident and reflected light. For pupillary border data generation the output should be a list of 2-dimensional points representing coordinates on the camera’s sensor plane.

For eye tracking algorithms the output should be a set of key-value tuples containing the reconstructed eye parameters.

All output should contain relevant partial results and information from each component and computational step and be presented in a human readable format that is suitable for further investigation and experimentation.

4.2 User Friendly

The API is intended for rapid prototyping and evaluation of potential eye tracking systems, therefore it must be easy to use. The user interface should be focused around constructing a simulation model and not depend on the users ability to understand the underlying implementations or procedures to obtain simulation results.

Since the user should not need to understand any part of the implementation, errors must be handled gracefully and error messages be clear and informative. Errors that are caused by users must have error messages that helps the user correct that error. When an error originates from inside the API it must contain information so that a software developer can trace the bug.
The program is not intended to be used outside of the R&D department so the user group is well defined. The design of the logical interface should be aimed towards these people.

The API should use coherent naming conventions that correspond to the terminology used in the field. Some terms used by the Tobii R&D team are specific to Tobii products and traditions. We try to use these terms where there are no established alternatives. The naming convention should also adhere to that used in other Tobii software. Use concepts from the problem domain.

The API should include an optimization algorithm that can be reused in the data generation and calibration algorithms that are part of the API itself, but also available for clients. Users that are writing prototype code neither want to find nor implement such functionality since it is time consuming and error prone.

4.3 High Coding Standards

The API structure should reflect a well thought through strategy to tackle the problem. The code should be easy to read, have relevant documentation and reuse coding conventions from other Tobii Software Development Kits (SDKs).

The API should be designed as one complete unit but each component must be possible to use in isolation and independent of other components. The user should be able to use each component outside of the API and experiment with it. Modeled components should not depend on the type or implementation of each other. The relationship between components should be defined by the context they are used in. For example, the camera should not depend on whether incoming light originated from an eye, an illuminator or some user defined mechanism.

The documentation should contain a visual display of the class hierarchy. Avoid long chains of dependencies and do not accept cyclic dependencies.

4.4 Open for Extension

The API will be used to test different mathematical models for eyes, cameras and illuminators and eye tracking algorithms. The API should provide interfaces for all of these objects so that it can be extended without changes to the simulation core.

However, the design must also reflect the possibility that new requirements may arise in the future that would cause modifications of the core. The design should facilitate maintainability of the core as well as extension of the API. Future extensions of the API should be able to benefit from existing functionality.

It is preferred that the API contains fewer and simplified mathematical models and that the API can be easily extended, than have a lot of functionality that needs to be maintained or improved soon after a project deadline and software release.

Use the programming language C++ and STL types or Python.
4.5. ROBUST AND TESTED

4.5 Robust and Tested

The intended use of the API is such that errors could have a large impact on other software and future decisions based on simulation results. Tobii must be able to rely on mathematical correctness. Correctness should be verified by unit tests and robustness ensured by necessary runtime checks for bad input and exceptional conditions.

Tests and use cases should be used as documentation of program behavior as well as verification of correctness. Use cases should be verified by doing the same calculations by hand. Each unit of code that can be unit tested should be. The code should be designed for unit tests.

Tests are an important part of the deliverable since they will be vital when changes are made in the future.
Chapter 5

System Design

Given the considerations outlined in chapters 3 and 4, this chapter now describes the architecture, major components, modules, interfaces, and data for the system. A user centric application workflow was designed to facilitate an implementation that meets the specified requirements.

5.1 Application Workflow

The requirements state that the API should provide a simulation core as well as standalone components.

We start to design the core by identifying steps that take a simulation scenario from problem statement to result output. The user is not required to understand either the implemented algorithms or the mathematical models. The user should only handle very high-level abstractions of objects that correspond to real world concepts.

In this use case the simulation core should be a black box solution (facade pattern, see 2.3.3). We design a 3-step process for performing simulations (fig. 5.1):

1. User defines the physical objects and selects the algorithms.
2. Black box simulator execution.
3. Display results.

5.2 Major Components

As we review the 3-step process we find that it translates well to a set of three major modules (fig. 5.2).
5.2. MAJOR COMPONENTS

1. Geometrical models for the eyes, sensors and illuminators.

2. A set of simulation strategies.

3. A documentation unit to process simulation output.

Each component in these modules must work as a stand-alone component and the user should be able to define custom components that work seamlessly with the rest of the API modules. For this to work we rely heavily on inheritance (fig. 5.2). The simulation core will only have method definitions that take abstract or interface types as arguments.

Designing with interfaces makes it clear that the components do not depend on each other’s implementation, only the invariants specified in interface documentation. By strictly adhering to this principle we achieve encapsulation, modularity and loose coupling, making the API more extendable [13].

The fact that each component is functionally isolated so users can perform smaller experiments without engaging the entire API suggests an idea of progressive disclosure [2]. By having the most general methods specified in the API interface and always referring to objects as their interface type within the boundary of the API we keep the usability of the components high and keep them simple and intu-
itive to use. The most implementation specific functionality for each object should be specified on the subclass itself. This means that when the object is used as the general type, only the few selected methods are visible. But when it is used separately, declared as the more specific type, it will disclose all of its features.

When a user decides to extend the API by inheritance he or she should make sure that when referred to as their interface type there should be an absolute guarantee that the type behaves according to interface specifications. The user can be very careful, or not at all careful, with the type specific functionality and decide on programming practices, depending on the context that the implementation should be used in. In the same way that the API code matures so can user extensions. An implementation can be a very quick, dirty and rapid prototype at first but in order to use it with the simulation algorithms provided, only the methods defined on the general type must be refined to guarantee their respective invariants.

On another abstraction level we must also handle modularity for simulation algorithms since these should be independent and interchangeable. A possible way to deal with this is to use the Strategy pattern described by Gamma et al..

The algorithm interface requires one single method to be implemented that takes a geometrical model packed as a scene (fig. 5.2) and performs analysis on the components within that scene. Algorithms are supplied to the simulation core, which calls this method on every algorithm, once for each scene.

The interfaces let users extend the API by defining new algorithms in the same manner as geometrical models. A user does not need to know that this is an

\[\text{Figure 5.2: Distilled Class Diagram}\]
5.3. DEPENDENCIES

application of the strategy pattern because the API only exposes the algorithm interface, which does not need to have the word strategy in its name.

5.3 Dependencies

It is not feasible to have no external dependencies. Implementing new classes for unit tests and numerical calculations would take too much time and require advanced testing on those constructs themselves. There are third-party sources for this and it is left as an implementation decision to find and select one. The implementation should apply Adapter and Facade patterns to encapsulate these dependencies.

Numerical vectors and matrices are types that are likely to be used throughout the API and these will come from third-party libraries. These types should be separated from the API implementation so that they appear to be a specifically designed to be compliant with the API (adapter pattern, see 2.3.3), but in reality they are part of a well-tested third party library. If someday there are reasons for using another source, only the code inside the wrapper classes will have to change.

As long as it is correct and efficient the implementation of the third party software is irrelevant to the API implementation. This way matrices and vectors can be used throughout the API without risking future headaches.

The use of wrappers makes the API maintainable and the merits of a carefully selected third party library make it reliable. Clients will also benefit from loose coupling inside of the API since this makes it less likely that the public API interface will change in future releases.

5.4 Extendibility

Scientific programming is often focused around experimental work and therefore it must be possible to implement new ideas quickly. It is not desirable to invest a lot of time writing code that turns out not to work in the end. But when a good idea turns out to be a great one it should be easy to stabilize the code to a level where it can be used for advanced simulation and in a production environment. No user should be forced to port prototype code to fit it into the API. Especially if only a small part of the prototype is the useful one.

Based on this we make the slightly controversial design decision [22] to document and include all internal interfaces and headers in the release. This is also a wish that Tobii expresses (see 4.4).

All support and scaffolding code such as matrix and vector wrappers are public. This way, even prototyped extensions can follow the general style of the API and we reduce the workload on the user that needs to convert prototype code to production code and include it in the API.

Every part of the implementation evolves around the simulation core. After the core is implemented every interface implementation is in a sense an extension. The idea is to let interfaces evolve with the public methods that are necessary for the
utility of the object and not try to think ahead too much. This makes it more likely that the threshold to make later extensions will be low, even if they mean extending the API interfaces.

By being restrictive with what methods are included in the public interfaces we aim to create an API that is minimally complete and that follows the Principle of Least Astonishment [21], but also one that is open and encourages rapid prototyping and creative eye tracking experiments. It should naturally encourage the user to continue working in the same style of good programming practices as the API.
Chapter 6

Implementation and Results

Here we describe how practical work moved an idea from concept to reality. We tried to implement the application according to plan, model and design and can now present examples of how functionality and usability was improved by decisions based on theory and best practices for software design covered in previous chapters.

6.1 Programming Environment

6.1.1 C++ Programming Language

Tobii suggested that we use either Python or C++ as the implementation language for the eye tracking API. We decided to use C++ for the following reasons.

C++ is an Object-oriented (OO) language, which is a natural choice when designing a modular system, and it can be extended through inheritance. It provides exception handling, container classes, iterators and many of the basic algorithms and data structures of computer science.

C++ has the drawback of reduced performance with overridden [19, 22] methods but we decided that the advantages of inheritance far outweigh the drawbacks of using virtual methods.

C++ is not platform independent but there are compilers for most platforms such as GNU Compiler Collection (GCC) for various Linux distributions and Visual C++ (VC++) for Windows.

This does not mean that the API can never be used in Python or other languages in the future. In section 7.5.4 we discuss the possibility to map the C++ implementation to other programming environments. This would likely have been harder to achieve if the original implementation was in Python.

6.1.2 BOOST

For functionality that is not an integral part of the C++ language we decided to use the BOOST libraries. BOOST is a set of external libraries for C++ that include support for linear algebra and unit testing.
According to Reddy [22] dependencies to BOOST libraries are generally manageable for clients, since dependencies to BOOST are so common that clients are likely to have them anyway.

The BOOST libraries are unit tested [22] which means that we do not introduce legacy code [8]. The downside is that BOOST libraries must be available at compile-time and if the BOOST interfaces change, parts of the API must also be refactored.

**uBLAS**

Basic Linear Algebra Subprograms (BLAS) is an API standard for publishing libraries to perform basic linear algebra operations such as vector and matrix multiplication. The BOOST implementation of the BLAS interface is called uBLAS and extends the concepts of the STL vector class in a suitable way for numerics.

The library covers basic linear algebra operations on vectors and matrices: norms, addition and subtraction of vectors and matrices and multiplication with a scalar, inner and outer products of vectors, matrix-vector and matrix-matrix products.

We have chosen to use the vector and matrix types from uBLAS. We also make use of uBLAS multiplication operators. To isolate dependencies the types vector and matrix are wrapped inside API specific vector and matrix wrapper classes (see 2.3.3).

**BOOST Test Library**

The BOOST Test Library provides a set of components for writing test programs, organizing tests in to simple test cases and test suites, and controlling their runtime execution.

We used the BOOST Test Library for unit testing the API implementation.

### 6.2 The Optimization Problem

The eye is often described as a sphere (fig. 6.4) or a union of intersecting spheres (fig. 6.1). However, these models are simplifications and we should assume that the API could eventually be used to evaluate eye models that mimic the random imperfections of a real eye.

A recurring problem in eye model calibration is adapting a light path to generated data by reflection or refraction in different materials. In the general case there is no guarantee that an adapted light path can be found with a closed form expression, so instead the calibration algorithm must search for a minima of some function using an optimization algorithm.

The previous simulator used by Tobii (see 2.1) uses optimization algorithms provided by MATLAB. In the C++ environment we do not have this luxury. Instead we used a pre-existing implementation of the Levenberg–Marquardt Algorithm (LMA)
6.2. THE OPTIMIZATION PROBLEM

provided by Tobii. We now describe how parts of the MATLAB code was ported to C++ and modified for use with LMA and what we had to do to make it work.

6.2.1 Levenberg Merquardt Algorithm

In mathematics and computing, LMA provides a numerical solution to the problem of minimizing a function. The function can be nonlinear, as in the case of a non-smooth eye surface. We do not describe LMA in detail, only what is relevant to the specific problems that we had to solve.

LMA takes as input a value function and start guesses for each of the parameters for that function. When the algorithm terminates the parameter array contains the parameters that minimizes the value function. Since LMA only finds a local minima it is important to find good start guesses and a well behaved value function to increase the chances of converging to a global minima.

We are reusing an implementation of LMA from another application. It does not come with any unit tests, so we are introducing legacy code [8]. This is of course very bad, but we do so with Tobii’s approval and we encapsulate the code with a technique similar to the adapter pattern (see 2.3.3) so that it can be replaced without breaking any other part of the API implementation or clients.

6.2.2 Value Function Parameters

Tobii’s MATLAB simulator code (see 2.1) uses a value function with parameters \((x, y, z)\) for the three components of the directional vector of the emitted light and then returns a measure of how accurately the reflected light hits the target.

We must remember that LMA has no way of dealing with discontinuities and will alter the parameters as it sees fit. For some parameter values the light will miss the sphere entirely, causing the returned value to jump unexpectedly. Indeed the MATLAB code contains many checks to detect and manage this, but the solution is far from ideal. By changing the parameterization we can avoid this problematic situation.

We replace the parameters \(x, y, z\) with \(a, b\), such that \(a, b \in \mathbb{R}\), and create a function that produces a point on the surface of the eye \(\vec{r} = \vec{r}(a, b)\).

```c
point3d point_on_surface(double a, double b) {
    ...
}
```

Now, to select the light direction we simply draw a line from the light source \(\vec{s}\) to the reflection point \(\vec{r}\) (fig. 6.4). Instead of LMA optimizing the light direction, it finds the optimal reflection point on the sphere. The generated point may be on the back side of the sphere, viewed from the light source, but this does not matter since the ray will be reflected where it intersects the sphere surface the first time.

We can calculate the direction \(\hat{w}\) for light leaving light source \(\vec{s}\) (fig. 6.4) with equation 6.1.
The parameters $a$ and $b$ can have different interpretations for different surfaces because the search algorithm is oblivious to context. The programmer that extends the API with a new eye model can use them as they are best applied to that model. For a perfectly spherical eye it is suitable to use the parameters as radial distance and polar angle in a spherical coordinate system.

Finding the light direction this way does not affect how the measure of error is calculated, we have only made the code a little less likely to fail and easier to extend because clients do not have to write checks to exclude special cases.

### 6.2.3 Finding a Good Start Guess

The parameters $a$ and $b$ can be reused to produce good start guesses for LMA. Good start guesses are initial parameter values that direct LMA towards a desirable minima.

When performing pupil data generation light is not emitted from a point light source (as with glint data generation). Instead we let the light path start on points along the pupillary border, refract in the layers of material in the eye (fig. 6.1) and then project the light onto a sensor.

We start with the same scheme of picking a point on the surface of the eye with $\vec{r} = \vec{r}(a, b)$ and direct the light through that point but we have to extend our previous solution to handle total internal reflection. This is an optical phenomenon that happens when a ray of light enters a medium at an angle larger than a particular critical angle with respect to the normal to the surface. The critical angle is determined by the ratio of density of the materials that the light leaves and enters.

When the critical angle is exceeded light is reflected instead of refracted, as demonstrated in figure 6.1.

If LMA starts with parameters that result in total internal reflection that could lead it into a local minima very far away from the optimal one.

The solution is to implement a function that produces good start guesses for $a$ and $b$. Given the eye model and the number of desired points on the pupillary border the function can produce a reasonable initial guess for each point.

Reasonable initial guesses are $a$ and $b$ corresponding to a point on the eye surface that is intersected by a line that intersects both the eye center and the point on the pupillary border. In most cases there will be more than one such point, in those cases we choose the one that is closest to the sensor that we are projecting the pupil onto.

For a spherical eye the initial light direction will be aligned with the surface normal, so neither reflected nor refracted.
6.2. THE OPTIMIZATION PROBLEM

![Diagram of internal reflection](image)

**Figure 6.1: Internal Reflection**

6.2.4 Value Function Output

We have defined the input for the value function and managed to produce good initial values for its parameters. The last step is to find a way to express the value of different combinations of parameters. Once again we turn our eyes to the MATLAB simulation code and find equation 6.2 for calculating the measurement of error and optimization. The equation is illustrated by figure 6.2, where

- \( \bar{s} \) = light source
- \( \bar{t} \) = light target
- \( \bar{p} \) = virtual light target
- \( \hat{v} = \frac{\bar{p} - \bar{s}}{||\bar{p} - \bar{s}||} \) = light direction
- \( \hat{v}' = \frac{\bar{t} - \bar{s}}{||\bar{t} - \bar{s}||} \) = optimal light direction

\[
e_1(\hat{v}) = ||\bar{t} - \bar{s} - ((\bar{t} - \bar{s}) \cdot \hat{v})\hat{v}||
\]  

(6.2)

A virtual target \( \bar{p} \) is created on the line that coincides with the directional vector \( \hat{v} \) so that \( \bar{s}, \bar{t} \) and \( \bar{p} \) form a right-angled triangle with \( \bar{t} - \bar{s} \) as the hypotenuse. The measure of error is \( ||\bar{t} - \bar{p}|| \). The measured error is zero when the light direction is optimal (equation 6.3). Notice, however, that \( e_1 \) disobeys mathematical intuition by not considering the direction of light (fig. 6.2). The value function evaluates to zero when light is pointing directly away from the target (equation 6.4). So \( e_1 \) has two local minimas and from a bad start guess LMA could converge to a solution at the reverse optimal light direction.


The MATLAB simulator code checks the light direction and attempts to add large constants to the value function and randomly make new start guesses. This is a risky strategy since there is no new information to improve start guesses on and it is uncertain if the checks will catch if the light is reflected instead of refracted.

\[ e_1(\hat{v}') = 0 \]  \hspace{1cm} (6.3)
\[ e_1(-\hat{v}') = \|\vec{t} - ((\vec{t} - \vec{s}) \cdot -\hat{v}) - \hat{v}\| = 0 \]  \hspace{1cm} (6.4)
\[ e_1(\hat{v}') = e_1(-\hat{v}') \]  \hspace{1cm} (6.5)

Instead, we devise a new way to measure inaccuracy (equation 6.6) that is more suitable to use with LMA.

\[ e_2(\hat{v}) = \|\vec{t} - (\vec{s} + \|\vec{t} - \vec{s}\|\hat{v})\| \]  \hspace{1cm} (6.6)

Figure 6.3 demonstrates how \( e_2 \) creates a virtual target \( \vec{p} \) on the extension of the directional vector \( \hat{v} \) so that the points \( \vec{s}, \vec{t} \) and \( \vec{p} \) form an isosceles triangle with \( \|\vec{t} - \vec{s}\| = \|\vec{p} - \vec{s}\| \). The value of \( e_2 \) is the distance \( \|\vec{t} - \vec{p}\| \).

This value function also evaluates to zero when the direction of light is the optimal \( \hat{v}' \) (equation 6.7) and when light is travelling away from the camera the error is strictly greater than zero. The highest possible value of \( e_2 \) is for the reversed optimal light direction \(-\hat{v}'\) (equations 6.8). From \( e_2(-\hat{v}') \), the function is decreasing for all changes in light direction.
6.3 Rotating Objects in Space

To make the API more intuitive we reviewed naming and parameter types for several functions in the MATLAB simulator. One of these was the code that the user called

\[
e_2(\hat{v}) = 0
\]

(6.7)

\[
e_2(-\hat{v}') = 2 * \|\bar{t} - \bar{s}\|
\]

(6.8)

Normalizing a vector requires finding a square root, which is a computationally intensive task. Equation \(e_2\) involves two normalizations and \(e_1\) only one; therefore \(e_2\) is slower than \(e_1\) to compute. We believe that \(e_2\) eliminates so many special cases that this loss of performance can be disregarded.

Figure 6.4 shows how the value function works in context. LMA searches for a minima of \(e_2\) by adjusting the parameters \(a\) and \(b\). The function \(\bar{r}(a, b)\) returns a point on the surface of the eye. The point maps to light directions \(\hat{w}\) originating in \(\bar{s}\) (equation 6.1). \(\hat{w}_{refl}\) is the reflection of \(\hat{w}\) in the spherical surface at \(\bar{r}\). Equation 6.9 implicitly constructs the virtual target \(\bar{p}\) and computes the distance between \(\bar{p}\) and the real target \(\bar{t}\).

\[
e_2(\hat{w}_{refl}) = \|\bar{t} - (\bar{r} + \|\bar{t} - \bar{r}\|\hat{w}_{refl})\|
\]

(6.9)

When \(e_2\) is minimized LMA has found such values for \(a\) and \(b\) that \(\|\bar{t} - \bar{p}\| < \epsilon\) for a sufficiently small \(\epsilon\), depending in size of the computer hardware architecture.
to rotate components in the geometrical model. We decided to replace an existing rotation matrix implementation with an axis-angle representation as a usability improvement.

### 6.3.1 Rotation Matrix

The following pseudo code demonstrates how a vector is rotated in the MATLAB simulation environment.

```pseudo
def create_rotation_matrix(x, y, z):
    Rx = []  # around x-axis
    Ry = []  # around y-axis
    Rz = []  # around z-axis
    return Rz * (Ry * Rx)
```

```pseudo
R = create_rotation_matrix(1, 2, 3);
v = [4, 5, 6];
u = R * v;
```

The intent of this code is to create a rotation matrix that rotates a vector $x$ radians around the x-axis, $y$ radians around the y-axis and so on. There are several issues with this code.

- It is unclear which axes the matrix rotates vectors around. The axes $x$, $y$ and $z$ may be the most obvious choice but there is nothing preventing the code from using different axes.

![Figure 6.4: Reflection in a Spherical Surface](image)
6.3. ROTATING OBJECTS IN SPACE

- The user does not know if the parameters are angles or if they should be given in degrees or radians.
- The method signature does not communicate in which order the rotations are performed.
- It is easy to mix up the order of the arguments since they are of the same type. For example listing the angle of rotation around the y-axis first.
- The actual matrix-vector multiplication is in the client code.
- It is difficult for the user to visualize the result of three rotations performed after each other.

The code is confusing for anyone trying to maintain the API as well. Notice that the arguments are given in alphabetical order from left to right. This seems most intuitive and is an (weak) indication that the rotation matrix will rotate the vector around the x, y and z-axes in that order. But the matrix-matrix multiplication is in order $R_zR_yR_x$, opposite of the argument list. This multiplication order rotates vectors around the x-axis first and the z-axis last.

The multiplication $R_xR_yR_z$ consequently rotates around the z-axis first and the x-axis last. It may not be obvious that this affects the final result, as it is very hard to visualize composite rotations, but matrix multiplication is a noncommutative operation.

$$R_xR_yR_z \neq R_zR_yR_x$$

The multiplication order may be wrongly spotted as an error and changed by a programmer trying to modify the API. In our case this mistake would be caught by unit tests but it is not good to keep code that is confusing or incomprehensible to other programmers.

6.3.2 Axis-angle Representation

An alternative to rotation matrices is an axis-angle representation, useful to compute rotation matrices and to perform rotations in space with the Rodrigues’ rotation formula.

If $\vec{v}$ is a vector in $\mathbb{R}^3$ and $\hat{k}$ is a unit vector describing an axis of rotation about which we want to rotate $\vec{v}$ by an angle $\theta$ the Rodrigues formula is:

$$\vec{v}_{rot} = \vec{v}\cos\theta + (\hat{k} \times \vec{v})\sin\theta + \hat{k}(\hat{k} \cdot \vec{v})(1 - \cos\theta)$$

We suggest that representing rotations on an axis-angle format addresses the issues listed for a rotation matrices. Consider the following pseudo code:

```cpp
vector rotate(vector v, unit_vector around_axis, double radians) {
    ...
```
vector v(1, 2, 3);
unit_vector around_axis(0.45584, 0.56980, 0.68376);
double radians = 3.1416;

vector u = rotate(v, around_axis, radians);

- It is clear from the parameter names that the rotation is performed around $\hat{k}$ and that the unit of $\theta$ is radians.
- Since only one rotation is performed the order is trivial.
- Each parameter has a different type, making it impossible to list them in the wrong order.
- The rotation is performed by the API code and not in the client code.
- It is easier to visualize one rotation than several chained rotations.

Good naming is an important usability issue (see 2.4.1). The last line of code clearly describes, almost in plain English, its own behavior.

An extra advantage of the axis-angle implementation is that the axis parameter can be specifically typed as a unit vector, subclass of a vector base type. As a consequence the user faces a compiler error if $\|\hat{k}\| \neq 1$, the API fails fast (see 2.4.2) under such conditions and it is impossible to misuse in this way.

The compiler error is a better solution than throwing an exception since the possible error is static and can be detected before runtime.

An alternative implementation choice could be to use the same parameter type for both the axis of rotation and vector. The method could accept that $\|\hat{k}\| \neq 1$ and perform the rotation around $\hat{k}' = \frac{\hat{k}}{\|\hat{k}\|}$ instead. However, there is a possibility that the user specified some other non-normalized axis by mistake and in that case we would enable the user to make that mistake. We choose to only accept a normalized axis of rotation to force the user to think about which vector is rotated around which.

Using strong static typing and different types for each argument and a more comprehensible transformation reduces the possibility of human error and makes the code difficult to misuse (see 2.4.2).

Rotation matrices are convenient for vector transformations and suitable to perform multiple rotations as series of matrix multiplications. This would be an argument in favor of rotation matrices, but the API never performs series of rotations on the same vector. To orient a component in the simulated world only one rotation is necessary. We actively remove functionality that is not necessary in order to keep the API minimally complete (see 2.4.3).

We will now demonstrate how the axis-angle representation is used in practice. For the purpose of directing rays of light we decide to use normalized vectors, which
simplifies the procedure. We want to transform the directional vector $\vec{v}$ into $\vec{v}_{\text{rot}}$, where $\|\vec{v}\| = 1$ and $\|\vec{v}_{\text{rot}}\| = 1$. The unknown variables are $\hat{k}$ and $\theta$. Calculate and normalize the bisection between $\vec{v}$ and $\vec{v}_{\text{rot}}$ (equation 6.10) and let $\theta = \pi$ radians. The transformation is a rotation of $\vec{v}$ half a revolution around $\hat{k}$.

$\theta$ and $\hat{k}$ are used to manage relative light directions depending on how objects are positioned and rotated.

\[
\hat{k} = \left\| \left( \frac{\vec{v}}{\|\vec{v}\|} + \frac{\vec{v}_{\text{rot}}}{\|\vec{v}_{\text{rot}}\|} \right) \right\| \tag{6.10}
\]

\[
\theta = \pi \tag{6.11}
\]

### 6.4 Tests and Documentation

Throughout the implementation process, the idea of complete unit test coverage has been a guiding philosophy. Every unit of code is tested on a very fine level of granularity and the tests are considered an equally important part of the implementation as the production code itself.

Every class is implemented in such a way that its dependencies are parameters to the constructor (dependency injection pattern, see 2.3.3). Together with mock objects we have been able to test each class in isolation to exercise both general behavior and edge cases.

Every class has a corresponding test class in a separate test library and all public methods are covered by tests exercising both general behavior and edge cases. In many cases calculations have been performed by hand and in MATLAB to verify test results and ensure correctness (see 4.5).

To assist future programmers seeking information or recreating some runtime condition the test code should be easy to maintain and understand. We have strived for good naming (see 2.4.1) and intuitive grouping of test cases by component. The test code have been kept minimal but complete and meet the highest coding standards as far as possible (see 4.3).

Unit tests are an important part of the API documentation (see 4.5). The target audience of this API is programmers who will continue our work. The tests can in many cases act as templates or examples for future extensions.

Other documentation are header files and implemented use cases. Together with the API we have written a client program that uses the API to perform simple but realistic simulations. The client program is bundled with the API upon delivery and is well documented with explanatory comments and build scripts.

After a user has studied the client program, header files and when applicable the unit tests it should be possible to use the API with only the documentation contained in the header files as guidance.
Chapter 7

Conclusions

This final chapter compares the project outcome to the requirements. We discuss the strengths of the final implementation and possible improvements that could be implemented in future versions.

7.1 Usability

The application workflow and user interface was detailed in close collaboration with the developers in Tobii’s R&D department and was designed before we familiarized ourselves with the mathematics involved. This contributed to an interface that hides the underlying complexity well. The gap between abstraction and implementation was bridged later to preserve the integrity of the interface.

In order to keep technical code maintainable it was natural to put a strong focus on a consistent naming convention and finding a reasonable level of abstraction. This liberates the programmer from understanding complex mathematics, which in turn reduces possible mistakes and errors. Rethinking implementation details reduced the number of cases where errors can occur (see 6.2.2) and the likelihood of user mistakes (see 6.3).

As a separate exercise we have tried to adapt the syntax so that the code reads as close to natural language as possible.

The use case application is a complete demonstration of the API capabilities, yet it is compact and descriptive of the intentions of the API.

We have found that the relative position and orientation between objects in space is what causes the most confusion for users. All geometrical models and algorithms use coherent variable names and straightforward ways of dealing with relative directions and positions.

Unit tests are used to verify that exceptions are thrown when appropriate. Rather than throwing exceptions in specific bad states, the API tries to detect individual factors that break assertions. This way we can present more detailed error messages, informing users exactly what condition triggered the exception.

Output is presented on a text format that can be copied into MATLAB which is
a good tool for continued experiments with simulation results or comparing results generated by different geometrical models.

The API works well for rapid prototyping but it is impractical to describe simulations in C++ code since it requires hard coding values and compiling each simulation setup. This is an unwanted overhead and we suggest two possible ways around this in sections 7.5.4 and 7.5.5.

### 7.2 Functionality

In section 1.5 we set out to match the data generation capabilities of the MATLAB simulation environment and to perform eye model calibration based on glint data. We left it as a task outside of the project to interpret data generated from the subject’s pupils.

The three obligatory tasks are complete, unit tested and integration tested by use cases. The unit tests show the individual parts of the API working in isolation and put invariants on each component. Use cases demonstrate how the API can be used on a high level of abstraction, which is closer to the user.

The use cases are sufficiently complex to demonstrate that the tool is capable of performing data generation and eye model calibration. During a demo session for the Tobii R&D team the use cases received positive feedback for demonstrating the purpose and potency of the tool. The use case application is available in appendices A and B.

Of course, without the possibility to perform calibrations from generated pupil data the API is not complete and its use restricted. The verbal handover of the tool to Tobii focused heavily on how the API is prepared for this extension and recommendations for how to merge it gracefully with the overall abstraction. This matter is discussed further in sections 7.4 and 7.5.1.

### 7.3 Robust and Tested

Only the test of time will show how robust the API really is. We have taken every precaution to make the code fault tolerant with exception handling and extensive automated testing.

As a principle, every time a problem was found it has been corrected and converted into a unit test to prevent the problem from arising again. The test suite has become a knowledge base of possible mistakes and how to detect them.

We have strived to keep the unit tests relevant and minimally complete. Upon delivery the API code itself spans 2963 lines of code and the test suite contains 150 individual tests, which extends over 2653 lines of code. That is 47% of the entire deliverable. Of course, the number of tests has nothing to do with how relevant they are.

Along the way we have performed a number of major refactorizations of the API code, such as replacing matrix and vector classes with wrappers around BOOST
implementations. We have also performed major reviews of naming and changed the names of several classes and variables to make the terminology more coherent. We feel that we could make these changes with less effort than we could have done without good unit tests and ensure that there were no changes in behavior.

An indication that the algorithms are working correctly is that we are able to perform eye model calibration from data that was generated by the API itself. The chances that the same error would appear in algorithms working in both directions are very small. The result of an example simulation is available in Appendix C.

### 7.3.1 Dependencies

Dependencies are possible weaknesses, especially when they are hard to test. The API implementation relies heavily on correctness of the BOOST libraries. The matrix and vector classes are the most commonly instantiated types in the entire application and the BOOST test libraries are solely responsible for verifying correctness of the API. A problem with any of these dependencies would question any simulation result obtained from the API.

If a problem in the numerical libraries was to be discovered they could be replaced without much more work than finding replacements. A problem in the unit test framework would require much more work and probably require rewriting the entire test suite for some other test framework. The probability of a bug being present in any of these dependencies is low since using them is an industry standard anyway [22].

Whether it is safe to use the legacy Levenberg–Marquardt Algorithm code is debatable. It is not covered by any tests and the original author of the code is unknown. Its only merit is that it is used in several other applications and produces good results. It is not written in a way that follows any of the principles or qualities mentioned in chapter 2. Understanding the algorithm itself is challenging and the value functions that it operates on are not trivial with hard to anticipate local minima as we discovered in section 6.2.4. From an engineering point of view, and backed up by literature [8, 16, 22], this code should be either formally verified as correct or replaced.

### 7.4 Open For Extension

Thanks to the top-down development process the implementation phase was launched with a well-defined application workflow (fig. 5.1). This led to early interface design (fig. 5.2) and after proving that the simulator was working correctly with placeholder objects on the scene, more realistic components were added on a module-by-module basis.

The incremental additions led to a positive continuous modification process. Several changes were made to both the simulator core and public interfaces throughout the implementation phase. As overall functionality evolved, components placed new demands on each other’s functionality and this is how the API matured into
7.5. Future Work

7.5.1 Gaze Tracking

As noted the API is not complete without the possibility to calibrate the subject’s direction of gaze. Discussion with Tobii in regard to this indicated that a second data structure should be added alongside the scene object, and the middle step of the current application workflow (fig. 5.1) subdivided into two steps for data generation and data interpretation (fig. 7.1). Data generation algorithms should forward their results to interpreters through the added data structure.

A less desirable alternative is for data generators to add information, such as glints, to the scene for interpretation algorithms to fetch. The intent of the scene is to abstract the experimental setup. Writing to the scene instance during simulation violates the notion of the virtual world being static. Therefore the first alternative is favorable.

It was never intended that the API could be extended without modifications, only that modifications should be straightforward and manageable. To implement the proposed change, the internal interface for simulation algorithms must change. Existing implementations must be modified so that method signatures accept the inferred data container as an injection parameter.

The suggested modification does not change the workflow for users or clients of the simulation core, so backwards compatibility would be preserved.

7.5.2 Experimental Eye Models

A major use of the API is to study how differences between the actual eye and the eye model affect eye tracking precision. Because of this, the framework will likely
be extended with data generation eye models that are not perfect spheres or do not have a perfectly reflecting surface.

The simulator could evaluate slightly deformed models, such as ellipsoid eyes, and non-smooth layers of tissue with different refractive indices.

In addition to these refinements it is also relevant to explore certain simplifications to possibly increase computational speed. The simulator could be used to test how anatomically less accurate eye models affect eye tracking precision. Reduced time for computations could result in faster calibrations or increase the frame rate of future eye trackers.

7.5.3 Extended Sensor Models

To investigate the effects of different eye models on the calibration result the simple pinhole camera is fully sufficient, but it cannot be used to study effects of imperfections in the camera itself.

The pinhole camera class can be used as a recyclable component and lens distortion added by extending it with a wrapper class. Different types of distortion
7.5. FUTURE WORK

should be applied and parameterized by configuring the wrapper class.

Reusing the pinhole camera class as depiction mechanism has several benefits. The outer class would immediately benefit from improvements to the pinhole camera courtesy to code reuse and modularity.

The wrapper class should implement the sensor interface (fig. 5.2) so it can be added to the scene as a generalized depiction mechanism and the distorted camera injected as a constructor parameter. This way distortion can be applied to any API compliant sensor implementation to investigate how sensor distortion affects eye tracking.

Distortions that are relevant to simulate are light dispersion in the camera lens, imperfections in the lens material and sensors that are manufactured or mounted incorrectly.

7.5.4 Language Mappings

C++ was a practical choice of programming language for the API implementation, but not necessarily the best choice for implementing clients. It would be a great service if users could import the API to the language that they are most familiar with.

A language mapping describes how the types and declarations from C++ are expressed and how clients can be implemented in a particular programming language, such as Python, MATLAB or Java. Python would be a good initial choice because most of the user group has at least some experience with it. A mapping to MATLAB would strengthen the rapid prototype and computational usages of the simulator core.

BOOST.Python is a C++ library, which enables interoperability between C++ and the Python programming language. Since the dependency to BOOST already exists this could be a natural choice.

7.5.5 Text-Based API

As a complement to C++ and language mappings one could device a text-based API specification that is both human-readable and machine-readable and a simple API client that interprets simulation setups from text files and create the necessary object model dynamically.

The text parser removes all user interaction through programming language and puts all focus on the task rather than implementation. It would provide an easy way to customize the format for users that work with eye tracking algorithms on a theoretical level rather than implementing them.

There are several standardized formats available to represent data structures and objects as text, so the effort is limited to creating software for data interpretation. JavaScript Object Notation (JSON) is a text-based open standard designed for human-readable representations of data structures and objects. Despite its name JSON is language-independent. Alternatively Extensible Markup Language (XML)
defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.

7.5.6 Improved Display of Results

The API specifies an interface for the report document component (fig. 5.2) and classes are provided for console and text file output. Client applications can either use existing or custom implementations to facilitate any form of output or persistent storage of results.

If the workflow extension is implemented as suggested in section 7.5.1 the data structure (fig. 7.1) could be connected to an event provider by which clients could register callback functions to receive notifications when simulation output is produced. Providing events adds the opportunity for clients to filter data and therefore algorithms could be more generous with what and how much data they publish. More advanced output streams could process partial results from algorithms as well as final results.

If the workflow is not modified, the display of results could be improved by a composite report implementation. A composite object acts as a single report instance but distributes data to several recipients. It is an elegant way to publish data in the console and in a text file at the same time.

7.5.7 Multithreading

The emphasis of this project has deliberately been on functionality and usability, not performance. The performance aspect has not been neglected, as we have followed best practices whenever we can, but there are plenty of improvements to be made.

One optimization that should be straightforward and yet possibly very powerful is to parallelize calculations using multiple threads to enable parallel execution on a multiprocessor system.

The pupil data generation algorithm projects points at the pupillary border onto the sensor plane and conducts one LMA optimization per point. The default number of points per pupil is 100 and it is perfectly reasonable to use more than that.

Currently the points are produced sequentially, but the calculations are inherently independent of each other and might just as well be performed in parallel. The performance benefits would most likely be noticeable.

Once again BOOST provides all the necessary tools. The BOOST thread library enables parallel program execution. It provides classes and functions for synchronizing data between threads or providing private copies of data specific to each thread.

If the LMA implementation is ever to be replaced, it should be investigated if the replacement can utilize parallel computation as well since this would be beneficial to all computational components in the API.
7.6. **Final Words**

During this project we have worked with methodological principles for software design in order to reduce the number of bugs and increase maintainability of software and APIs.

Working with techniques such as encapsulation and coupling has increased our incentive to educate ourselves and actively start searching for information on how to work proactively with other techniques to improve the quality of our work. Creating mathematical models that must obey the laws of nature has been an excellent opportunity to work with successive refinement and complex class structures.

It has been a great lesson to create an API instead of a desktop application. Because the code itself is the deliverable it feels more important to deliver high quality code and clever solutions than much functionality. Working with public interfaces that are going to be viewed by fellow programmers made us begin to examine the quality of our own work to a greater extent.

We have learned great lessons about the usefulness of test suites and the art of writing good unit tests.

We are very happy that Tobii provided us with such an interesting project to work on and hope that it will come to good use in the future.
Bibliography


Appendix A

Data Generation Example

This is an example of a C++ class that describes a geometrical model and applies data generation algorithms to it.

/*
 * DataGeneration.cpp
 * Created on: Nov 26, 2012
 * Author: Tobias Widén
 */

#include "Beagle.hpp"
#include "DataGeneration.hpp"

namespace Beagle {
namespace Client {

void DataGeneration::runUsecase() {
    // Sensor properties
    double focalLength = 1;
    Algebra::Rectangle sensorSize(5, 5);
    Algebra::Point3D camera1Center(300, -200, 0);
    Algebra::Point3D camera2Center(-300, -200, 0);

    // Create sensors
    Model::PinholeCamera camera1(
        focalLength, sensorSize, camera1Center, "Camera_1");
    Model::PinholeCamera camera2(
        focalLength, sensorSize, camera2Center, "Camera_2");

}
APPENDIX A. DATA GENERATION EXAMPLE

// Illuminator properties
Algebra::Point3D illuminator1Center(300, 200, 0);
Algebra::Point3D illuminator2Center(-300, 200, 0);

// Create illuminators
Model::PointLightSource illuminator1(illuminator1Center, "Illuminator_1");
Model::PointLightSource illuminator2(illuminator2Center, "Illuminator_2");

// Eye properties
Algebra::Point3D eye1Center(29.9, 0, 300);
Algebra::Point3D eye2Center(-29.7, 0, 300);
Algebra::UnitVector rotationAxis = Algebra::UnitVector::xAxis();
double angle = M_PI;
double eyeRadius = 12;
double pupilRadius = 2;
double corneaThickness = 0.5;
double refractiveIndexEye = 1.333;
double refractiveIndexAir = 1.0;

// Create eyes
Model::SphereEye eye1(eye1Center, rotationAxis, angle, eyeRadius, pupilRadius, corneaThickness, refractiveIndexEye, refractiveIndexAir, "Eye_1");
Model::SphereEye eye2(eye2Center, rotationAxis, angle, eyeRadius, pupilRadius, corneaThickness, refractiveIndexEye, refractiveIndexAir, "Eye_2");

// Create scene
Model::Scene scene;
scene.add(camera1);
scene.add(camera2);
scene.add(illuminator1);
scene.add(illuminator2);
scene.add(eye1);
scene.add(eye2);

// Create simulator
Simulator simulator;

// Run simulation
simulator.registerScene(scene);
simulator.runSimulations();
}
Appendix B

Data Interpretation Example

This is an example of a C++ class that takes a geometrical model and performs eye tracking simulation, estimating the eye's position and radius.

```cpp
/*
 * DataInterpretation.cpp
 * 
 * Created on: Nov 26, 2012
 * Author: Tobias Widén
 */

#include "Beagle.hpp"
#include "DataInterpretation.hpp"

namespace Beagle {
namespace Client {

void DataInterpretation::runUseCase() {

    // Camera properties
    double focalLength = 1;
    Algebra::Rectangle sensorSize(5, 5);
    Algebra::Point3D camera1Center(300, -200, 0);
    Algebra::Point3D camera2Center(-300, -200, 0);

    // Create cameras
    Model::PinholeCamera camera1(
        focalLength, sensorSize, camera1Center, "Camera_1");
    Model::PinholeCamera camera2(
        focalLength, sensorSize, camera2Center, "Camera_2");
```
APPENDIX B. DATA INTERPRETATION EXAMPLE

// Illuminator properties
Algebra::Point3D illuminator1Center(300,200,0);
Algebra::Point3D illuminator2Center(-300,200,0);

// Create illuminators
Model::PointLightSource illuminator1(illuminator1Center, "Illuminator_1");
Model::PointLightSource illuminator2(illuminator2Center, "Illuminator_2");

// Create scene
Model::Scene scene;
scene.add(camera1);
scene.add(camera2);
scene.add(illuminator1);
scene.add(illuminator2);

// Add glints from data generation
Model::Glint glint1(3.40033,1.81291, &illuminator1);
scene.addGlintForSensor(glint1, camera1);
Model::Glint glint2(3.44019,1.80676, &illuminator2);
scene.addGlintForSensor(glint2, camera1);
Model::Glint glint3(1.35704,1.80447, &illuminator1);
scene.addGlintForSensor(glint3, camera2);
Model::Glint glint4(1.40033,1.8149, &illuminator2);
scene.addGlintForSensor(glint4, camera2);

// Create simulator
Simulator simulator;

// Change to data interpretation
Analysis::GlintDataInterpreter gdi;
simulator.clearAnalyzers();
simulator.addAnalyzer(gdi);

// Run simulation
simulator.registerScene(scene);
simulator.runSimulations();
}
Appendix C

Simulation Output

This is a plot generated from one of the simulated cameras in Appendix A. The black dots are projections of points along the subject’s pupillary border. The red dots are illuminator corneal reflections captured by the camera. We can see that two reflections fell inside the pupil area and the other two on the irises.

The simulator was able to reconstruct model parameters for the eyes’ position and radius. In the table below we show the calibration results for the left eye in the plot. Unit for all values are millimeters.

<table>
<thead>
<tr>
<th>Property</th>
<th>Actual Value</th>
<th>Calibrated Value</th>
<th>Error Relative to Eye Radius</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eye Center x</td>
<td>29.9</td>
<td>29.9005</td>
<td>$4.1667 \times 10^{-4}$</td>
</tr>
<tr>
<td>Eye Center y</td>
<td>0</td>
<td>$-1.02207 \times 10^{-5}$</td>
<td>$-8.5173 \times 10^{-7}$</td>
</tr>
<tr>
<td>Eye Center z</td>
<td>300</td>
<td>300</td>
<td>0</td>
</tr>
<tr>
<td>Eye Radius</td>
<td>12</td>
<td>12</td>
<td>0</td>
</tr>
</tbody>
</table>