University of Southern Queensland
Faculty of Health, Engineering & Sciences

Low-Cost Eye Tracker

A dissertation submitted by

Thomas J. Bradford

in fulfilment of the requirements of

ENG4112 Research Project

towards the degree of

Bachelor of Engineering (Computer Systems)

Submitted: October, 2014

Abstract

Eye trackers have useful applications in numerous industries and fields of research. However, while commercial eye trackers currently exist for these applications, they are exceedingly expensive, limiting their application to high-end speciality products and thus making them unsuitable as a low-cost solution.

For this reason, a low-cost eye tracker utilising open-source software would greatly increase the accessibility of eye trackers to those who would benefit from the technology. Modern technology advancements have enabled off-the-shelf video hardware such as computer webcams and consumer video cameras to be suitable for use in an eye tracking hardware configuration while still maintaining their low cost and high accessibility. Furthermore, the development of open-source eye tracking software to operate in conjunction with this hardware has significantly facilitated the implementation of such hardware in the eye tracking system.

With the increasing dependence on computers and technology in everyday life, it is of increasing importance to study software usability testing and human-computer interaction to enhance the user experience. This dissertation details the development of a low-cost eye tracker using off-the-shelf hardware and open-source software to analyse how learning tools are used by students.

Throughout this process, a low-cost head-mounted eye tracking hardware configuration was designed and developed. Implementation of the hardware was achieved using the ITU Gaze Tracker software, developed by the ITU University of Copenhagen (San Agustin, Skovsgaard, Mollenbach, Barret, Tall, Hansen & Hansen 2010). The Gaze Analyser software was also developed to analyse and visualise fixations identified in the raw eye tracking data.

University of Southern Queensland

Faculty of Health, Engineering & Sciences

ENG4111/2 *Research Project*

Limitations of Use

The Council of the University of Southern Queensland, its Faculty of Health, Engineering & Sciences, and the staff of the University of Southern Queensland, do not accept any responsibility for the truth, accuracy or completeness of material contained within or associated with this dissertation.

Persons using all or any part of this material do so at their own risk, and not at the risk of the Council of the University of Southern Queensland, its Faculty of Health, Engineering & Sciences or the staff of the University of Southern Queensland.

This dissertation reports an educational exercise and has no purpose or validity beyond this exercise. The sole purpose of the course pair entitled "Research Project" is to contribute to the overall education within the student's chosen degree program. This document, the associated hardware, software, drawings, and other material set out in the associated appendices should not be used for any other purpose: if they are so used, it is entirely at the risk of the user.

Dean

Faculty of Health, Engineering & Sciences

Certification of Dissertation

I certify that the ideas, designs and experimental work, results, analyses and conclusions set out in this dissertation are entirely my own effort, except where otherwise indicated and acknowledged.
I further certify that the work is original and has not been previously submitted for assessment in any other course or institution, except where specifically stated.

Thomas J. Bradford

0050101049

_____

Signature

_____

Date

Acknowledgments

This research project would not have been possible without the invaluable assistance and support of some key people.

First and foremost, I would like to thank my project supervisor Dr. Alexander Kist for offering the project and providing timely constructive feedback and guidance, keeping me on track throughout its course. I chose the project based on its element of design, providing a challenging, interesting and rewarding experience, and it did not fail to deliver. I am very thankful for this experience.

Secondly, I would like to thank my work supervisor, Ms. Karyn Onley, as well as my colleagues at the Leslie Research Facility Queensland Grains Research Laboratory for being incredibly flexible and understanding throughout the year, allowing me to prioritise my project work and take time off whenever required. I am extremely grateful to work in such an excellent workplace.

Finally, I would like to acknowledge the unconditional support and endless encouragement provided by my family and friends. They have been by my side not only for the duration of this project, but for all six years of my studies. Without them I wouldn't be where I am now.

Thomas J. Bradford

University of Southern Queensland
October 2014

Contents

List of Figures

List of Tables

Nomenclature

API Application Programmer Interface

FPS Frames Per Second

GDI Graphics Device Interface

HCI Human-Computer Interaction

I-AOI Area of Interest Identification

I-DT Dispersion Threshold Identification

I-HMM Hidden Markov Model Identification

I-MST Minimum Spanning Tree Identification

I-VT Velocity Threshold Identification

IDE Integrated Development Environment

IR Infrared

ITU Information Technology University of Copenhagen

LED Light Emitting Diode

PCB Printed Circuit Board

RANSAC Random Sample Consensus

TCP Transmission Control Protocol

UDP User Datagram Protocol

USB Universal Serial Bus

Chapter 1

Introduction

1.1 Background

Eye tracking is the process of measuring the gaze direction of a person to determine their line-of-sight or point-of-regard. In essence, it determines where a person is looking. Eye tracking technologies have useful applications in numerous industries and fields of research such as ophthalmology, psychology and psycholinguistics, cognitive linguistics, medical research, marketing and advertising research, product design, sport psychology and research, security and law enforcement, road safety and even aviation cockpit design (Tobii Technology 2013).

More recently, applications have extended to disability support, where a patient's gaze is used as an input for human-computer interaction (HCI) (San Agustin 2009). Eye tracking technologies are also useful in usability testing and HCI research (Jacob & Karn 2003). With the increasing dependence on the use of computers in everyday lives, it is of equally increasing importance to study and analyse software usability testing and HCI in order to improve and enhance the user experience.

Despite the applications of eye tracking technologies in HCI research and usability testing, Jacob & Karn (2003) state that "The study of eye movements pre-dates the widespread use of computers by almost 100 years". In 1879, Louis Émile Javal observed that, whilst reading, the eye does not sweep continuously from left to right across the text, rather eye movements are defined as either *fixations* and *saccades*. A fixation

occurs when the eye is fixed on a point of interest, typically for a duration of at least 100 milliseconds, whereas a saccade is defined by the rapid eye movement between two fixation points (Jacob & Karn 2003). This phenomenon is displayed in Figure 1.1, with fixations represented by shaded ellipses and saccades represented by solid lines connecting those ellipses.

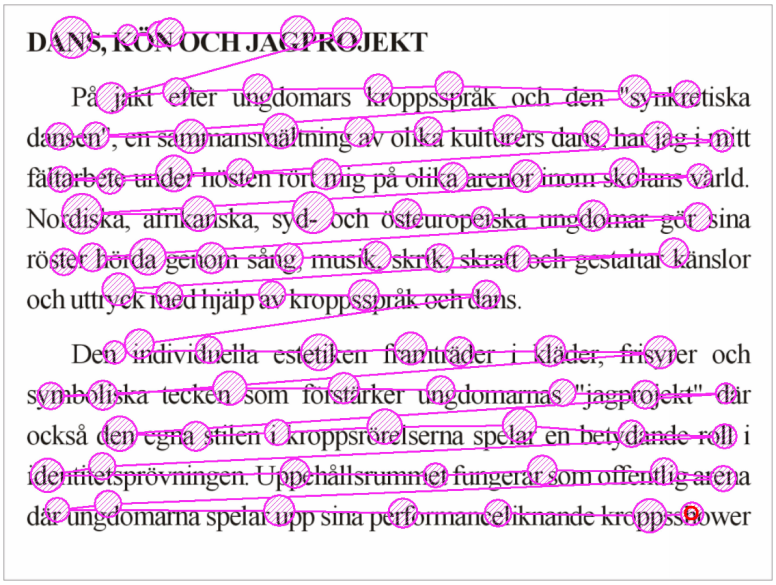


Figure 1.1: Fixations and saccades when reading text (Wikipedia 2014).

Eye tracking technologies can be applied to the use of learning tools to improve their effectiveness. In recent times, computers have become a dependency in educational environments, with students of all ages frequently required to use them for research, writing, word processing, design, calculations and learning development. Learning tools can aid students in learning new concepts or practising learnt concepts, both consciously and subconsciously.

Wang, Chignell & Ishizuka (2006) state that, “Eye movements provide an indication of learner interest and focus of attention. They provide useful feedback to character agents attempting to personalise learning interactions”. Learning tools are most effective and appealing when they are intuitive, have a well presented graphical user interface and have minimal usability issues. The analysis of the use of learning tools by students using eye tracking is beneficial in improving the effectiveness of these learning tools (Wang et al. 2006).

1.2 Motivation

Section 1.1 identified the many eye tracking applications for research; however, Johansen & Hansen (2006) questioned the need for eye trackers in their article, entitled *Do We Need Eye Trackers to Tell Where People Look?*. They concluded that the validity of user's memory of their scan paths is limited in comparison to the recorded data, thus justifying the need for further research and developments within the field of eye tracking.

While commercial eye tracking solutions currently exist, they are exceptionally expensive and relatively inaccessible. San Agustin et al. (2010) state that "Commercial gaze tracking systems have been available for more than 15 years, but the technology is still a high priced niche". Li, Babcock & Parkhurst (2006) report that "Until only recently, eye trackers were custom made upon demand by a very few select production houses. Even today, eye tracking systems from these sources range in price from 5,000 US dollars to 40,000 US dollars, and thus limit their application to high-end specialty products". Furthermore, commercially available eye trackers can be platform specific, and difficult to use. A low-cost, open source system would allow anyone to explore eye tracking in many new ways (Babcock & Pelz 2004).

A seemingly popular commercial remote eye tracking solution is the Tobii X2 series from Tobii Technology, a company specialising in eye tracking and gaze interaction technology. As of March 2013, the Tobii X2-30 can be hired for 1,100 Australian dollars per month, or purchased outright for 20,000 Australian dollars, while the Tobii X2-60, with superior performance characteristics such as a higher degree of accuracy, faster refresh rate and reduced system latency can be hired or purchased for more than twice these respective prices. These prices are also non-inclusive of GST (r3dux.org 2013, Tobii Technology 2014).

From this information, it can be deduced that the high cost of commercial eye tracking systems makes them unsuitable for consumer use, and even unideal for some commercial use, particularly amongst small businesses and low-budget research corporations within the various industries that would benefit from such solutions.

Fortunately, development in camera technologies and computer peripherals over the past decade has enabled off-the-shelf hardware such as webcams and video cameras to

evolve to a point where they are suitable for eye tracking hardware implementations while still maintaining their low cost and high accessibility. This has led to a growing interest in the use of low-cost components for eye tracking systems and the emergence of low-cost eye tracking as a field of research (San Agustin 2009).

As stated in Section 1.1, the use of eye tracking technology has its applications within an educational environment. The development of a low-cost eye tracking system can greatly improve the accessibility to eye trackers in the classroom for the analysis of the use of learning tools and educational software. This analysis can lead to improvements in the user interface of these tools and in-turn improve their effectiveness and appeal to students (Wang et al. 2006).

1.3 Project Aim

The overall aim of this research project is to develop a low-cost eye tracking system that can be used to analyse how learning tools are used by students.

From the information presented in Section 1.2, it is important that this research project should have an emphasis on low-cost design and accessibility to consumers, thus the system should utilise inexpensive off-the-shelf hardware and open source software.

Both the hardware and software implementations should be designed to perform optimally, maximising the degree of accuracy of the hardware and the efficiency of the software while keeping the design simple and low-cost.

The outcome of this research project is aimed at the eye tracking system being utilised to improve the graphical user interface and thus, the effectiveness of learning tools. Furthermore, the design and development of the system can be used as a basis or resource for future research and developments in low-cost eye tracking with respect to software usability testing and HCI research.

1.4 Research Objectives

The research objectives involved with this research project form the basis of the project. Using the project aim presented in Section 1.3, the following research objectives were identified:

- Outline current eye tracking solutions, identifying their uses, cost and limitations
- Identify methods of eye tracking
- Identify techniques of analysis and visualisation of eye tracking data
- Select appropriate hardware and software for use with the low-cost eye tracking system
- Design, develop and implement the hardware and software components of the system
- Critically analyse and evaluate the system performance, optimising where necessary

The Project Specification in Appendix A presents a concise list of project objectives as identified and stipulated prior to the commencement of the project.

1.5 Overview

The remaining sections of this report are organised as follows:

Chapter 2 presents a literature review relating to and detailing numerous aspects of eye tracking, particularly with the use of low-cost off-the-shelf hardware.

Chapter 3 outlines the methodology for the development of the low-cost eye tracking system, specifying resource requirements for both the hardware and software designs.

Chapter 4 details both the hardware and software design elements of the low-cost eye tracking system, as well as the implementation of the hardware and software to form the system.

Chapter 5 evaluates the performance of the low-cost eye tracking system, recognizes limitations in the system and identifies alternative uses for the system.

Chapter 6 summarises the dissertation, verifying the achievement of objectives for the research project and outlining possibilities of further work on the topic.

Chapter 2

Literature Review

2.1 Outline

This chapter reviews literature relating to and detailing numerous aspects of eye tracking, particularly with the use of low-cost off-the-shelf hardware. The topics covered in this literature review are separated into two main sections; eye tracking and data analysis and visualisation.

2.2 Eye Tracking

Comprehensive knowledge of eye tracking and gaze estimation is necessary to form the required in-depth understanding of principles for this research project. This section discusses the fundamental concepts of eye tracking and gaze estimation, identifying methods and techniques of execution and, subsequently, the various hardware configurations and supporting software, particularly with respect to low-cost eye tracking.

2.2.1 Methods

A user's eyes can be tracked via a number of different methods and hardware configurations, with the intrusiveness and accuracy dependent on the method employed (San Agustin 2009). Historically, these methods have involved the implementation of

electro-oculography – a technique used to detect changes in electrical potential near the eye as the eye moved by placing electrodes on the skin around the eyes. A more accurate, yet more intrusive method required the use of contact lenses with an embedded coil, where changes in electrical potential as the eye moved were able to be measured when a voltage was induced in the coil with an electromagnetic field (Mantiuk et al. 2012). This required the contact lenses to be wired to an appropriate measuring device, thus making them very uncomfortable (San Agustin 2009).

A much less intrusive method of eye tracking involves the implementation of video-oculography – a technique which makes use of video capture hardware to record the gaze of the user. Image processing software is used in conjunction with this hardware to extract information about different eye features to measure the point-of-regard or line-of-sight. These eye tracking systems are arguably the most popular solution due to their non-intrusiveness while maintaining appropriate degrees of accuracy (San Agustin 2009).

Most common video-oculography eye tracking systems measure point-of-regard using the *corneal reflection* method (Goldberg & Wichansky 2003). In this method, the eye is directly illuminated by infrared light which is reflected on the retina, causing the pupil to appear as a concise disc in contrast to the surrounding iris. If the infrared light source is co-axial with the camera, the camera's sensor is able to detect infrared light reflected on the retina, which causes the pupil to appear bright (known as the 'bright pupil' effect) (Poole & Ball 2005). This phenomenon can also be seen in flash photography, when a subject's eyes can appear red if the flash is close to the camera lens. Conversely, if the infrared light source is off axis with the camera, the pupil will appear dark (Figure 2.1). In some eye tracking systems, images of both the bright pupil and dark pupil are obtained and subtracted from one another in order to detect the pupil more accurately and make the system more robust (Morimoto et al. 1999).

The corneal reflection method also involves the formation of four Purkinje images, which are reflections of the infrared light source from within the structure of the eye that are visible on the external surface of the eye (San Agustin 2009) (Figure 2.3). The first Purkinje image, also known as the corneal reflection or glint (Figure 2.2), is the reflection from the external surface of the cornea, and is generally the most visible of the four (San Agustin 2009). The three remaining images are reflections from internal surface of the cornea, and both surfaces of the lens (Mantiuk et al. 2012).

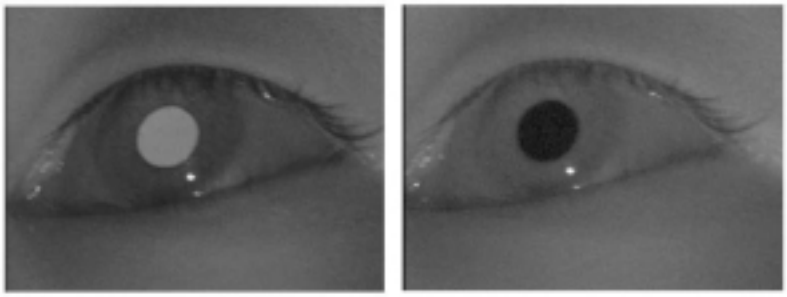


Figure 2.1: Comparison of a bright pupil (left) and a dark pupil (right) (Morimoto et al. 1999).

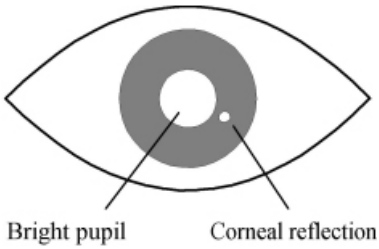


Figure 2.2: Bright pupil and corneal reflection due to infrared illumination (Poole & Ball 2005).

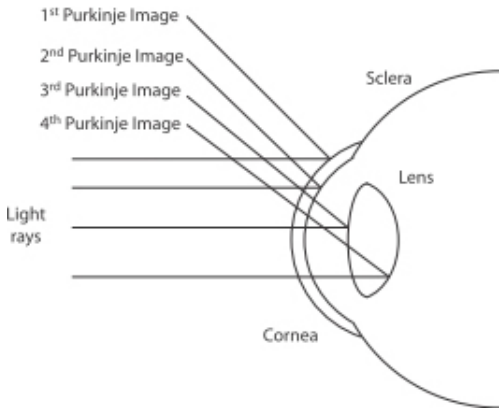


Figure 2.3: Four Purkinje images on the eye (San Agustin 2009).

Eye tracking using natural light illumination is possible to an extent, but at a loss of accuracy. As stated by San Agustin (2009), "When only natural light is available, few assumptions on the image data can be made. The iris is the most prominent feature due to its high contrast with the sclera. However, it might be partially covered by the eyelids, making its detection more difficult and inaccurate."

2.2.2 Hardware

Video-oculography eye tracking systems can usually be classified based on two hardware configurations: *remote* and *head-mounted*. Remote configurations involve both the camera and infrared light source being placed at a small distance from the user's eyes, typically near the base of the computer monitor in which the user is observing (San Agustin 2009). Remote configurations have an advantage over head-mounted configurations in that they are much less obtrusive for the user and do not obstruct the user's field of view, however, they are only able to estimate a user's gaze within a very limited area, making them less tolerant to head movement (Cooke 2005). Figure 2.4 shows a remote eye tracking system implemented by (San Agustin 2009) using an off-the-shelf video camera and two off-the-shelf infrared lights.

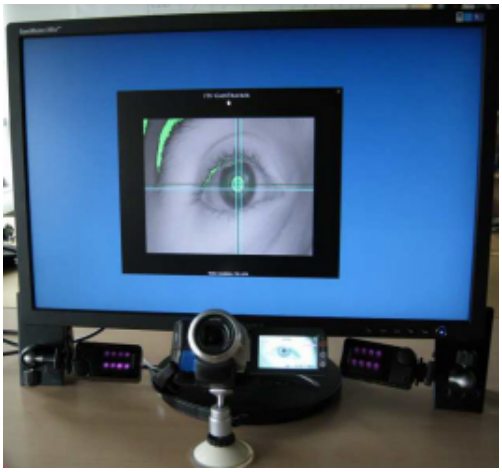


Figure 2.4: Remote eye tracking system implemented by San Agustin (2009).

Head-mounted configurations involve both the camera and infrared light source being mounted to a wearable accessory, such as extending from glasses frames or fitted to a helmet or hat. Depending on the application, some head-mounted configurations may also utilise a scene camera to record the user's field of view (Babcock & Pelz 2004). Contrary to remote configurations, the portable nature of the contraption allows it to be used for mobile eye tracking applications (Cooke 2005). However, head-mounted configurations can be somewhat obtrusive and distracting for the user due to the placement of components in front of the eyes and may also cause physical discomfort if worn for long periods of time. To reduce these effects, it is ideal that the hardware components be as small and lightweight as possible (San Agustin 2009). Figure 2.5 shows a head-mounted eye tracker built by Mantuik et al. (2012) using a modified low-cost off-the-shelf webcam surrounded by three infrared LEDs for illumination. The webcam

and LEDs are mounted to the frames of some safety glasses.

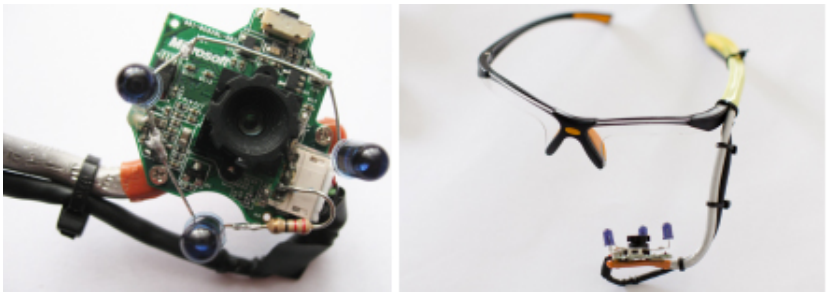


Figure 2.5: Head-mounted eye tracking system constructed by Mantiuk et al. (2012).

In order to capture the bright pupil and corneal reflection under infrared illumination, the video recording hardware must be sensitive to infrared light (Mantiuk et al. 2012). The sensor must also be able to capture video at an appropriate resolution to produce suitably detailed images of the eye for processing. The required resolution is dependent on whether a remote or head-mounted system is used, as the detail of the image of the eye is dependent on distance between the camera and the eye.

Off-the-shelf webcams without an infrared blocking filter can generally be used in head-mounted configurations. San Agustin et al. (2010) state that "Standard webcams usually have a low resolution and a broad field of view, but by placing the camera close to the user's eye we can obtain images of sufficient quality". Most off-the-shelf webcams contain an infrared blocking filter behind the lens, so some modification to remove this filter may be required to enable the webcam to detect infrared light. Furthermore, it may be necessary to insert a visible light blocking filter in its place to remove undesirable corneal reflections or interference caused by room lighting. Mantiuk et al. (2012) reported that a piece of developed analogue camera film can be used as a visible light filter due to its similar spectral characteristics to a properly designed infrared pass filter. Other sources have claimed that a piece of the magnetic layer of a floppy disc can also be used for this purpose (FreeTrack Forum User Gian92 2012).

In remote configurations, a video camera with optical zoom capabilities is more effective than a webcam due to their typically higher resolution and ability to zoom into the user's eye. It is desirable that the video camera also have a night shot mode to make it sensitive to infrared light without modification, as video cameras are typically more expensive than webcams and modifications will likely void the manufacturer's warranty or even damage the hardware (San Agustin 2009).

Generally, most off-the-shelf webcams do not have built-in infrared light sources. Some video cameras may contain small infrared light emitters, but they may not be bright enough to create a corneal reflection on the eye for remote eye tracking applications (San Agustin 2009). Therefore, the use of external infrared LEDs or illuminators is usually required in both remote and head-mounted configurations to sufficiently illuminate the eye and create a corneal reflection.

2.2.3 Software

Image processing software is required to be implemented in conjunction with the hardware to extract information about the eye features and calculate the approximate gaze position of the user. Generally utilising pupil detection and the corneal reflection method, the algorithms within the software measure the relative movement of the pupil and corneal reflection. This generally requires a calibration procedure where the user looks at a set of target points on the screen to establish a mapping between the pupil-glint vector and the corresponding screen coordinates (Mantiuk et al. 2012).

Morimoto et al. (1999) established an eye tracking technique using the corneal reflection method. In their technique, the surface of the eye is approximated by a sphere. With the infrared light source fixed, the corneal reflection can be taken as a reference point. After a calibration procedure, the vector from the corneal reflection to the centre of the pupil will describe gaze direction. A block diagram of this procedure is shown in Figure 2.6.

After estimating the gaze direction, it is important to map the corresponding screen coordinates to establish where the user's gaze is positioned on the screen. In the technique proposed by Morimoto et al. (1999), this is achieved using a simple second order polynomial transformation computed from the initial calibration procedure. In the calibration procedure, the user is prompted to fixate their gaze towards a number of target points. For each point, the vector from the center of the pupil to the corneal reflection is used to determine the coefficients in the polynomial transformation by interpolation, which is then solved to obtain useful calibration information.

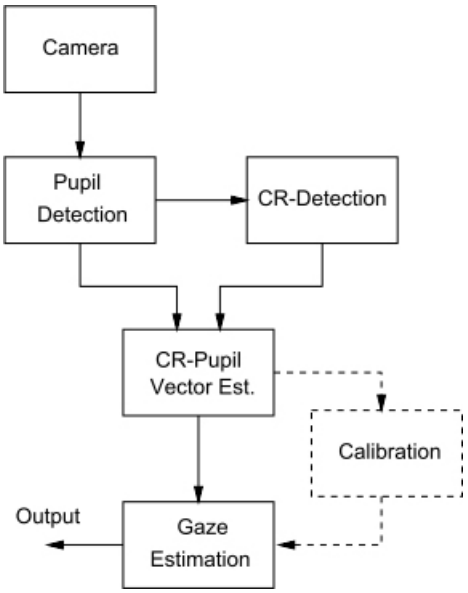


Figure 2.6: Block diagram of the corneal reflection eye tracking system as proposed by Morimoto et al. (1999).

ITU Gaze Tracker

In order to provide an accessible, low-cost eye tracking system compatible with off-the-shelf hardware, some researchers and software developers have worked towards developing their own eye tracking software to benefit both end-users and researchers alike. A notable example of this is the ITU Gaze Tracker, developed by researchers in the ITU GazeGroup at the IT University of Copenhagen in Denmark and released as open-source in 2009 (San Agustin et al. 2010).

The ITU Gaze Tracker was developed with the following design considerations (San Agustin et al. 2010):

1. The gaze tracker should be robust and accurate enough to work with at least one gaze-communication system.

2. Use of low-cost off-the-shelf components. The hardware employed should be available in any electronics store or at online shops to allow for easy acquisition and replacement. Furthermore, no hardware moderations should be needed.

3. The user should be given flexibility to place the different components (camera, infrared lights, computer display) at various locations to fit specific needs. For instance, mounting the display on a wheel chair, moving it to a table or having

it accessible in bed should make no difference.

4. Open-source software. Developing an efficient gaze tracker from low-cost components is a huge endeavour. No single group of developers is likely to come up with the ultimate solution. Open source allows anybody to improve and modify the source code to fit specific needs.

The ITU Gaze Tracker runs on the Microsoft Windows operating system and supports both remote and head-mounted hardware configurations, making it easily accessible and versatile with low-cost off-the-shelf hardware. It is developed in C# and utilises OpenCV, a vision library developed by Intel and free to use under the open source BSD licence, for image processing (San Agustin et al. 2010).

Both remote and head-mounted configurations are supported by the ITU Gaze Tracker. For remote configurations, the software implements both pupil detection and the corneal reflection method for increased robustness and head movement tolerance in the X and Y plane. For head-mounted configurations, only pupil detection is utilised, therefore any head movement will decrease the calibration accuracy.

Detection of the pupil in the ITU Gaze Tracker is performed by the OpenCV package, with the algorithm parameters adjusted within the ITU Gaze Tracker user interface to specify the pupil size and detection sensitivity (Mantiuk et al. 2012). Points in the contour between the pupil and the iris are then extracted and fitted to an ellipse using the RANSAC regression method to eliminate possible outliers (San Agustin et al. 2010). The pupil centre is then identified using these points. Corneal reflections are detected using a different threshold, with the assumption that corneal reflections produced by the infrared light source are the ones closest to the pupil, thus eliminating any potential undesirable corneal reflections caused by room lighting (San Agustin 2009).

Gaze estimation in the ITU Gaze Tracker is implemented using the interpolation technique proposed by Morimoto et al. (1999) as described previously in this section. Users are required to perform the calibration procedure using either 9, 12 or 16 points presented in either a set or random order, with the level of accuracy in the calibration dependent on these settings (San Agustin et al. 2010).

The ITU Gaze Tracker is able to use the user's gaze as an input to control the cursor on the screen. The software is able to detect the type of eye movement being performed,

whether it be a fixation or a saccade, and input that information into a smoothing algorithm to produce a smooth cursor on screen (San Agustin 2009). Alternatively, it can stream gaze coordinates via UDP using the built-in network/client API, which can be received by an external application.

2.3 Data Analysis and Visualisation

Data analysis and visualisation is an important feature of eye tracking systems utilised in usability testing applications. This section discusses methods and algorithms for identifying fixations within raw eye tracking data obtained from an eye tracker. From there, it details techniques of both the analysis and visualisation of fixations identified from eye tracking data, outlining various eye tracking metrics and visualisation techniques.

2.3.1 Fixation Identification

Section 1.1 briefly explained that whilst reading text, eye movements are defined as either fixations and saccades. Kumar (2007) states that, "Data from an eye tracker is noisy and includes jitter due to errors in tracking and because of the physiology of the eye." Therefore, in order to statistically analyse and visualise eye tracking data, the raw eye tracking data must be filtered to identify fixations and separate them from saccades.

Jacob & Karn (2003) define a fixation as "a relatively stable eye-in-head position within some threshold of dispersion (typically approximately 2 degrees) over some minimum duration (typically 100 to 200 milliseconds), and with a velocity below some threshold (typically 15 to 100 degrees per second)." A saccade is defined as the fast eye movement between two fixation points (Kumar, Klingner, Puranik, Winograd & Paepcke 2008).

Fixation identification algorithms can be distinguished as either velocity-based, dispersion-based or area-based. Velocity-based algorithms emphasise the velocity information in the data, identifying fixation points by the low velocity between them. Dispersion-based algorithms emphasise the dispersion of fixation points in the data, specifying a fixation radius and assuming that fixation points occur within that radius. Finally,

area-based algorithms identify fixation points within specified areas of interest (Salvucci & Goldberg 2000).

Velocity-Based Algorithms

I-VT (Velocity-Threshold Identification) identifies fixation and saccade points based on their point-to-point velocities. Fixation points typically have a velocity of less than 100 degrees per second, while saccade points typically have a velocity of greater than 300 degrees per second. The I-VT algorithm requires one parameter; the velocity threshold which is dependent on the distance from the eye to the visual stimuli. The pseudocode for the I-VT algorithm, as outlined by Salvucci & Goldberg (2000), is shown in Figure 2.7:

```
I-VT (protocol, velocity threshold)

Calculate point-to-point velocities for each
point in the protocol

Label each point below velocity threshold as
a fixation point, otherwise as a saccade
point

Collapse consecutive fixation points into
fixation groups, removing saccade points

Map each fixation group to a fixation at the
centroid of its points

Return fixations
```

Figure 2.7: I-VT Algorithm (Salvucci & Goldberg 2000).

I-HMM (Hidden Markov Model Identification) is a complex algorithm which uses Hidden Markov models to determine the most likely identification for a set of points, whether it be a fixation or a saccade. "I-HMM uses a two-state HMM in which the states represent the velocity distributions for saccade and fixation points. This probabilistic representation helps I-HMM perform more robust identification than a fixed-threshold method such as I-VT" (Salvucci & Goldberg 2000). The I-HMM algorithm requires eight parameters; two observation probability parameters and two transition probability parameters for both states. The pseudocode for the I-HMM algorithm, as outlined by Salvucci & Goldberg (2000), is shown in Figure 2.8:

```
I-HMM (protocol, HMM)

Calculate point-to-point velocities for each
point in the protocol

Decode velocities with two-state HMM to
identify points as fixation or saccade points

Collapse consecutive fixation points into
fixation groups, removing saccade points

Map each fixation group to a fixation at the
centroid of its points

Return fixations
```

Figure 2.8: I-HMM Algorithm (Salvucci & Goldberg 2000).

Dispersion-Based Algorithms

I-DT (Dispersion-Threshold Identification) identifies fixation points based on the assumption that, due to their low velocity, fixation points lie in close proximity to one another. Fixations are identified as groups of consecutive points within a particular dispersion threshold or radius. Salvucci & Goldberg (2000) utilised a moving window technique which calculates the dispersion between the minimum and maximum coordinates of the points within the window (Equation 2.1) and expands the window if the dispersion is below the specified threshold. The I-DT algorithm requires two parameters; the dispersion threshold and the duration threshold (typically 100 milliseconds). The pseudocode for the I-DT algorithm, as outlined by Salvucci & Goldberg (2000), is shown in Figure 2.9.

$$D = (x_{max} - x_{min}) + (y_{max} - y_{min}) \tag{2.1}$$

I-MST (Minimum Spanning Tree Identification) connects a set of points such that the total length of the lines between the points is minimised. The algorithm requires construction of the minimum spanning tree using Prim's algorithm, followed by a search of the tree to identify fixations and saccades within the points. The I-MST algorithm requires two parameters; the branching depth threshold and the maximum ratio of the mean (μ) and standard deviation (σ) of the edge lengths within the fixation points. The pseudocode for the I-MST algorithm, as outlined by Salvucci & Goldberg (2000), is shown in Figure 2.10.

```
I-DT (protocol, dispersion threshold,
duration threshold)

While there are still points

    Initialize window over first points to
    cover the duration threshold

    If dispersion of window points <=
    threshold

        Add additional points to the window
        until dispersion > threshold

        Note a fixation at the centroid of the
        window points

        Remove window points from points

    Else

        Remove first point from points

Return fixations
```

Figure 2.9: I-DT Algorithm (Salvucci & Goldberg 2000).

```
I-MST (protocol, edge ratio, edge sd)

Construct MST from protocol data points using
Prim's algorithm

Find the maximum branching depth for each MST
point using a depth-first search

Identify saccades as edges whose distances
exceed predefined criteria

Define the parametric properties (μ, σ) of
local edges, identifying saccades when an
edge length exceeds a defined ratio

Identify fixations as clusters of points not
separated by saccades

Return fixations
```

Figure 2.10: I-MST Algorithm (Salvucci & Goldberg 2000).

Area-Based Algorithms

I-AOI (Area of Interest Identification) identifies only fixations that occur within specified areas of interest. It utilises a duration threshold to separate fixations within the area of interest from saccades passing over the area of interest. The I-AOI algorithm requires one parameter; the duration threshold (typically 100 milliseconds), however, it also requires identification of specific areas of interest on the stimuli. The pseudocode for the I-AOI algorithm, as outlined by Salvucci & Goldberg (2000), is shown in Figure 2.11.

```
I-AOI (protocol, duration threshold,
target areas)

Label each point as a fixation point for the
target area in which it lies, or as a saccade
point if none

Collapse consecutive fixation points for the
same target into fixation groups, removing
saccade points

Remove fixation groups that do not span the
minimum duration threshold

Map each fixation group to a fixation at the
centroid of its points

Return fixations
```

Figure 2.11: I-AOI Algorithm (Salvucci & Goldberg 2000).

2.3.2 Eye Tracking Metrics

Eye tracking metrics are used for the analysis and interpretation of eye tracking data. Jacob & Karn (2003) state that, "The usability researcher must choose eye tracking metrics that are relevant to the tasks and their inherent cognitive activities for each usability study individually".

After analysing 20 different usability studies that have incorporated eye tracking, Jacob & Karn (2003) determined the six most frequently used metrics are as follows:

Overall number of fixations is an indication of the user's search efficiency. A larger number of fixations indicates a less efficient search as the user shifts their gaze trying to locate areas of interest. This is possibly an indication of poor arrangement of display elements

Percentage of time spent on each area of interest is a reflection of the importance of the respective area of interest. A higher percentage of time indicates that the area of interest is of high importance.

Overall mean fixation duration is an indication of difficulty in the extraction of information from the screen. A longer duration is the result of the user experiencing difficulty extracting information.

Number of fixations on each area of interest is closely related to the percentage of time spent on each area of interest metric and is also a reflection of the impor-

tance of that area of interest.

Mean fixation duration on each area of interest is an indication of difficulty in the extraction of information from the respective area of interest. Much like the overall mean fixation duration metric, a longer duration is the result of the user experiencing difficulty extracting information.

Overall fixation rate is the number of fixations divided by the time spent fixating, which is closely related to the overall mean fixation duration metric, thus it is an indication of difficulty in the extraction of information from the screen.

Poole & Ball (2005) and Ehmke & Wilson (2007) also compiled their own extensive lists of eye movement metrics for use with eye tracking data analysis. Their sources included previous usability studies and supporting literature.

2.3.3 Visualisation

Eye tracking data can be represented using visualisations, which are graphical representations of the eye tracking metrics. These visualisations are generally superimposed over the original stimulus in order to assist the analyst with interpretation of the visualisation and formulate conclusions about the stimulus. (Blignaut 2010). Two common visualisation techniques utilised by commercial eye tracking systems include gaze plots and heat maps (Tobii Technology 2010).

Gaze Plots

A gaze plot, also known as a saccade plot or scan path, is a visualisation of the sequential organisation of the fixations detected by the eye tracking system (Cooke 2005). On a gaze plot, fixations are displayed as ellipses, and are connected by thin lines representing the saccade between them. Ellipses are numbered in sequential order and have a varying radius, the size of which is directly proportional to the duration of the respective fixation.

Gaze plots can be used to analyse how users interact with a user interface, displaying the frequency of transitions between areas of interest and thus providing an indication

of the efficiency of the program's user interface (Jacob & Karn 2003). An example of a gaze plot superimposed on top of an advertisement is shown in Figure 2.12.

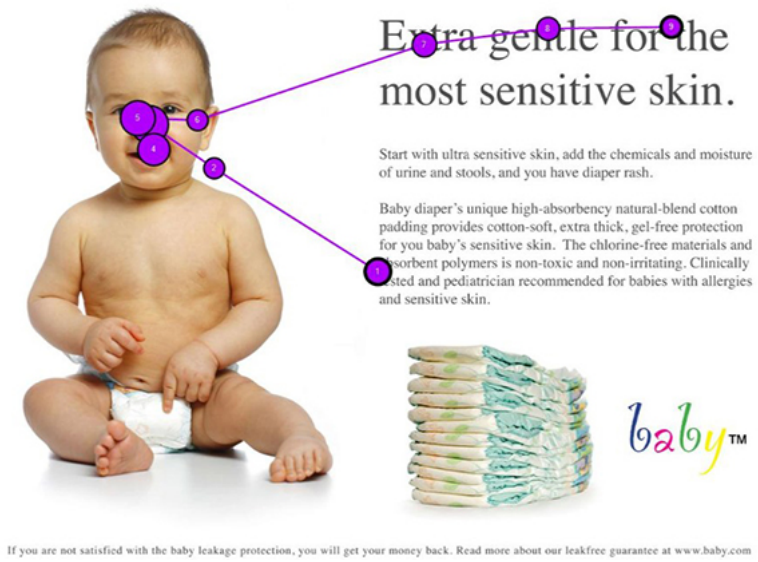


Figure 2.12: A gaze plot produced by an eye tracking system.

Heat Maps

Blignaut (2010) states that, "Heat maps are semi-transparent, multi-coloured layers that cover areas of higher attention with warmer colours and areas of less attention with cooler colours". Colour transitions are generally smoothed with a gradient for increased readability and visual appeal. Some heat maps may only use a single colour, with the level of attention on the respective area of interest indicated by the intensity or opacity of that colour.

Gradient heat maps are are effectively a graphical representation of the number of fixations on each area of interest metric, indicating the importance of the respective area of interest. An example of an eye tracking heat map superimposed on top of an advertisement is shown in Figure 2.13.

Heat maps can also be displayed using a grid, with each cell in the grid coloured to represent the level of attention on the area of interest covered by that cell. Separating each area of interest into discrete cells allows grid heat maps to be more versatile than gradient heat maps, providing the ability visualise a number of different metrics such as the mean fixation duration on each area of interest.

Figure 2.13: A heat map produced by an eye tracking system.

2.4 Summary

This literature review detailed the various aspects of eye tracking, identifying methods of eye tracking, as well as low-cost hardware configurations and open source software solutions. It outlined algorithms used to identify fixations and saccades within raw eye tracking data, as well as the techniques involved in data analysis and visualisation including eye tracking metrics and graphical representations of eye tracking data.

The information presented in this literature review provided the required understanding and knowledge base to formulate the methodology of this research project.

Chapter 3

Methodology

3.1 Outline

This chapter proposes a methodology for this research project, identifying and analysing both the hardware and software requirements of the low-cost eye tracking system while listing the resources necessary for implementation. With the methodology defined, it was subsequently important to assess the consequential effects associated with this research project and undertake a risk assessment to evaluate and mitigate the risks involved with the project work.

3.2 Requirements Analysis

Prior to designing and developing the low-cost eye tracking system, it was important to perform a detailed analysis of the requirements of the system. The requirements analysis sets the basis for the system design and its implementation, specifying the system's required functionality and feature set. This section outlines both the hardware and software requirements of the system, enabling the resource requirements to be analysed and listed for each.

3.2.1 Hardware Requirements

The first requirement of the low-cost eye tracking hardware is that it must be *low-cost*, as specified by the aim of this research project. Section 2.2.2 explained the implementation of both remote and head-mounted eye tracking hardware configurations with consideration of low-cost off-the-shelf hardware. It outlined that video cameras with optical zoom capabilities were required for remote configurations, whereas webcams were appropriate for use in head-mounted configurations. Generally, webcams are significantly lower in cost compared to video cameras. With this in mind, it can be decided that a head-mounted hardware configuration is the most appropriate solution for this research project.

In order to minimise intrusiveness and maximise comfort of the head-mounted hardware for the user, the camera module must be as small and lightweight as possible. To achieve this, only the necessary components of the webcam—the webcam's PCB containing the sensor and the lens, and the USB cable to interface the camera to a computer—should be implemented in the design. Therefore, the webcam must be completely disassembled, allowing these necessary components to be removed from the webcam's casing and thus enabling them to be mounted to a piece of head-mounted hardware.

Section 2.2.2 described the requirement for infrared illumination in the eye tracking hardware to create the dark pupil effect, increasing pupil definition with respect to the iris. This illumination can be provided by infrared LEDs positioned near the webcam. Subsequently, the webcam must be sensitive to infrared light, and must be modified accordingly with the removal of the infrared blocking filter and, ideally, replacing it with an infrared pass filter/visible light blocking filter to reduce the effects of ambient light on the camera hardware.

It has been reported that both the magnetic layer of a floppy disc and overdeveloped photographic film have similar spectral properties to an infrared pass filter, and thus either could be suitably utilised as a low-cost replacement (FreeTrack Forum User Gian92 2012). According to the spectral analysis performed, the magnetic layer of a floppy disc has a filtering threshold between 550 and 600 nanometres, while photographic film has a filtering threshold between 700 and 750 nanometres. Two layers of photographic film were also reported to produce more consistent filtering, with a filtering threshold of 750 nanometres. Using this information, it was decided that two

layers of overdeveloped photographic film should be used as a low-cost alternative to an infrared pass filter in the webcam for best eye tracking performance.

The head-mounted eye tracking hardware was constructed using pre-existing low-cost designs as a basis. Mantiuk et al. (2012) designed and constructed a head-mounted hardware configuration by mounting a modified Microsoft LifeCam VX-1000 webcam and a small illuminator circuit consisting of three infrared LEDs to the frames of a pair of safety glasses, as shown in Figure 2.5 in Section 2.2.2. This procedure is detailed by Kowalik (2010).

Resource Analysis

Using the low-cost head-mounted design by Mantiuk et al. (2012) as detailed by Kowalik (2010) as a basis, the required hardware resources and their respective costs and sources were identified as listed in Table 3.1. Note that all prices are expressed in Australian dollars.

Item	Qty.	$/Unit	Total $	Source
Electronic Components:				
Microsoft LifeCam VX-1000 webcam	1	20.00	20.00	Online auction
ZD1946 3mm IR LED	3	1.25	3.75	Jaycar
220Ω 1W carbon film resistor (2 pack)	1	0.44	0.44	Jaycar
Photographic film	1	N/A	N/A	On-hand item
Mounting Hardware:				
Protector safety glasses	1	13.70	13.70	Bunnings
10 gauge alumnium wire (1.0m)	1	1.83	1.83	Bunnings
Consumables:				
10mm heat-shrink tubing (1.2m)	1	2.95	2.95	Jaycar
100x25mm cable ties (25 pack)	1	0.99	0.99	Bunnings
Insulation tape (1 roll)	1	2.20	2.20	Bunnings
General purpose adhesive	1	3.50	3.50	Bunnings

Table 3.1: Hardware resource requirements for the head-mounted system design based on Mantiuk et al. (2012).

The specifications for the Microsoft LifeCam VX-1000 webcam and the ZD1946 infrared

LED are outlined on their respective datasheets in Appendix C.

The price of the hardware listed in Table 3.1 totalled just 49.36 Australian dollars, which includes the price of consumables—undisputedly placing it in the 'low-cost' price bracket as specified by the aim of this research project.

In addition to the hardware requirements of the low-cost eye tracker, construction of the head-mounted eye tracking hardware required the use of a large variety of tools:

Philips head screwdrivers (multiple sizes) — required for the disassembly of the Microsoft LifeCam VX-1000 webcam to remove the PCB containing the webcam's sensor and lens from its casing, as well as the securing of the modified webcam to the aluminium wire.

Plastic separation tool — facilitated the separation of the front face of the webcam from the rear casing.

Utility knife — required to remove the plastic collar securing the USB cable to the rear casing of the webcam. It was then utilised to dislodge the webcam's infrared sensor from its position behind the lens and subsequently cut the photographic film to its required size in order to secure it behind the lens to act as an infrared pass filter. It was also used for general purposes during the assembly procedure, such as the cutting of heat-shrink tubing and insulation tape.

Wire cutters — utilised to trim terminals on the electronic components and trim excess aluminium after assembly

Soldering iron, solder and solder fluid — enabled the soldering of the infrared illumination circuit to the webcam's 5 volt USB power supply.

Digital multimeter — utilised to test the voltage and current at various points throughout the infrared illumination circuit, ensuring correct operation.

Ruler with 1 millimetre precision — ensured correct and precise measurements were taken during construction and assembly.

Mallet — required to flatten one end of the aluminium wire, producing a surface to mount the webcam

Drill fitted with a 1 millimetre drill piece — required to produce 1 millimetre diameter drill holes on the flattened surface of aluminium wire to mount the webcam.

Centre punch and hammer — used to ensure the drill holes correctly aligned with the screw holes on the webcam (approximately 26 millimetres between hole centres).

Heat gun — utilised to apply sufficient heat to shrink the heat-shrink tubing.

3.2.2 Software Requirements

The ITU Gaze Tracker, presented in Section 2.2.3, provided an ideal open-source software solution for implementation with the low-cost eye tracking system due to its compatibility with low-cost off-the-shelf hardware. The software's features outlined in Section 2.2.3 make it an integral component of the eye tracking system.

In addition to Gaze Tracker, analysis software was designed and developed for implementation in the eye tracking system. The software was required to perform the following tasks:

- Define specific session details such as trial name and test subject name
- Initiate a calibration procedure in Gaze Tracker
- Once calibrated, launch an eye tracking session via a key combination press
- Receive gaze coordinates streamed by Gaze Tracker
- Cease an eye tracking session via a key combination press
- Store gaze coordinates for the respective session to a file
- Implement a fixation identification algorithm to identify fixations within the gaze coordinates
- Calculate relevant eye tracking metrics for the identified fixations and display a statistical report of the analysis
- Generate visualisations of the identified fixations, including a gaze plot and heat maps

It was decided that I-DT fixation identification described in Section 2.3.1 would be implemented in the software, as its implementation is relatively simple in comparison to other fixation identification algorithms, and it only requires one user-defined parameter; the dispersion threshold, or radius, in pixels. This parameter should be estimated based on the size and resolution of the screen in which the eye tracking session is being performed on, as well as the test subject's view distance to the screen. For example, if the test subject is gazing at a large screen from a small distance, the screen will occupy a greater percentage of their field of view, thus a smaller dispersion threshold should be selected. Conversely, if the test subject is gazing at a small screen from a large distance, the screen will occupy a smaller percentage of their field of view, thus a larger dispersion threshold should be selected (Blignaut 2009).

The eye tracking metrics to be calculated by the software were chosen from the comprehensive list collaborated by Jacob & Karn (2003). These metrics include:

- Overall number of fixations (defined as either on-screen or off-screen fixations)
- Overall mean fixation duration
- Overall fixation rate
- Mean saccade length
- Number of fixations on each area of interest
- Total fixation duration on each area of interest
- Mean fixation duration on each area of interest

Finally, with these metrics in mind, it was decided that the software should generate heat maps utilising the grid technique outlined in Section 2.3.3, where each cell in the grid arbitrarily defines areas of interest on the screen. This allows the three area of interest-dependent metrics chosen to be easily visualised within the grid.

Resource Analysis

The following software resources were identified as requirements for the development of the analysis software:

MinGW GCC/C++ compiler — required for compilation of source code files

Code::Blocks IDE — provides an environment to facilitate the development of the software, with features such as syntax highlighting, debugging and tools for developing and building entire projects.

Windows API — defined in `windows.h`, it enables the utilisation of Windows API functions. In addition to this, it provides GDI graphics functions with the use of the `gdi32` library, required by the software for drawing gaze plots and heat maps.

Windows Sockets 2 API — defined `winsock2.h`, it provides the API functions required to implement networking functions in the software, particularly the TCP/IP and UDP protocols required by the software to send a calibration command and receive gaze coordinates streamed by Gaze Tracker, respectively. The `wsock32` library is also required.

3.3 Assessment of Consequential Effects

The potential consequential effects that result from this research project involve both safety issues and ethical considerations. These effects are can have negative impacts for the user, thus the research project was undertaken with the user's best interests in mind. This section identifies and describes those safety issues, evaluating their severity with respect to the user. Furthermore, the ethical aspects of the research project are outlined and considered.

3.3.1 Safety Issues

A potential safety issue associated with this research project both prior to and after its completion involves the use of infrared light to illuminate the user's eye. As described in Section 2.2.1, in order to produce the dark pupil effect to improve pupil identification, the user's eye must be adequately illuminated by a single or multiple infrared light sources directed at the eye. The relatively long wavelength of infrared light (approx 780-1400nm as defined by the IR-A spectral region (Mulvey, Villanueva, Sliney, Lange, Cotmore & Donegan 2008)) causes it to fall outside of the visible light spectrum, so the hazard is not apparent to the user.

Mulvey et al. (2008) state that "The ACGIH and ICNIRP recommend a maximal daily corneal exposure of 10 milliwatts per square centimetre total irradiance for wavelengths 770 to 3,000 nanometres for day-long, continuous exposures". Other low-cost eye trackers using infrared illumination have reported irradiance levels well below 10 milliwatts per square centimetre (Babcock & Pelz 2004), however, this recommendation was still considered when implementing infrared illumination circuit on the head-mounted eye tracking hardware.

3.3.2 Ethical Considerations

Engineers Australia (2010) specify their ethical standards and requirements in the Code of Ethics. It was of utmost importance that the Code of Ethics be strictly upheld at all times throughout the duration of this research project.

Secondly, in order to track the user's gaze, a portion their face, and more importantly; the direction of their gaze, must be recorded by the camera for the duration of the eye tracking session. This could be considered as a violation of the user's privacy. Furthermore, the eye tracking data and output generated from the session will likely be stored for further analysis or comparison with other users. Therefore, it is important that the user fully understands and acknowledges these requirements of the eye tracking system and thus agrees to partake in the session.

3.4 Risk Assessment

A risk assessment of this research project was required to be performed to identify all potential hazards and the risks they may pose. Risks can be associated with either undertaking the project work or the completed project itself. Identification of the hazards involved with this project, their subsequent potential risks and steps to mitigate those risks are detailed in Appendix B.1. Table B.1 presents a risk assessment of this research project, grading each hazard on its exposure, likelihood and severity while using those values to determine its overall risk level.

3.5 Project Timeline

Figure B.1 in Appendix B.2 presents a graphical timeline of the tasks undertaken throughout the course of this research project and their approximate commencement and completion dates. Key dates are displayed as thick vertical lines. Table B.2 lists these key dates with their corresponding number from the timeline.

3.6 Summary

This chapter outlined a methodology for the low-cost eye tracking system. The requirements of both the hardware and software in the system were identified and analysed, providing the basis for the design and implementation of the system. Furthermore, an assessment of the consequential effects of the system was performed, evaluating the potential impacts of the system in terms of both ethics and safety. Finally, a risk assessment was undertaken and approaches of mitigation of those risks were suggested.

Chapter 4

Design and Implementation

4.1 Outline

The design elements of this research project involve both hardware design and software design. The implementation of these elements form the low-cost eye tracking system. As stated in Section 3.2.2, the ITU Gaze Tracker software is an integral component of the eye tracking system, thus both the hardware and software must be designed to operate in conjunction with Gaze Tracker. This chapter details both the hardware and software designs and their implementation with Gaze Tracker to form the eye tracking system.

4.2 Hardware Design

As specified in Section 3.2.1, a head-mounted configuration with a modified Microsoft LifeCam VX-1000 webcam (Figure 4.1) (Appendix C) forms the basis of the low-cost eye tracking hardware. This section discusses the hardware design, describing how the camera was modified for use in a head-mounted configuration and, subsequently, the assembly procedure of the head-mounted hardware.

Figure 4.1: Microsoft LifeCam VX-1000 webcam (Microsoft Corporation 2011).

4.2.1 Webcam Modification

The following procedure outlines the steps that were performed to modify the webcam for use in a head-mounted configuration.

In order for the camera to be as small and lightweight as possible for use in a head-mounted configuration, the PCB containing the webcam's sensor and lens was required to be removed from the its casing:

1. The screws identified at Ⓐ and Ⓑ in Figure 4.2 were unscrewed and the base of the webcam was removed.

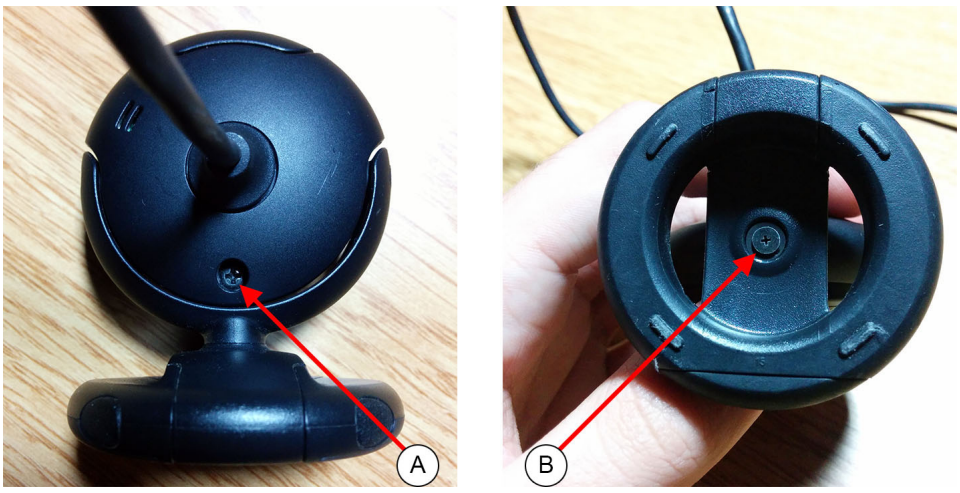


Figure 4.2: Webcam screw locations.

2. The front face of the webcam was separated from the rear casing as shown in

Figure 4.3.

Figure 4.3: Removing the front face.

3. After unplugging the USB cable at Ⓐ in Figure 4.4, the screws identified at Ⓑ and Ⓒ were unscrewed to remove the PCB containing the camera from the rear casing of the webcam. The microphone cable at Ⓓ was also removed.

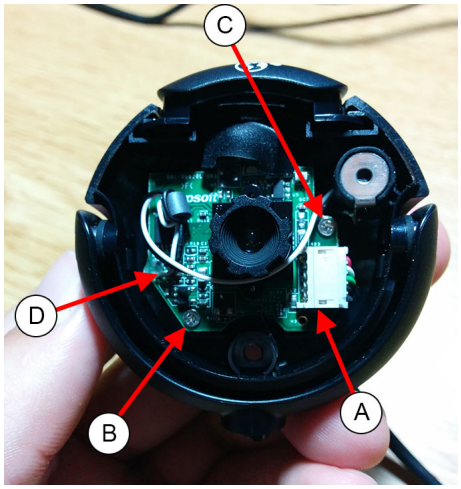


Figure 4.4: Removing the PCB.

4. The USB cable was removed from the rear casing of the webcam by removing the plastic collar securing it to the case.

In order for the camera to be sensitive to the infrared light spectrum, the infrared blocking filter required removal from the camera's lens and replaced with two layers of overdeveloped photographic film as explained in Section 3.2.1:

5. The lens was unscrewed from the lens holder on the webcam. The small infrared

blocking filter was located behind the lens as shown at Ⓐ in Figure 4.5 and removed using a thin, sharp blade. The broken infrared blocking filter is shown at Ⓑ.

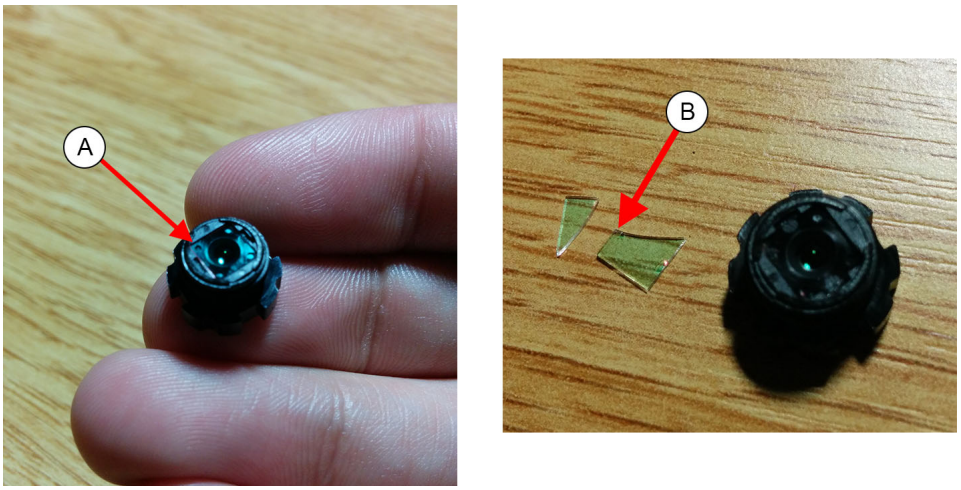


Figure 4.5: Removing the infrared blocking filter from the lens.

6. Two small squares of the same size as the infrared blocking filter (approximately 5 square millimetres) were cut from a piece of overdeveloped photographic film and secured in place behind the lens with a small amount of glue as shown in Figure 4.6.

Figure 4.6: Photographic film behind the lens.

7. The lens was reinserted into the lens holder on the webcam.

As explained in Section 3.2.1, the head-mounted configuration requires infrared illumination to create the dark pupil effect for eye tracking:

8. Three 3 millimetre ZD1946 infrared LEDs (Appendix C) surrounding the webcam's lens were soldered in series with a 22 ohm resistor and the 5 volt USB power supply on the webcam's USB connector, as shown by the schematic in Figure 4.7. Assuming typical conditions, the ZD1946 infared LED has a forward voltage range of 1.2 to 1.4 volts, allowing three in series to be powered by the 5 volt USB power supply, providing sufficient illumination with a minimalistic implementation. A 22 ohm resistor was chosen to provide a supply current within the ZD1946 infrared LED's forward current range of 20 to 100 milliamps (Equations 4.1 and 4.2) for typical conditions.

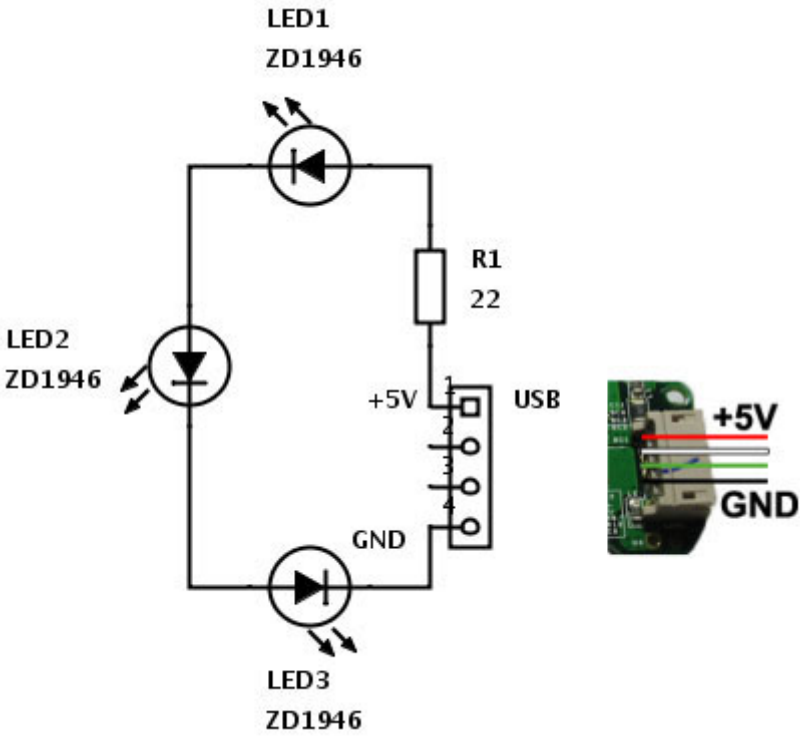


Figure 4.7: Schametic of the infrared illumination circuit.

$$\begin{aligned} R_{1min} &\approx \frac{V_{USB} - 3V_{Fmin}}{I_{Fmax}} \\ &\approx \frac{5 - 3 \times 1.2}{0.10} \\ &\approx 14\Omega \end{aligned} \tag{4.1}$$

$$\begin{aligned} R_{1max} &\approx \frac{V_{USB} - 3V_{Fmax}}{I_{Fmin}} \\ &\approx \frac{5 - 3 \times 1.4}{0.02} \\ &\approx 40\Omega \end{aligned} \tag{4.2}$$

The final capture module which can be seen in Figure 4.10 is a lightweight, infrared sensitive version of the Microsoft LifeCam VX-1000 webcam with an infrared illumination circuit.

4.2.2 Mounting and Assembly

The following procedure outlines the steps that were performed to assemble the head-mounted hardware, including mounting the modified webcam.

Firstly, the modified webcam was mounted to a 300 millimetre length of 10 gauge aluminium wire:

1. A section of approximately 40 millimetres at one end of the aluminium wire was flattened by striking it with a mallet on a hard surface (Figure 4.8).

2. To align with the mounting screw holes on the webcam, two holes of 1 millimetre diameter were drilled at a distance of 26 millimetres between hole centres on the flattened section of aluminium wire (Figure 4.8).

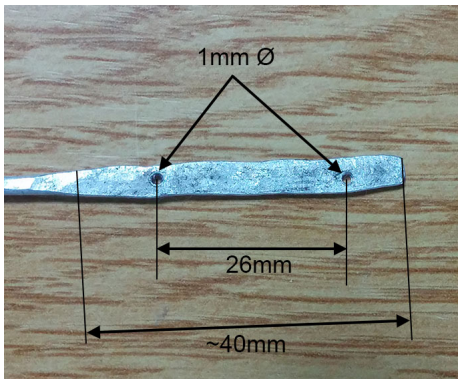


Figure 4.8: Flattened section of aluminium wire with screw holes.

3. The flattened section of the aluminium wire was wrapped in insulation tape and a thin slice of soft foam was adhered to one side. The two holes were re-drilled to create holes in both the tape and foam (Figure 4.9).

4. The remaining length of aluminium wire was covered in heat-shrink tubing of 10 millimetre diameter.

5. The webcam's USB cable was gradually pushed through the heat-shrink tubing, starting by inserting the small webcam connector into the heat-shrink tubing at

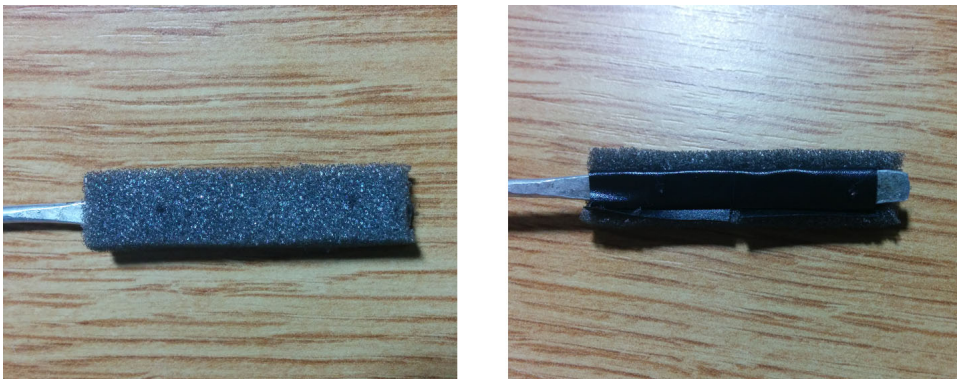


Figure 4.9: Front (left) and back (right) of the flattened section of aluminium wire wrapped in insulation tape with foam attached.

the non-flattened end of the aluminium wire and working it towards the flattened end.

6. The webcam was placed on top of the foam on the flattened end of the aluminium wire with the mounting screw holes aligned with the drilled holes. It was then screwed into place using the original mounting screws (Figure 4.10).

7. The webcam's USB cable was inserted into its connector on the webcam and secured to the aluminium wire at the end of the heat-shrink tubing with a cable tie (Figure 4.10).

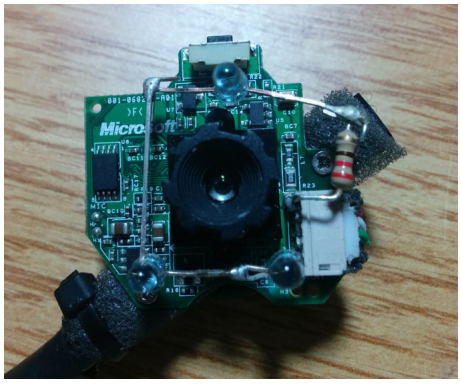


Figure 4.10: The webcam mounted to the alumnium wire.

8. The heat-shrink tubing was then shrunk with a heat gun to tightly wrap the USB cable and aluminium wire.

Secondly, the head-mounted hardware was assembled by attaching the aluminium wire to the frames of a pair of safety glasses:

9. The protective shield of the safety glasses was removed by releasing the clip above the nose pad.

10. The aluminium wire was secured to the left arm of the frames using multiple cable ties with the webcam extending forwards and facing inwards, allowing approximately 200 millimetres distance from the hinge to the tip of the flattened end of the aluminium wire.

11. With the point of inflection near the hinge of the left arm, the aluminium wire was bent downward at an angle of approximately 55 degrees as shown in Figure 4.11. It was also bent slightly outward at an angle of approximately 15 degrees to reduce obstruction to the user's field of view.

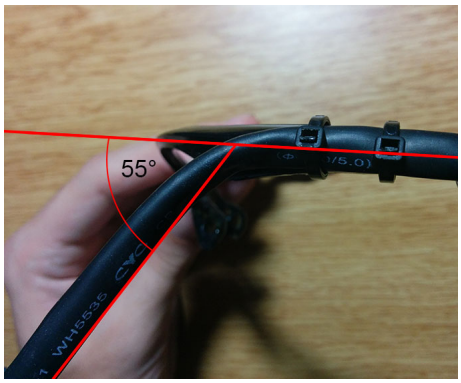


Figure 4.11: Downward bending angle of the aluminium wire at the hinge.

12. With the point of inflection approximately 30 millimetres from the edge of the webcam, the aluminium wire was bent upward and inward at an angle of approximately 105 degrees, creating an inside angle of approximately 75 degrees as shown in Figure 4.12. This caused the webcam to point towards the left eye region of the frames at a distance of approximately 80 millimetres.

13. Finally, the excess aluminium wire was removed from behind the left arm of the frames and any exposed aluminium wire was covered with heat-shrink tubing.

The final head-mounted hardware design is shown from multiple angles in Figure 4.13, while Figure 4.14 displays the head-mounted hardware being worn.

Figure 4.15 displays an image captured by the webcam demonstrating the dark pupil effect caused by the infrared illumination circuit in the hardware.

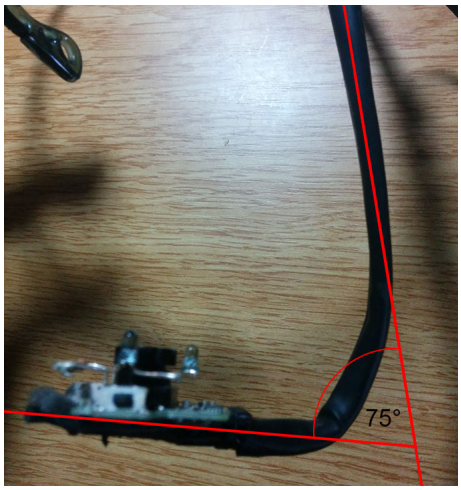


Figure 4.12: Inward bending angle of the aluminium wire at the webcam.

Figure 4.13: Final head-mounted hardware design.

Figure 4.14: Head-mounted hardware being worn.

Figure 4.15: Image captured by the head-mounted hardware.

4.3 Software Design

Eye tracking analysis software informally named Gaze Analyser was designed and developed for use in the eye tracking system. As specified in Section 3.2.2, the software was developed using the C++ programming language and was designed to facilitate an eye tracking session for the user; setting session details, receiving and storing gaze coordinate data, identifying fixations within that data and analysing and creating visualisations of those fixations. This section describes the software design and development, detailing class functions and their implementation.

4.3.1 Gaze Analyser Overview

Users interact with Gaze Analyser via a text-based user interface. Upon launching the program, the user is presented with a text-driven menu, prompting them to either start a new eye tracking session or exit the program. This menu is implemented in the `main()` function (Appendix D.1) utilising switch-case statements to handle and validate user input. Figure 4.16 displays Gaze Analyser's main window presenting the text-driven menu.

By considering the software requirements specified in Section 3.2.2, four unique classes were identified for Gaze Analyser:

- Session class (Section 4.3.2)

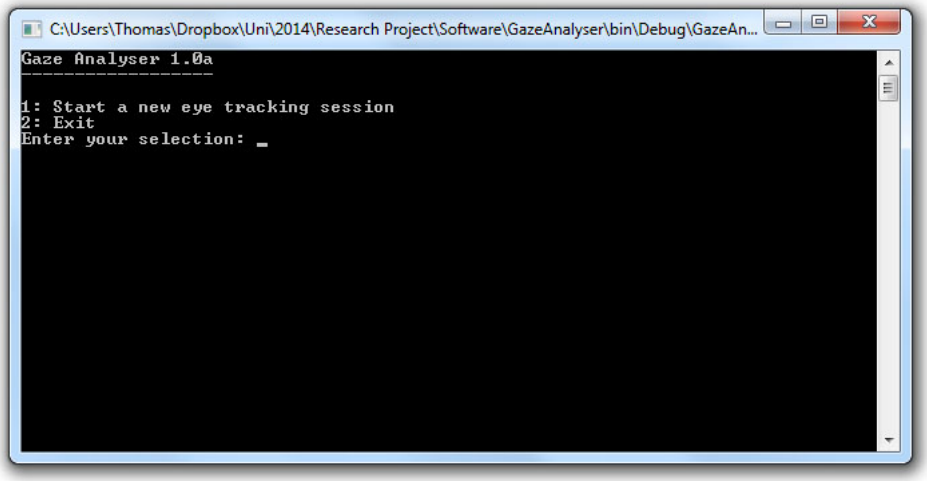


Figure 4.16: Gaze Analyser's main window.

- Metrics class (Section 4.3.3)
- GazePlot class (Section 4.3.4)
- HeatMap class (Section 4.3.5)

Four global variables were also required by the program (Section D.3):

- `res_width` and `res_height`: the screen's horizontal and vertical resolution values in pixels respectively, initialised when the program is launched
- `t_fix_min`: the constant specifying the minimum fixation duration threshold in milliseconds for the I-DT fixation identification algorithm implementation, set to 100 milliseconds; and
- `track`: the Boolean to control gaze coordinate data retrieval from Gaze Tracker in the `udp_receive()` function (Section D.5)

4.3.2 Session Class

Listing 4.1: Session Class Definition

```
/**
 * @file Session.hpp
 * @author Thomas Bradford (thomas.bradford91@gmail.com)
 *
 * Session class definition
 */
```

```
#ifndef SESSION_HPP
#define SESSION_HPP

#include <string>
#include <vector>
#include <windows.h>

#include "Point.hpp"

class Session
{
    void get_details();
    void calibrate();
    void tracking();
    unsigned int process();
    void get_limits(std::vector<Point>);
    void add_fixation(std::vector<Point>);
    Point min;
    Point max;
public:
    Session();
    ~Session();
    std::string trial;
    std::string subject;
    int fix_radius;
    HDC hCaptureDC;
    std::vector<Point> fix_path;
    int t_session;
};

#endif // SESSION_HPP
```

Refer to Appendix D.8 for the Session class implementation.

The Session class performs three main operations: initialisation of a new eye tracking session, calibration, initiation and cessation of that eye tracking session; and the processing of the gaze coordinate data obtained in that session to identify fixations. All operations are handled by the default constructor for the class.

A new Session object is declared whenever the user selects the 'Start a new eye tracking session' menu option in Gaze Analyser's main window. Upon declaration of a new Session object, the default constructor calls the `get_details()` function to obtain the user-specific details for that session. This function prompts the user to enter the names of the eye tracking trial and the test subject, as well as define the desired fixation radius in pixels for that session. The input from each of these prompts are stored in public

class variables `trial`, `subject` and `fix_radius` respectively. The function ensures valid input for all three variables: checking if the trial name is not blank, the subject name is not blank and does not already have a results file for that trial, and if the fixation radius is greater than 0 pixels.

After obtaining the user-specific details for the Session, the default constructor calls the `calibrate()` function to calibrate the eye tracker. The function calls an external function, `tcp_send()` (Appendix D.4), to send the 'CAL_START' command via TCP port 5555 on localhost; the command server for Gaze Tracker running concurrently on the machine. It ensures the command is sent successfully, creating a calibration window in Gaze Tracker and setting it as the foreground window. After calibration in Gaze Tracker is complete, the function prompts the user to either accept the calibration or perform another calibration procedure to potentially achieve a more accurate calibration.

Once the calibration is accepted by the user, the default constructor calls the `tracking()` function which handles the eye tracking procedure itself. The function informs the user that the eye tracker is ready and instructs them to press the 'Ctrl+Alt+E' key combination to begin an eye tracking session. When the key combination press is detected, the eye tracking data file '`subject`_data.txt' is created in the `trial` directory and the header containing the `trial`, `subject` and the time of commencement of the eye tracking session is written to the file. The function then sets the `track` Boolean to true and creates a new thread, calling an external function, `udp_receive()` (Appendix D.5), which ensures a connection to UDP port 6666; the data server for Gaze Tracker, receives gaze coordinate data from the data server, writes it to a buffer and sends the buffer to the file stream for the data file. The `udp_receive()` function receives gaze coordinate data until the `track` Boolean is set to false in its calling thread.

After creating a new thread for the `udp_receive()` function, the `tracking()` function calls an external function, `screenshot()` (Appendix D.6), which returns a handle to the device context of the current desktop screen. This handle is stored in public class variable `hCaptureDC`. The user is then instructed to press the 'Ctrl+Alt+E' key combination to end the eye tracking session. When the key combination press is detected, the `track` Boolean is set to false, thus ending the previously created thread and waiting for it to finish. Finally, the footer containing the time of completion of the eye tracking session is written to the data file.

The default constructor then calls the `process()` function to process the gaze coordinate data stored to the data file and identify fixations within that data, storing them to the public class vector `fix_path`. After opening the file, the function loops through the file line by line until the end-of-file is reached. Lines containing gaze coordinate data contain the string 'STREAM_DATA' and are formatted as follows:

```
STREAM_DATA <timestamp> <x-coordinate> <y-coordinate>
```

Where the timestamp is expressed in milliseconds. If the 'STREAM_DATA' string is found in the current line, the timestamp, x-coordinate and y-coordinate of the respective point are stored in a Point struct (Listing 4.2).

Listing 4.2: Point Struct Definition

```
/**
 * @file Point.hpp
 * @author Thomas Bradford (thomas.bradford91@gmail.com)
 *
 * Point struct definition
 */

#ifndef POINT_HPP
#define POINT_HPP

struct Point
{
    Point(double = 0, double = 0, long long = 0);
    double x;
    double y;
    long long t;
    bool on_screen;
};

#endif // POINT_HPP
```

The function then implements an I-DT fixation identification algorithm, as described in Section 3.2.2. Points are added to the current set of potential fixation points until the difference in the time stamp between the first and last points exceeds the minimum fixation duration threshold specified by `t_fix_min`, thus initialising the current set of potential fixation points. Points are then continued to be added to the current set of potential fixation points until the resultant dispersion of those points exceeds double the value of the `fix_radius` defined by the user, in which case a fixation is identified. Otherwise, the first point in the current set of potential fixation points is removed from

the set and the current point is added to the set, thus shifting the current potential fixation.

The algorithm was adapted from that proposed by Salvucci & Goldberg (2000) to support the line-by-line processing of gaze coordinate data. The pseudocode for the I-DT algorithm implemented in this function is displayed in Algorithm 4.1.

Algorithm 4.1 I-DT algorithm implementation

```
if t_last − t_first < t_fix_min then
    Add current point to fixation points vector
else if ((x_max − x_min) + (y_max − y_min) ≤ 2*fix_radius then
    Add current point to fixation points vector
    Get new min and max coordinates
    if (x_max − x_min) + (y_max − y_min) > 2*fix_radius then
        Remove current point from fixation points vector
        Calculate fixation coordinates using fixation points vector
        Add fixation to fix_path vector
        Clear fixation points vector
        Add current point back to fixation points vector
    end if
else
    Erase first point from fixation points vector
    Add current point to fixation points vector
end if
Get new min and max coordinates
```

After a fixation is identified, the coordinates at its centroid are calculated by averaging the coordinates of all points in the fixation points vector (Equation 4.3). The duration of the fixation is also calculated by subtracting the time of the first point in the fixation from the time of the last point in the fixation (Equation 4.4).

$$(x, y)_{fixation} = \frac{\sum_{i=1}^{n}(x, y)_i}{n} \tag{4.3}$$

$$T_{fixation} = t_n - t_1 \tag{4.4}$$

Where n is the number of points in the fixation points vector.

The fixation's coordinates and duration are stored in a Point struct. The Point struct also contains an `on_screen` Boolean—which signifies whether or not the point lies within the boundaries of the screen as defined by the `res_width` and `res_height` global

variables—and is set accordingly. The Point data is then pushed back to the `fix_path` vector. This is implemented in the `add_fixation` class function.

Finally, after all gaze coordinate data has been processed, the function calculates the duration of the eye tracking session (`t_session`) by subtracting the time of the first point in the session from the time of the last point in the session and returns the number of fixations identified in the gaze coordinate data.

If fixations were identified in the gaze coordinate data (that is, if the return value of the `process()` function is greater than 0), the default constructor creates new Metrics (Section 4.3.3), GazePlot (Section 4.3.4) and HeatMap (Section 4.3.5) objects to generate results and visualisations for that Session.

4.3.3 Metrics Class

Listing 4.3: Metrics Class Definition

```
/**
 * @file Metrics.hpp
 * @author Thomas Bradford (thomas.bradford91@gmail.com)
 *
 * Metrics class definition
 */

#ifndef METRICS_HPP
#define METRICS_HPP

#include "Session.hpp"

class Metrics
{
    void write_results(std::string, std::string, int);
    unsigned int t_session_mins;
    unsigned int t_session_secs;
    unsigned int on;
    unsigned int off;
    double fix_rate;
    double t_fix_mean;
    double saccade_length_mean;
public:
    Metrics(Session*);
};

#endif // METRICS_HPP
```

Refer to Appendix D.9 for the Metrics class implementation.

The Metrics class is responsible for the calculation of the chosen eye tracking metrics identified in Section 3.2.2, as well as the generation of the results file for the session. The constructor accepts a pointer to a Session class object as an argument. Using the Session object's public class variables, it calculates the number of on-screen and off-screen fixations within the `fix_path` vector. The overall fixation rate for the session in fixations per second is calculated by dividing the number of fixations identified (equivalent to the size of the `fix_path` vector) by the session duration in seconds. The mean on-screen fixation duration in milliseconds is calculated by averaging the duration of each on-screen fixation identified. Finally, the length of each saccade in pixels between two on-screen fixation points is calculated using Pythagoras' theorem and the mean on-screen saccade length is calculated (Equation 4.5).

$$\overline{L_{saccade}} = \frac{\sum_{i=1}^{n-1} \sqrt{|x_i - x_{i+1}|^2 + |y_i - y_{i+1}|^2}}{n} \tag{4.5}$$

Where n is the number of on-screen fixations identified.

After calculating the eye tracking metrics, the constructor calls the `write_results()` function which creates the '`subject`_results.txt' file and writes the session details including the trial name, subject name, date, screen resolution, fixation radius and session duration. It then writes all eye tracking metrics calculated in the constructor. A sample of the results file is displayed in Figure E.1 in Appendix E.

4.3.4 GazePlot Class

Listing 4.4: GazePlot Class Definition

```
/**
 * @file GazePlot.hpp
 * @author Thomas Bradford (thomas.bradford91@gmail.com)
 *
 * GazePlot class definition
 */

#ifndef GAZEPLOT_HPP
#define GAZEPLOT_HPP

#include <vector>
#include <windows.h>

#include "Session.hpp"
```

```
class GazePlot {
    void get_upper(std::vector<Point>);
    void draw(std::vector<Point>, int);
    int get_radius(int, int);
    void draw_footer(std::string, std::string, int);
    unsigned int lower;
    unsigned int upper;
    HDC hGazePlotDC;
    HBITMAP hGazePlotBmp;
    HFONT hFont;
    HPEN hPurplePen;
    HPEN hGreenPen;
    HPEN hBluePen;
    HPEN hRedPen;
    HPEN hOrangePen;
    HPEN hBlackPen;
    HBRUSH hLightPurpleBrush;
    HBRUSH hLightGreenBrush;
    HBRUSH hLightBlueBrush;
    HBRUSH hLightRedBrush;
    HBRUSH hLightOrangeBrush;
    HBRUSH hWhiteBrush;
public:
    GazePlot(Session*);
    ~GazePlot();
};

#endif // GAZEPLOT_HPP
```

Refer to Appendix D.11 for the GazePlot class implementation.

As the name implies, the GazePlot class handles the drawing of the visual representations of fixations and saccades for generation of the gaze plot for the session. It utilises the functions provided by the Windows GDI API to draw such shapes.

The constructor accepts a pointer to a Session class object as an argument. When the constructor is called, the device context captured in the session is copied and its respective bitmap is created for the GazePlot object. The font, pens and brushes used in the class are also initialised.

As stated in Section 2.3.3, the radius of the ellipse representing a fixation is directly proportional to the duration of that fixation. Before achieving this, the duration limits for the session's fixations are obtained. With the `lower` limit set to the value of the `t_fix_min` global variable, the `upper` limit is obtained by finding the maximum duration

of fixations in the session's `fix_path` vector with the `get_upper()` function.

The constructor then calls the `draw()` function to draw the ellipses and connecting lines to the device context. Firstly, the connecting line between each on-screen fixation point is drawn using the `MoveToEx()` and `LineTo()` GDI functions. Secondly, the ellipse for each on-screen fixation is drawn using the `Ellipse()` GDI function.

Each fixation in the `fix_path` vector can be assigned a normalised weight proportional to its duration and relative to the maximum fixation duration using the `lower` and `upper` limits (Equation 4.6).

$$W_{fixation} = \frac{T_{fixation} - \texttt{lower}}{\texttt{upper} - \texttt{lower}} \tag{4.6}$$

This weight is then used to determine the radius of the fixation's respective ellipse on the gaze plot, with the minimum radius equal to the `fix_radius` defined in the session, and the maximum radius equal to twice the `fix_radius` (Equation 4.7). This is implemented in the `get_radius` function.

$$r_{fixation} = W_{fixation} \times \texttt{fix\_radius} + \texttt{fix\_radius} \tag{4.7}$$

With the ellipse radius for the fixation obtained, the fixation point is then identified as one of the following:

- First fixation in the session, represented as a green ellipse (Figure 4.17 Ⓐ)
- Last fixation in the session, represented as a red ellipse (Figure 4.17 Ⓑ)
- First on-screen fixation since leaving the screen's boundaries, represented as a blue ellipse (Figure 4.17 Ⓒ)
- Last on-screen fixation before leaving the screen's boundaries, represented as an orange ellipse (Figure 4.17 Ⓓ)
- Regular fixation, represented as a purple ellipse (Figure 4.17 Ⓔ)

Thirdly, the function sequentially numbers each fixation using the `DrawText()` GDI function.

Figure 4.17: Gaze plot fixation ellipses.

Finally, the constructor calls the `draw_footer()` function to draw the gaze plot's footer to the device context, displaying session details including the trial name, subject name, date and fixation radius, as well as the legend displaying each type of fixation in Figure 4.17. The device context's bitmap holding the final gaze plot is then saved to a file using the external `save_bitmap()` function (Appendix D.10). A sample gaze plot is displayed in Figure E.2 in Appendix E.

4.3.5 HeatMap Class

Listing 4.5: HeatMap Class Definition

```
/**
 * @file HeatMap.hpp
 * @author Thomas Bradford (thomas.bradford91@gmail.com)
 *
 * HeatMap class definition
 */

#ifndef HEATMAP_HPP
#define HEATMAP_HPP

#include <windows.h>

#include "Session.hpp"

class HeatMap
{
    void get_upper();
    void draw();
    void draw_footer(std::string, std::string, int);
    int cell_size;
    unsigned int **grid;
    unsigned int grid_width;
    unsigned int grid_height;
    unsigned int lower;
    unsigned int upper;
    HDC hHeatMapDC;
    HBITMAP hHeatMapBmp;
    HFONT hFont;
    HPEN hBlackPen;
    HBRUSH hWhiteBrush;
    ULONG_PTR gdiplusToken;
```

```
public:
    HeatMap(Session*);
    ~HeatMap();
    void fix_count(Session*);
    void t_fix(Session*);
    void t_fix_mean(Session*);
};

#endif // HEATMAP_HPP
```

Refer to Appendix D.12 for the HeatMap class implementation.

Similar to the GazePlot class, the HeatMap class handles the generation and drawing of various types of heat maps for the session. It utilises the Windows GDI+ API which runs on top of the GDI API and offers additional features required for heat map generation such as brush transparency and gradient fills.

The constructor accepts a pointer to a Session class object as an argument. When the constructor is called, the `cell_size` for the heat map grid in pixels is calculated as double the `fix_radius` for the session, and the `grid_width` and `grid_height` are calculated based on the `res_width` and `res_height` global variables. The multidimensional `grid` array is then created to the size of `grid_width` and `grid_height` and initialised to 0 using the `memset()` function. For example, a screen resolution of 1920 pixels wide by 1080 pixels high with a `fix_radius` of 20 pixels would yield a grid of 96 cells high by 54 cells wide, with each cell being 40 square pixels in size.

After creating and initialising the heat map's grid, the constructor copies the device context captured in the session and creates its respective bitmap for the HeatMap object. The font, pens and brushes used in the class are also initialised. Finally, the GDI+ API is initialised for use in the class member functions.

Once the class is initialised by the constructor, one of three heat maps can be generated. These heat maps visualise the following eye tracking metrics:

1. Number of fixations on each area if interest (using the `fix_count()` function)
2. Overall fixation duration on each area of interest (using the `t_fix()` function)
3. Mean fixation duration on each area of interest (using the `t_fix_mean()` function)

Where the areas of interest are arbitrarily defined as cells within the grid. It is also important to note that the mean fixation duration on each area of interest metric is effectively a factor of metrics 1 and 2.

To obtain the corresponding cell in the grid for a fixation point, the coordinates of each fixation point in the session's `fix_path` vector are divided by the `cell_size` and rounded down to the nearest integer. The corresponding cell is then set according to the metric being visualised, with the cell incremented for metric 1 or an accumulated sum of fixation durations for metric 2. Metric 3 performs both operations with separate grids and divides corresponding cells from each grid to obtain the final grid, thus generating three different grids.

Once the grid is obtained, the grid's `lower` and `upper` limits are set and obtained respectively, with the `lower` limit set to 1 for metric 1, and the value of the `t_fix_min` for metrics 2 and 3. The `get_upper()` function loops through each cell in the grid to obtain the `upper` limit.

The `draw()` function then draws the grid to the device context. Much like the GazePlot class's `draw()` function, each cell in the grid is assigned a normalised weight proportional to its value and relative to the maximum cell value in the grid using the `lower` and `upper` limits. While the implementation of this is identical for all three heat map metrics, Equation 4.8 describes the calculation of the weight of a cell for duration-based metrics 2 and 3.

$$W_{cell} = \frac{T_{cell} - \texttt{lower}}{\texttt{upper} - \texttt{lower}} \tag{4.8}$$

This weight can then be used to determine the corresponding RGB colour value of the respective cell, with the weight directly proportional to the warmth of the colour. The gradient in Figure 4.18 displays the colour transition with increasing weight values.

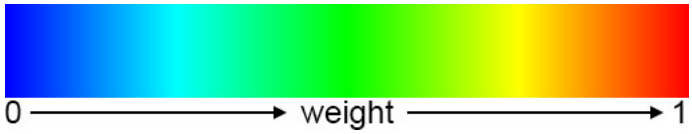


Figure 4.18: Heat map weight gradient.

Finally, with the corresponding RGB colour value of the respective cell obtained, the

function draws the cell at that colour to the device context at 50 percent transparency.

The `draw_footer()` function is then called to draw the heat map's footer to the device context, displaying session details including the trial name, subject name, date and fixation radius, as well as the legend displaying the gradient in Figure 4.18. The device context's bitmap holding the final heat map is then saved to a file using the external `save_bitmap()` function (Appendix D.10). Sample heat maps for each of the three metrics are displayed in Figure E.3, Figure E.4 and Figure E.5 respectively in Appendix E.

4.4 System Implementation

In order to form the eye tracking system, all hardware and software components in the system must be implemented to operate in conjunction with one another. As stated in Section 3.2.2, the ITU Gaze Tracker software is an integral component of the eye tracking system. Extending on this, it must be noted that Gaze Tracker plays central role in the system's implementation. This section describes the implementation process in Gaze Tracker to initialise the eye tracking system.

Figure 4.19 displays a detailed block diagram of the eye tracking system.

4.4.1 Hardware Setup

With the eye tracking hardware connected to the computer on which the eye tracking session will be conducted, the first step in the implementation procedure is to install the Microsoft LifeCam webcam drivers. This process is automated on a machine running the Windows 7 operating system through the use of the Windows Update tool. After the drivers are installed, the Microsoft LifeCam VX-1000 can be selected as an input device in Gaze Tracker via the 'Camera' tab in the 'Setup' window. It is also important that the 'Mode' should be set to '640 x 480 @ 30 FPS', ensuring that the webcam is operating at its maximum resolution to obtain optimum performance. Furthermore, should the user wish to tweak the image parameters such as the brightness and contrast, advanced configuration options for the camera are available. Figure 4.20 outlines the required camera settings for the eye tracker.

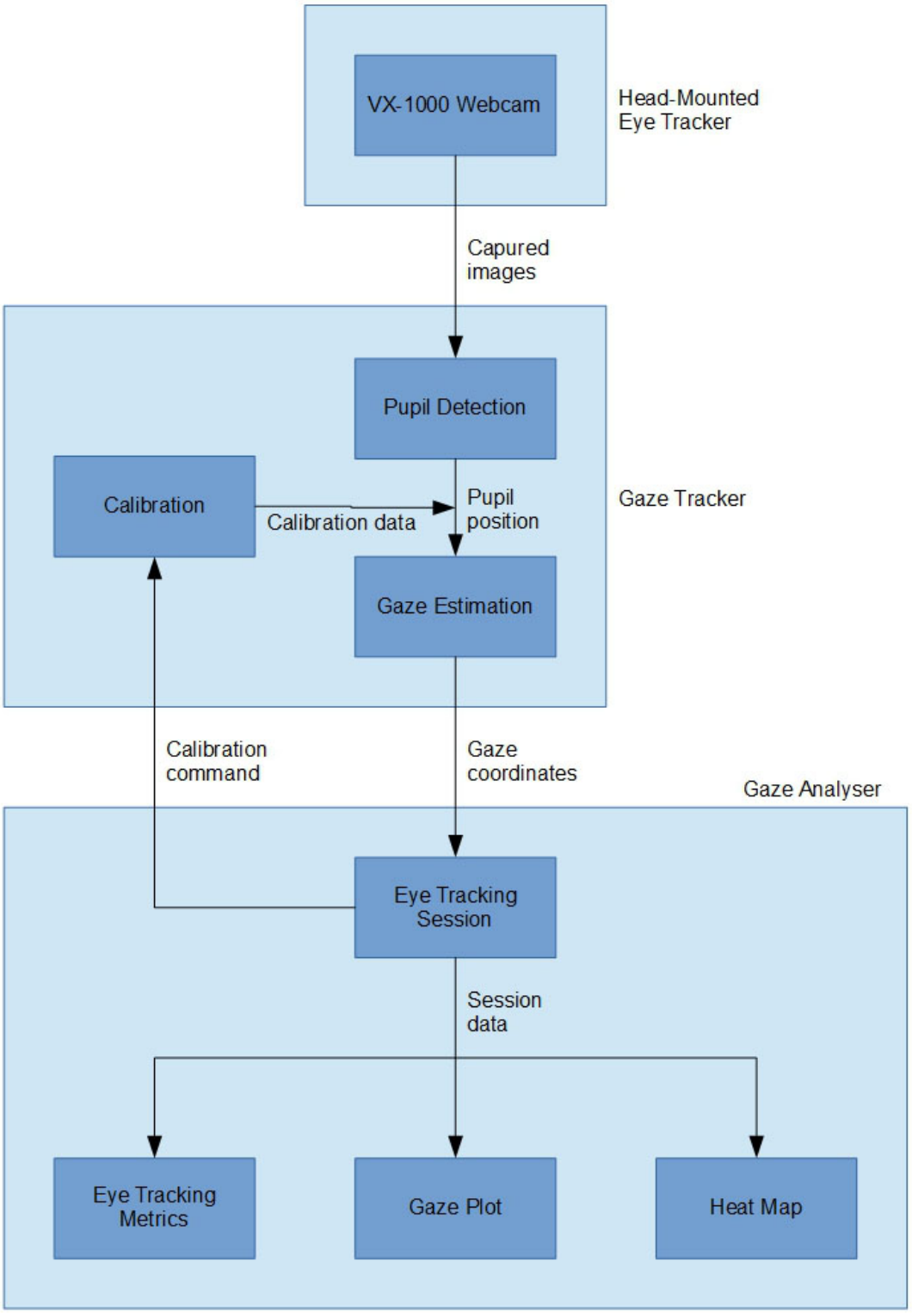


Figure 4.19: System block diagram.

Once the webcam is selected and the correct mode is set, the user can tweak the eye and pupil detection parameters in the Tracking tab. With the 'Headmounted' configuration selected, both 'Eye' and 'Pupil' tracking must be enabled in the advanced configuration options. 'Glint' tracking must also be disabled, as Gaze Tracker does not incorporate

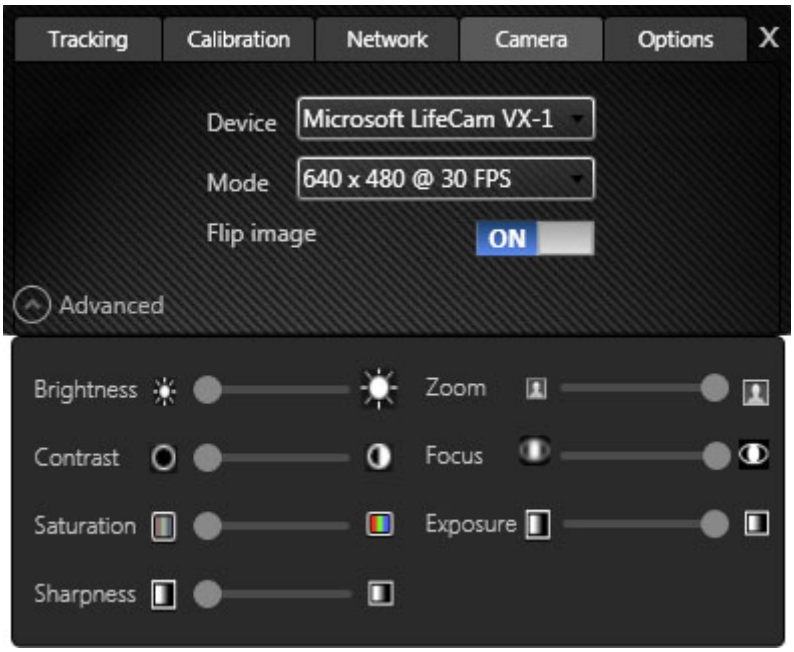


Figure 4.20: Gaze Tracker camera settings.

corneal reflection detection for increased robustness and head movement tolerance in head-mounted configurations.

Observations indicate that adjusting the 'Sensitivity' and 'Range' parameters for 'Eye' tracking in Gaze Tracker does not have a recognisable influence on eye tracking performance, thus it is recommended that they be set to 'Auto'. Conversely, these parameters have a significant influence on 'Pupil' tracking performance. The 'Sensitivity' slider adjusts the blackness threshold for the pupil, while the 'Range' parameters specify the minimum and maximum size of the pupil. The values of these parameters are likely to vary between different room lighting conditions and test subjects respectively, and should be adjusted accordingly prior to each eye tracking session. Figure 4.21 displays the configuration of these parameters for a typical user environment.

4.4.2 Network Servers

In order to enable communication with the Gaze Analyser software, both the command server and data server must be turned on in the 'Network' tab of the 'Setup' window (Figure 4.22). The command server enables control of Gaze Tracker via commands sent over TCP port 5555, while the data server enables the real-time streaming of gaze coordinates via UDP port 6666. This allows the `tcp_send()` and `udp_receive()`

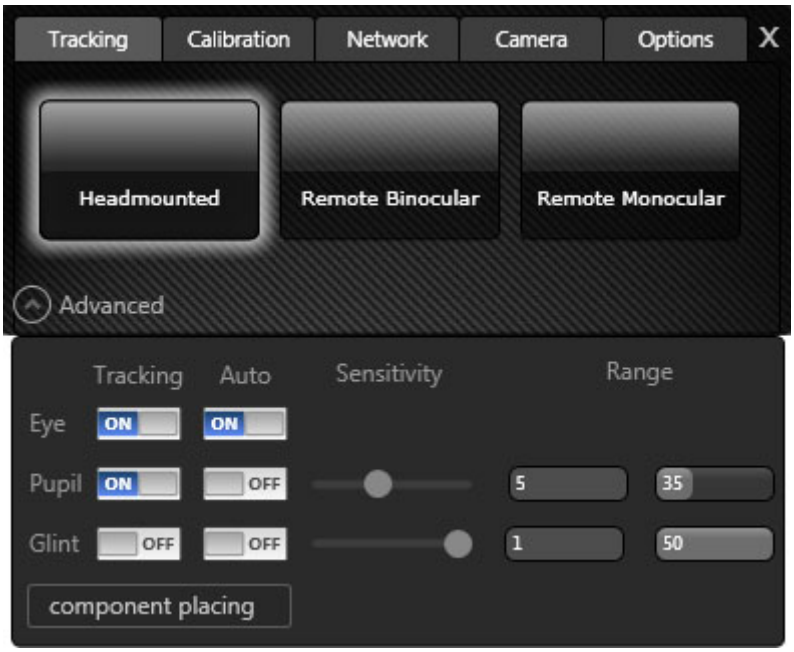


Figure 4.21: Gaze Tracker tracking settings.

functions in Gaze Analyser to connect to Gaze Tracker and perform their respective operations.

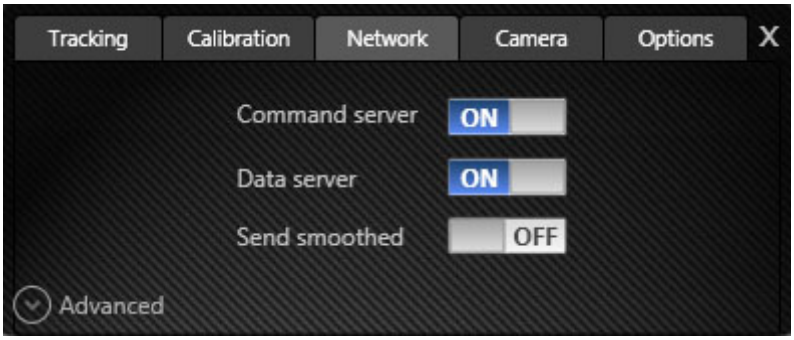


Figure 4.22: Gaze Tracker network settings.

One anomaly, or bug, in Gaze Tracker is that 'Smooth mouse' must be enabled in the advanced configuration options under the 'Options' tab in order for Gaze Tracker to properly broadcast gaze coordinates. If this option is disabled, the data server will broadcast the same point continuously, causing Gaze Analyser to identify just one fixation within the data.

Finally, exceptions for both Gaze Tracker and Gaze Analyser must be granted in Windows Firewall (or an alternative firewall package in use on the machine).

4.4.3 Calibration

Gaze Tracker's calibration procedure is arguably the most crucial element of the entire eye tracking system. As stated in Section 2.2.3, the calibration procedure translates pupil position into on-screen gaze coordinates.

The calibration procedure takes place in a full-screen window on the machine. Users are expected to trace the position of a single point around the screen with their eyes. Prior to performing a calibration procedure, users are able to specify various calibration parameters in the 'Calibration' tab in the 'Setup' window including the number of calibration points displayed during the procedure, whether or not those points are displayed in a random order, the colour of those points and the calibration window's background and the speed at which the points transition during the calibration procedure.

Generally, a greater number of calibration points will produce a more accurate calibration, but will take a longer time to complete. The calibration procedure takes approximately 20 seconds for 9 calibration points, 30 seconds for 12 calibration points and 40 seconds for 16 calibration points. When the calibration procedure is complete, the results of the calibration are presented, displaying a star rating out of 5 for the calibration and the gaze accuracy relative to the view distance (assumed to be 600 millimetres if none is provided). The dispersion of individual gaze samples at each calibration point is also displayed, providing feedback on their accuracy and precision at each calibration point. A calibration results screen for 9 calibration points is displayed in Figure 4.23.

A successful eye tracker calibration generally completes the implementation process of the hardware and software, readying it for use in an eye tracking session. A detailed analysis of calibration accuracies with respect to various eye tracking environments is presented in Section 5.3.

4.5 Summary

This chapter detailed all aspects of the head-mounted hardware design, sequentially outlining the hardware design process and presenting a step-by-step procedure of the

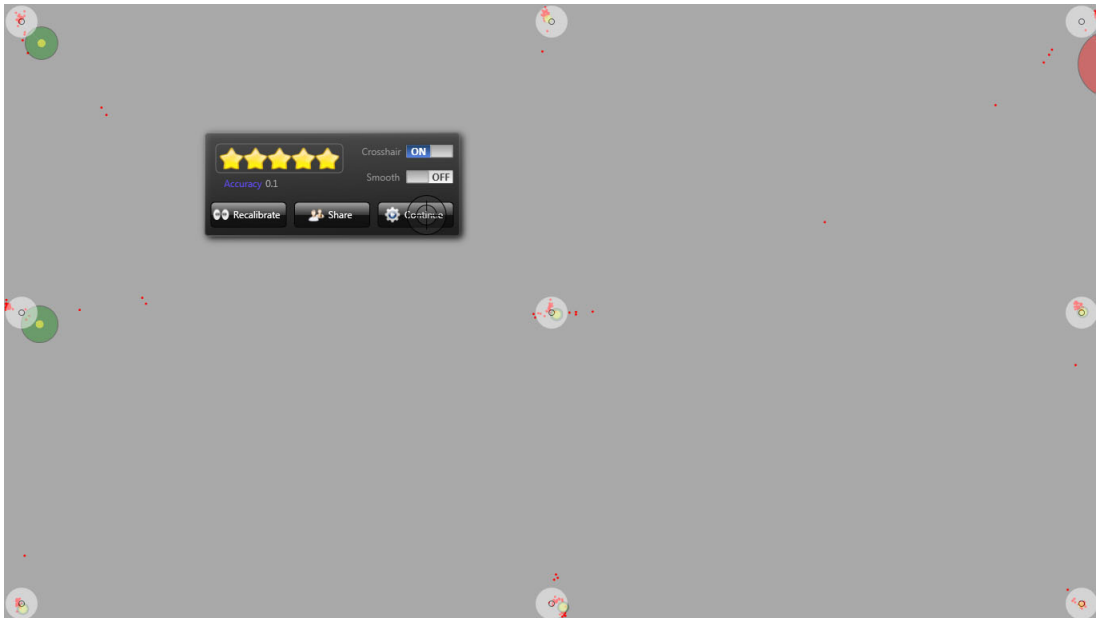


Figure 4.23: Calibration results screen.

Microsoft LifeCam VX-1000 webcam modification and assembly of the head-mounted hardware contraption. Gaze Analyser's software design and development was described for each class, detailing class functions and their implementation. Finally, the implementation of both the hardware and software with Gaze Tracker was outlined, thus creating a fully functional low-cost eye tracking system.

Chapter 5

Performance and Usability Evaluation

5.1 Outline

Analysis and evaluation of system performance and usability is a critical step in the development process. This chapter analyses and evaluates the performance and usability of the low-cost eye tracking system detailed in Chapter 4, displaying sample output generated from a given set of fixations and evaluating calibration accuracy for different user environments. Limitations and shortcomings in the system are also identified and potential approaches of mitigation are suggested. Finally, alternative uses for the system are outlined.

5.2 Sample Output

A sample output was generated from a 30 second eye tracking session with a fixation radius of 20 pixels using the University of Southern Queensland 'Engineering Research Project 2014' StudyDesk page—a learning tool used by Engineering Research Project students at the University of Southern Queensland—as stimulus. The session was conducted on a 13 inch notebook screen with a resolution of 1366 pixels wide by 768 pixels high. The fixations identified in the session are listed in Table E.1. Refer to

Appendix E for a full sample output generated by Gaze Analyser, including:

- Results/metrics text file (Figure E.1)
- Gaze plot bitmap file (Figure E.2)
- 'Number of fixations' heat map bitmap file (Figure E.3)
- 'Total fixation duration' heat map bitmap file (Figure E.4)
- 'Mean fixation duration' heat map bitmap file (Figure E.5)

5.3 Gaze Accuracy

The performance of the head-mounted hardware configuration and its implementation in the ITU Gaze Tracker software can be evaluated by its gaze accuracy. Gaze accuracy describes the angular average distance between the actual gaze point of the user and the resultant gaze coordinates measured by Gaze Tracker. A system with a high gaze accuracy is able to generate more valid results and visualisations than one with a low gaze accuracy due to the smaller on-screen error in the measured gaze coordinates. Figure 5.1 displays this phenomenon, where d describes the user's view distance to the screen, e describes the on-screen distance between the gaze point of the user and the measured gaze coordinates, and ϕ describes the angle of accuracy.

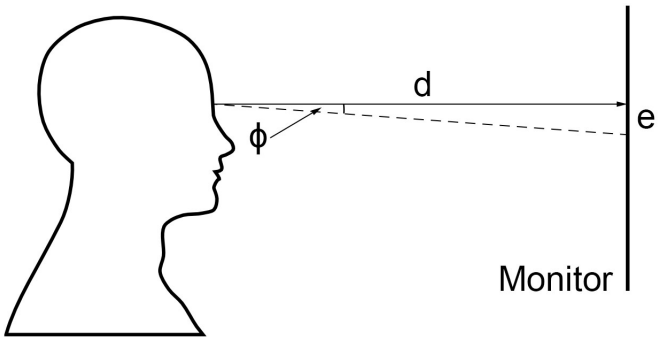


Figure 5.1: Gaze accuracy describing the angular distance between the gaze point of the user and the measured gaze coordinates.

After completing the calibration procedure, Gaze Tracker provides a feature to compute the gaze accuracy of the system on the calibration results screen (Figure 4.23). Users are able to specify their view distance to the screen in millimetres, enabling Gaze Tracker

to compute the average angle of error of all calibration points. If no view distance is specified, a distance of 600 millimetres is assumed. The use of this feature provides a means of evaluating the gaze accuracy of the low-cost eye tracking system.

A gaze accuracy test was designed to evaluate the gaze accuracy of the low-cost eye tracking system for different user environments and variables. Four variables for a typical eye tracking session were identified:

- Whether or not the test subject is wearing glasses
- Room lighting (natural/artificial and intensity)
- Screen size
- View distance to the screen

The test involved performing five consecutive calibrations in Gaze Tracker, with each calibration consisting of 16 calibration points. The resultant gaze accuracy calculated by Gaze Tracker was noted after each calibration and the mean gaze accuracy of all five results was calculated. The corresponding mean on-screen error was then calculated using Equation 5.1, with the variables defined in Figure 5.1.

$$e = d \tan \phi \tag{5.1}$$

Where ϕ is expressed in degrees.

A total of five test subjects were selected to participate in the gaze accuracy test. These test subjects ranged in age from 18 to 54 years old, with three male test subjects and two female test subjects. Two of the five test subjects wore glasses during the test. Each test subject performed the test in their native user environment using their own computer, producing a variety of room lighting conditions, screen sizes (both desktop and notebook screens) and view distances. One subject also repeated the test multiple times varying only the room lighting to evaluate the potential effects of room lighting on gaze accuracy. Finally, a chin rest was utilised in each test to stabilise the head, thus minimising head movement during all five calibration procedures in the test, the need for which is explained in Section 5.4.1. Table 5.1 displays the mean gaze accuracy and corresponding mean on-screen error for each test subject.

Test Subject	Glasses	Room Lighting	Screen Size (mm)	View Distance (mm)	Mean Accuracy (°)	Mean Error (mm)
A	No	Natural/high	23.0	550	0.68	7
		Artificial/high	23.0	550	0.74	7
		Artificial/medium	23.0	550	0.70	7
		Artificial/low	23.0	550	0.72	7
		Screen only	23.0	550	0.66	6
B	Yes	Artificial/medium	15.6	470	No results	
C	Yes	Artificial/medium	15.6	440	0.74	6
D	No	Artificial/medium	23.0	550	0.52	5
E	No	Artificial/high	15.6	520	0.64	6

Table 5.1: Gaze accuracy for different user environments.

From the results in Table 5.1, there does not appear to be any observable effect on gaze accuracy across various user environments and room lighting conditions. All tests resulted in a mean gaze accuracy between 0.4 and 0.8 degrees. Comparably, Tobii Technology (n.d.) reported gaze accuracies of 0.2 to 0.6 degrees for ideal conditions in its specification of gaze accuracy and gaze precision for the Tobii X2-30 commercial eye tracker.

It must be noted, however, that pupil detection in Gaze Tracker was not reliable for Test Subject B, thus no calibration results were obtained. The test subject's glasses were fitted with strong corrective lenses, causing the pupil to appear quite small in diameter compared to that of other test subjects. The test subject also complained of fatigue and tiredness and, at times, part of the pupil was partially obstructed by the lower eyelid. Either one or a combination of these reasons could be attributed to the issue.

Finally, for all test subjects, the precision of individual gaze samples after each calibration appeared consistent with their respective calibration points. Gaze samples were generally dispersed in a small area no larger than the physical area covered by the calibration point, indicating good precision in the eye tracker (Figure 5.2). This in-turn improves the effectiveness of the I-DT fixation identification algorithm for smaller dispersion thresholds.

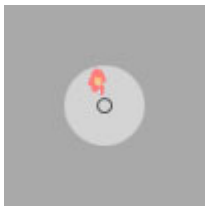


Figure 5.2: Calibration point (light grey) displaying individual gaze samples (red) dispersed over a small area, indicating good precision.

5.4 System Limitations

Being a low-cost eye tracking system developed under relatively short time constraints, there exist some limitations and shortcomings associated with both the hardware and software of the system. These limitations either hinder the system's functionality or restrict the use of the system to certain environments. This section identifies and describes those limitations and shortcomings in the system and suggests potential approaches to mitigate them with future development.

5.4.1 Head Movement Tolerance

One significant limitation with the use of a head-mounted hardware configuration in the ITU Gaze Tracker software is zero head movement tolerance. For optimum performance, a test subject's head must remain perfectly stationary from the initiation of the calibration procedure to the end of the eye tracking session. Theoretically, any deviation of the head from its position at calibration by the test subject will cause the angle of error in the calibration to significantly increase. Observations of use indicate that slight deviations in the order of a few millimetres only have a minor (but still noticeable) effect on the gaze position measured by Gaze Tracker.

To mitigate the effects of head movement in the system, a height-adjustable chin rest was constructed. A chin rest cup used for optometry applications was secured to a telescoping aluminium tube of approximately 500 millimetres in length at full extension. A section of approximately 40 millimetres in length at the bottom of the tube was flattened and bent at a perpendicular angle, enabling the chin rest to be mounted to a desk using a C-clamp. Figure 5.3 displays the chin rest mounted to a desk.

Zero head movement tolerance when using a head-mounted configuration in Gaze

Figure 5.3: Height-adjustable chin rest to stabilise the head for eye tracking.

Tracker is attributed to the lack of a point of reference captured by the camera. Some previous low-cost head-mounted configurations have accounted for this with the addition of a 'scene camera', which records the user's field of view (Babcock & Pelz 2004); however, this is not supported in Gaze Tracker and is perhaps more applicable to mobile eye tracking systems.

The implementation of a remote hardware configuration in Gaze Tracker utilises the corneal reflections on the user's eye created by fixed infrared illuminators as a point of reference, thus increasing eye tracking robustness and head-movement tolerance. This approach is briefly explored in Section 6.2.

5.4.2 Cross-Platform Support

Due to their respective utilisation of the Windows API, both the ITU Gaze Tracker and Gaze Analyser software are platform specific to the Microsoft Windows operating system, thus the eye tracking system can only be utilised on a Windows machine. With the relative widespread use of—and thus high accessibility to—the Windows operating system, this is not a significant limitation of the system.

A developer willing to support cross-platform implementation of the eye tracking system

would be required to translate the Windows API functions utilised in both Gaze Tracker and Gaze Analyser to the native API functions of the target platform. An alternative potential solution may involve exploring eye tracking software similar to Gaze Tracker which supports the target system. For example, Opengazer is an open-source eye tracking application designed for use with low-cost hardware which supports the Linux operating system (Nel 2009).

5.4.3 Software Shortcomings

Due to the time constraints of the research project, the Gaze Analyser software is not as feature-rich as it ideally should be for real eye tracking applications. While the shortcomings in the software limit its usefulness in real eye tracking applications, it provides necessary core functionality, such as fixation identification and calculation of eye tracking metrics, and additional improvements and features utilising this core functionality are able to be implemented with further work.

One significant limitation with Gaze Analyser is that gaze plot and heat map generation is restricted to a static, non-changing window. A screenshot of the desktop is obtained only at the time of initiation of the eye tracking session. Because of this, any interaction with or manipulation of a window during an eye tracking session is not reflected in the gaze plot or heat map.

A desirable feature of eye tracking analysis tools that was not implemented in Gaze Analyser is the ability to process and generate output for entire datasets of a trial. Currently, data is processed and output is generated on only a per-test subject basis. Accurate generation of visualisations for entire datasets of a trial would also be dependent on consistent window placement across all eye tracking sessions in that trial.

Finally, while it's not a limitation, the text-based user interface of Gaze Analyser may be considered aesthetically unappealing and unintuitive for novice users; however, a graphical user interface would not increase or improve the software's functionality.

5.5 Alternative Uses

While the eye tracking system was designed and developed to analyse how learning tools are used by students, alternative avenues of use for the system can be identified. The fact that the system is able to track a user's gaze position over an entire screen opens up numerous opportunities for software usability testing and HCI analysis. It could also be used to analyse a user's gaze position with respect to on-screen stimulus, such as an advertisement or slide presentation.

With the aid of the eye mouse feature in the ITU Gaze Tracker software, applications of the eye tracking system extend to gaze interaction and disability support. Users are able to utilise gaze direction as an input, controlling the position of the mouse cursor by simply fixating on areas of interest on the screen. This feature has significantly benefited severely disabled people whose only means of communication are via their eye movements. Assistive software such as GazeTalk, Dasher and StarGazer introduce various methods of eye-typing, providing a platform of communication for these people (San Agustin et al. 2010).

5.6 Summary

This chapter analysed and evaluated the performance and usability of the low-cost eye tracking system, displaying sample output generated from a given set of fixations and evaluating calibration accuracy for different user environments. It was shown that the system maintained excellent gaze accuracies across different user environments and room lighting conditions, however, usability experiences varied for one test subject. Limitations and shortcomings in the system were also identified and potential approaches of mitigation were suggested. Finally, alternative uses for the system were outlined, extending the potential applications of the system.

Chapter 6

Conclusions and Further Work

6.1 Achievement of Project Objectives

Prior to the commencement of this research project, a concise list of project objectives were identified and stipulated. These objects are outlined in the Project Specification in Appendix A. This section summarises the dissertation by evaluating the achievement of those project objectives.

Firstly, background information on eye tracking was researched and, subsequently, eye tracking applications throughout various industries and fields of research, as well as the cost and accessibility issues of commercial eye trackers, were outlined. This set the motivation behind the development of the low-cost eye tracking system, as addressed in Chapter 1.

Succeeding that, in-depth research was conducted in the form of a comprehensive literature review in Chapter 2, detailing all aspects of the research project. This included the methods of eye tracking with respect to low-cost hardware and open-source software, as well as various algorithms to identify fixations within the data in order to perform analysis and visualisations.

The knowledge gained from the literature review enabled the analysis of both the hardware and software requirements of the low-cost eye tracking system, thus the methodology for the design and development of the system was formed in Chapter 3. This led to the identification and selection of a low-cost hardware configuration and supporting

open-source software in the ITU Gaze Tracker. Further software requirements were also identified for the analysis and visualisation of eye tracking data.

From there, the low-cost eye tracking system was designed and implemented as detailed in Chapter 4. A head-mounted hardware configuration was constructed and assembled by suitably modifying a low-cost webcam and mounting it to a pair of safety glasses for a total cost of 49.36 Australian dollars. Data analysis software informally named Gaze Analyser was also designed and programmed to identify fixations within the eye tracking data, calculate eye tracking metrics and generate visualisations. The hardware and software were then implemented via Gaze Tracker to form the system.

Finally, the system's performance and usability was critically evaluated in Chapter 5. Sample output was generated from a given set of fixations. Gaze accuracy testing was conducted to evaluate the performance of the hardware and its implementation in Gaze Tracker for various user environments and variables. Limitations in the system were also identified and explained, including head movement tolerance, cross-platform support and shortcomings in the software. Potential approaches of mitigation were suggested for these limitations. Lastly, potential alternative applications of use for the system were explored.

6.2 Further Work

The completion of this research project enables numerous possibilities of further work on the topic of low-cost eye tracking. Many of these possibilities stem from the limitations identified in Section 5.4.

To address the lack of head movement tolerance using a head-mounted hardware configuration with Gaze Tracker, a low-cost remote configuration could be implemented as explained in Section 5.4.1. Potential solutions were briefly explored prior to undertaking the requirements analysis of this research project. A low-cost remote configuration would make use of an appropriate high definition webcam modified to utilise a lens with a longer focal length to produce a more focused and detailed image of the eyes. An M12 threaded lens with a 16 millimetre focal length and an M12 lens holder fitted to the webcam's PCB could prove to be successful in achieving this. Infrared illuminators would also be required to create the dark pupil effect, as with a head-mounted

configuration.

Further to this, functionality of Gaze Analyser could be improved and extended in order to address shortcomings in the software. Support for the generation of visualisations for window events such as scrolling would significantly increase the potential applications of the software, although this feature could be challenging to implement. Support for the generation of results for entire datasets of a trial would also improve the software's usefulness in eye tracking studies, allowing for a more comprehensive analysis of eye tracking data.

The eye tracking system could also be adapted to support multiple platforms. While the most obvious candidates for cross-platform support include alternative desktop operating systems such as Linux and Mac OS, the recent surge in popularity of mobile devices and tablets provide motivation for the development of a low-cost eye tracking system compatible with these devices.

Finally, if the limitations and shortcomings in the system are addressed as described in this section, the eye tracker could be utilised in numerous eye tracking studies by researchers from fields that would benefit from a low-cost solution.

References

Babcock, J. S. & Pelz, J. B. (2004), 'Building a lightweight eyetracking headgear', *Proceedings of the 2004 Symposium on Eye-Tracking Research & Applications* pp. 109–114. [Online; accessed May-2014].

Blignaut, P. (2009), 'Fixation identification: The optimum threshold for a dispersion algorithm', *Attention, Perception, & Psychophysics* **71**(4), 881–895. [Online; accessed October-2014].

Blignaut, P. (2010), 'Visual span and other parameters for the generation of heatmaps', *Proceedings of the 2010 Symposium on Eye-Tracking Research & Applications* pp. 125–128. [Online; accessed June-2014].

Cooke, L. (2005), 'Eye tracking: How it works and how it relates to usability', *Technical Communication* **52**(4), 456–463. [Online; accessed May-2014].

Ehmke, C. & Wilson, S. (2007), 'Identifying web usability problems from eye tracking data', *Proceedings of the 21st British HCI Group Annual Conference on People and Computers: HCI...but not as we know it* **1**, 119–128. [Online; accessed June-2014].

Engineers Australia (2010), 'Code of Ethics', `http://www.engineersaustralia.org.au//sites/default/files/shado/About%20Us/Overview/Governance/codeofethics2010.pdf`. [Online; accessed June-2014].

FreeTrack Forum User Gian92 (2012), 'Spectral analysis of home-made filters and ps3 eye filter', `http://forum.free-track.net/index.php?showtopic=3205&hl=filter`. [Online; accessed May-2014].

Goldberg, J. H. & Wichansky, A. M. (2003), 'Eye tracking in usability evaluation: A practitioner's guide', *The Mind's Eye: Cognitive and Applied Aspects of Eye Movement Research* pp. 493–516. [Online; accessed May-2014].

Hansen, D. W. & Pece, A. E. C. (2004), 'Eye tracking in the wild', *Computer Vision and Image Understanding* **98**, 155–181. [Online; accessed May-2014].

ITU GazeGroup (n.d.), *ITU Gaze Tracker: A Brief Users Guide (v1.0b).* [Online; accessed March-2014].

Jacob, R. J. K. & Karn, K. S. (2003), 'Eye tracking in human-computer interaction and usability research: Ready to deliver the promises', *The Mind's Eye: Cognitive and Applied Aspects of Eye Movement Research* pp. 573–603. [Online; accessed June-2014].

Johansen, S. A. & Hansen, J. P. (2006), 'Do we need eye trackers to tell where people look?', *CHI '06 Extended Abstracts on Human Factors in Computing Systems* pp. 923–928. [Online; accessed May-2014].

Kowalik, M. (2010), *How to Build Low Cost Eye Tracking Glasses for Head Mounted System.* [Online; accessed June-2014].

Kumar, M. (2007), GUIDe saccade detection and smoothing algorithm, Technical report, Stanford University, HCI Group. [Online; accessed October-2014].

Kumar, M., Klingner, J., Puranik, R., Winograd, T. & Paepcke, A. (2008), 'Improving the accuracy of gaze input for interaction', *Proceedings of the 2008 Symposium on Eye-Tracking Research & Applications* pp. 65–68. [Online; accessed October-2014].

Li, D., Babcock, J. & Parkhurst, D. J. (2006), 'openEyes: A low-cost head-mounted eye-tracking solution', *Proceedings of the 2006 Symposium on Eye-Tracking Research & Applications* pp. 95–100. [Online; accessed May-2014].

Mantiuk, R., Kowalik, M., Nowosielski, A. & Bazyluk, B. (2012), 'Do-it-yourself eye tracker: Low-cost pupil-based eye tracker for computer graphics applications', *Proceedings of the 18th international conference on Advances in Multimedia Modeling* pp. 115–125. [Online; accessed May-2014].

Microsoft LifeCam VX-1000 Technical Data Sheet (2011). [Online; accessed April-2014].

Morimoto, C. H., Koons, D., Amir, A., Flickner, M. & Zhai, S. (1999), 'Keeping an eye for HCI', *Proceedings of the XII Brazilian Symposium on Computer Graphics and Image Processing* pp. 171–176. [Online; accessed May-2014].

Mulvey, F., Villanueva, A., Sliney, D., Lange, R., Cotmore, S. & Donegan, M. (2008), Exploration of safety issues in eyetracking, Technical Report D5.4, COGAIN. [Online; accessed May-2014].

Nel, E.-M. (2009), 'Opengazer: open-source gaze tracker for ordinary webcams', `http://www.inference.phy.cam.ac.uk/opengazer/`. [Online; accessed February-2014].

Poole, A. & Ball, L. J. (2005), 'Eye tracking in human-computer interaction and usability research: Current status and future prospects', *Encyclopedia of Human Computer Interaction* . [Online; accessed May-2014].

r3dux.org (2013), 'How much does a Tobii X2-30 or X2-60 eye tracker cost?', `http://r3dux.org/2013/03/how-much-does-a-tobii-x2-30-or-x2-60-eye-tracker-cost/`. [Online; accessed March-2014].

Salvucci, D. D. & Goldberg, J. H. (2000), 'Identifying fixations and saccades in eye-tracking protocols', *Proceedings of the 2000 Symposium on Eye-Tracking Research & Applications* pp. 71–78. [Online; accessed September-2014].

San Agustin, J. (2009), Off-the-Shelf Gaze Interaction, PhD thesis, IT University of Copenhagen. [Online; accessed May-2014].

San Agustin, J., Skovsgaard, H., Mollenbach, E., Barret, M., Tall, M., Hansen, D. W. & Hansen, J. P. (2010), 'Evaluation of a low-cost open-source gaze tracker', *Proceedings of the 2010 Symposium on Eye-Tracking Research & Applications* pp. 77–80. [Online; accessed May-2014].

Tobii Technology (2010), 'Tobii Studio^{TM} 2.X', `http://www.tobii.com/Global/Analysis/Downloads/Product_Descriptions/Tobii_Studio_2.2_Product_Description.pdf?epslanguage=en`. [Online; accessed June-2014].

Tobii Technology (2013), 'Eye tracking research', `http://www.tobii.com/en/eye-tracking-research/global/research/`. [Online; accessed June-2014].

Tobii Technology (2014), 'Tobii X2 product description', `http://www.tobii.com/Global/Analysis/Downloads/Product_Descriptions/Tobii_X2_Product_Description.pdf?epslanguage=en`. [Online; accessed March-2014].

Tobii Technology (n.d.), 'Specification of gaze accuracy and gaze precision, Tobii X2-30 eye tracker', `http://www.tobii.com/Global/Analysis/Downloads/Product_Descriptions/Tobii_X2-30_Eye_Tracker_Technical_Specification.pdf`. [Online; accessed October-2014].

Wang, H., Chignell, M. & Ishizuka, M. (2006), 'Empathic tutoring software agents using real-time eye tracking', *Proceedings of the 2006 Symposium on Eye-Tracking Research & Applications* pp. 73–78. [Online; accessed May-2014].

Wikipedia (2014), 'Eye tracking — Wikipedia, the free encyclopedia', `http://en.wikipedia.org/w/index.php?title=Eye_tracking&oldid=620486171`. [Online; accessed May-2014].

Appendix A

Project Specification

ENG 4111/2 Research Project

Project Specification

For:	**Thomas Bradford**
Topic:	Low Cost Eye Tracker
Supervisors:	Alexander Kist
Project Aim:	To develop a low cost eye tracking system using inexpensive hardware and open source software that can be used to analyse how learning tools are used by students.

Program:

1. Research background information on eye tracking including currently available solutions, its uses, implementation costs and feasibility issues, as well as the benefits of intuitive learning tools on students.
2. Undertake a basic requirements analysis.
3. Undertake a comprehensive literature review covering all aspects of the project including eye tracking, methods of analysing the eye tracking data and how the analysis can be used to improve learning tools.
4. Identify appropriate hardware and software for use with the system, such as the ITU Gaze Tracker software and a supported webcam.
5. Identify methods of eye tracking analysis (e.g. heat maps, pupil path mapping, recognising trends between students)
6. Design the system.
7. Program and implement the system.
8. Critically analyse system performance and perform system testing, making improvements where necessary.

As time and resources permit:

1. Identify alternative uses for the system.
2. Consider multi-platform implementation (e.g. smartphones and tablets)

Agreed:

Student Name:	Thomas Bradford
Date:	17/03/2014
Supervisor Name:	Alexander Kist
Date:	
Examiner:	Chris Snook
Date:	

Appendix B

Preliminary Methodology

B.1 Risk Assessment

Hazard: Eye Strain

Extended use of a computer can cause strain on the eyes due to maintaining focus on the screen, light emitted by the screen and movement of the eye around the screen. As this project involves testing an eye tracking system, further eye strain may be caused by deliberate eye movements made while testing.

Risk: Headaches

Mitigation: Take regular short breaks from using a computer, look out a window, perform eye exercises

Hazard: Repetitive Movement

Repetitive movements such as typing for extended periods can cause repetitive strain injury and aching in the joints and muscles.

Risk: Repetitive strain injury, joint and muscle aches

Mitigation: Take regular short breaks from using a computer, ensure appropriate seat position and eye level, use ergonomic peripherals, stretch and exercise the affected areas

Hazard: Use of Hand Tools

Hand tools such as knives, scissors, screwdrivers and soldering irons can cause significant wounds due to skin penetration by sharp blades or points, particularly if handled incorrectly. Furthermore, the intense heat emitted by a soldering iron can cause significant burns to the user.

Risk: Minor to moderate wounds, incisions or burns

Mitigation: Always handle tools correctly and with care, ensure all blades are protected with a guard, organise tools appropriately and replace them in their correct position immediately after use. Ensure soldering iron is switched off and mounted in its holder when not in use

Hazard: Infrared Light Exposure

This project involves direct illumination of the eye using infrared light. As reported in Section 3.3.1, maximum recommended irradiance levels for infrared light is 10mW/cm^2. This hazard applies to both execution of project work and post-completion of the project.

Risk: Eye strain or soreness, temporary or permanent blurred vision, temporary or permanent retina damage causing partial blindness

Mitigation: Always ensure infrared light configuration is below the maximum recommended irradiance level of 10mW/cm^2, check datasheets and specifications of lights before use, do not use the eye tracker for extended periods of time

Hazard: Workload

Being engaged in full-time employment while undertaking project work, as well as other courses and external commitments simultaneously can cause high workload.

Risk: Stress, failure to meet deadlines

Mitigation: Efficient and effective time management, plan project work in advance

Hazard: Psychological Stress

Psychological stress can be caused by a number of factors including high workload, personal issues, work-related commitments or approaching deadlines. It can be classified as acute (short-term) or chronic (long-term). Acute stress can be beneficial for productivity, however, if prolonged it can lead to chronic stress.

Risk: Psychological illness such as anxiety and depression, vulnerability of the immune system leading to physical illness, lack of sleep

Mitigation: Stress management techniques (depending on the individual), efficient and effective time management, take regular short breaks from stress inducing activities

Hazard	Exposure	Likelihood	Severity	Risk Level
Eye strain	Frequently	Almost certain	Minor	Moderate
Repetitive movement	Frequently	Likely	Moderate	Moderate
Use of hand tools	Occasionally	Unlikely	Moderate	Moderate
Infrared light exposure	Occasionally	Unlikely	Major	Moderate
Psychological stress	Rarely	Possible	Moderate	Moderate
Workload	Occasionally	Likely	Minor	Moderate

Table B.1: Risk assessment table.

B.2 Project Timeline

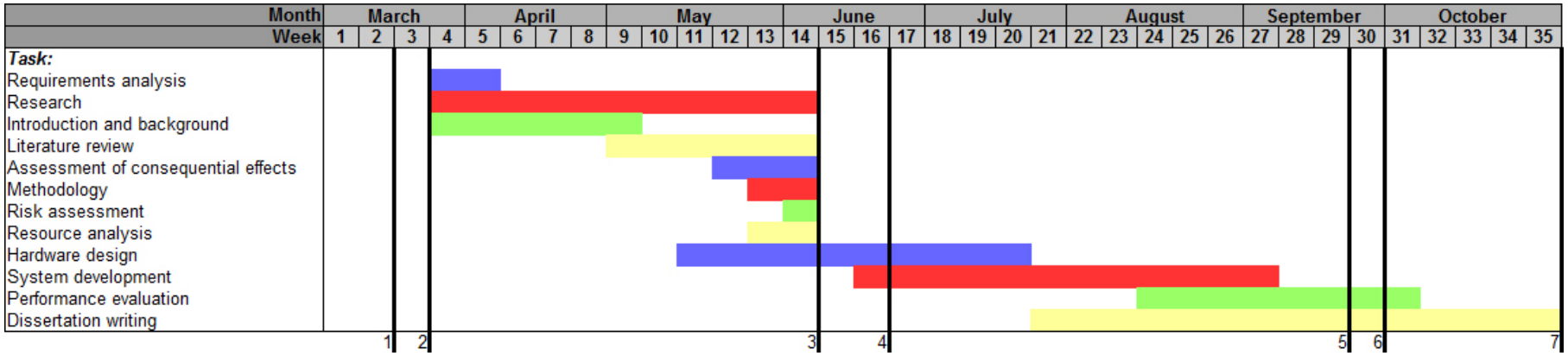


Figure B.1: Graphical timeline of project work.

No.	Item	Due Date
1	Topic allocation	12 March 2014
2	Project specification	19 March 2014
3	Preliminary report	4 June 2014
4	Progress assessment	18 June 2014
5	Partial draft dissertation	17 September 2014
6	Seminar presentation	22-26 September 2014
7	Final dissertation	30 October 2014

Table B.2: Key dates.

Appendix C

Data Sheets

Microsoft® LifeCam VX-1000

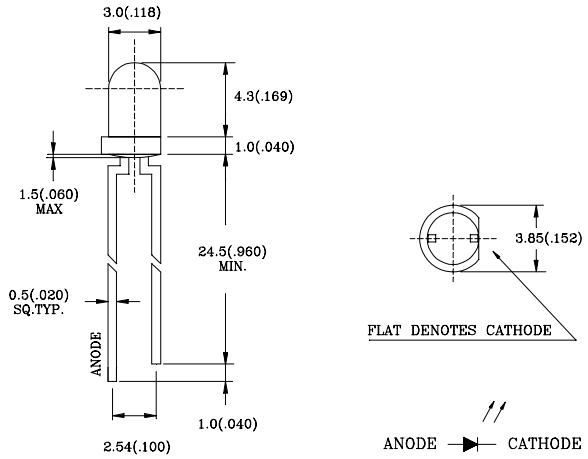
Easy to Use | Built-in Microphone

Technical Data Sheet

Version Information	
Product Name	Microsoft® LifeCam VX-1000
Product Version	Microsoft LifeCam VX-1000
Webcam Version	Microsoft LifeCam VX-1000
Product Dimensions	
Webcam Length	2.09 inches (53.1 millimeters)
Webcam Width	2.18 inches (55.5 millimeters)
Webcam Depth/Height	2.70 inches (68.8 millimeters)
Webcam Weight	3.36 ounces (95.3 grams)
Webcam Cable Length	72.0 inches +6/-0 inches (1828 millimeters +152/-0 millimeters)
Compatibility and Localization	
Interface	Full-speed USB compatible with the USB 2.0 specification
Operating Systems	Microsoft Windows® 7, Windows Vista®, and Windows XP with Service Pack 2 (excluding Windows XP 64-bit)
Top-line System Requirements	Requires a PC that meets the requirements for and has installed one of these operating systems: • Microsoft Windows 7, Windows Vista, or Windows XP with Service Pack 2 (excluding Windows XP 64-bit) • Intel Pentium® III 550 MHz (Intel Pentium 4 1.4 GHz recommended) • 256 MB of RAM • 300-700 MB hard drive space • Display adapter capable 16-bit color depth or higher • 2 MB or more video memory • Windows-compatible speakers or headphones • USB port 1.1 (USB 2.0 recommended) • CD ROM drive • Broadband internet access required, access fees may apply • Microsoft LifeCam software version 2.07 Internet functions (post to Windows Live™ Spaces, send in e-mail, video calls), also require: Internet Explorer 6/7 browser software required for installation; 25 MB hard drive space typically required (users can maintain other default Web browsers after installation) The Microsoft LifeCam The Microsoft LifeCam VX-1000 has basic Video & Audio Functionality with Windows Live Messenger, AOL® Instant Messenger™, Yahoo!® Messenger, Skype, and Microsoft Office Communicator
Compatibility Logos	• Works with Microsoft Windows Vista • Certified USB logo
Software Localization	Microsoft LifeCam software, version 2.07 may be installed in Simplified Chinese, Traditional Chinese, English, French, German, Italian, Japanese, Korean, Brazilian Portuguese, Iberian Portuguese, or Spanish. If available, standard setup will install the software in the default OS language. Otherwise, the English language version will be installed.
Windows Live™ Integration Features	
Video Conversation Feature	Windows Live call button delivers one touch access to video conversation
Call Button Life	10,000 actuations
Webcam Controls & Effects	LifeCam Dashboard provides access to animated video special effect features and webcam controls
Blogging Feature	Add photos to Windows Live Spaces with one mouse click
Imaging Features	
Sensor	CMOS VGA sensor technology
Resolution	• Motion Video: 0.31 megapixel (640 x 480 pixels)* • Still Image: 0.31 megapixel (640 x 480 pixels) without interpolation*
Field of View	55° diagonal field of view
Imaging Features	• Manual focus • Automatic image adjustment with manual override
Product Feature Performance	
Audio Features	Integrated microphone
Microphone technology	Omni directional microphone
Mounting Features	• Desktop and CRT universal attachment base • Notebook and LCD universal attachment base
Storage Temperature & Humidity	-40 °F (-40 °C) to 140 °F (60 °C) at <5% to 65% relative humidity (non-condensing)
Operating Temperature & Humidity	32° F (0° C) to 140° F (60° C) at <5% to 80% relative humidity (non-condensing)
Certification Information	
Country of Manufacture	People's Republic of China (PRC)
ISO 9001 Qualified Manufacturer	Yes
ISO 14001 Qualified Manufacturer	Yes
Restriction on Hazardous Substances	This device complies with all applicable worldwide regulations and restrictions including, but not limited to: EU directive 2002/95/EC on the Restriction of the Use of Certain Hazardous Substances in Electrical and Electronic Equipment and EU Registration Evaluation and Authorization of Chemicals (REACH) regulation regarding Substances of Very High Concern.
FCC ID	This device complies with part 15 of the FCC Rules and Industry Canada ICES-003. Operation is subject to the following two conditions: (1) This device may not cause harmful interference, and (2) this device must accept any interference received, including interference that may cause undesired operation. Tested to comply with FCC standards. For home and office use. Model number: 1080, LifeCam VX-1000.
Agency and Regulatory Marks	• ACA/MED Declaration of Conformity (Australia and New Zealand) • ICES-003 report on file (Canada) • EIP Pollution Control Mark, EPUP (China) • CE Declaration of Conformity, Safety and EMC (European Union) • WEEE (European Union) • VCCI Certificate (Japan) • CITC Letter (Kingdom of Saudi Arabia) • MIC Certificate (Korea) • GOST Certificate (Russia) • FCC Declaration of Conformity (USA) • UL and cUL Listed Accessory (USA and Canada) • CB Scheme Certificate (International)
Windows Hardware Quality Labs (WHQL)	ID: 1389818 Microsoft Windows 7

* One megapixel = 1,000,000 pixels. Lower resolution available when sending video via instant messaging.

◆**PACKAGE DIMENSIONS**



◆ABSOLUTE MAXIMUN RATING: (Ta=25℃)

Part No.	P_D（mw）	V_R（V）	Topr	Tstg
ZD1946	100	5	-35℃ to 85℃	-35℃ to 85℃
PARAMETER	Power Dissipation	Reverse Voltage	Operating Temperature Range	Storage Temperature Range
Lead Soldering Temperature { 1.6mm (0.063 inch) From Body } 250℃±5℃For 3 Seconds				

◆ELECTRO-OPTICAL CHARACTERISTICS: (Ta=25℃)

Part No.	V_F（V）			I_R（uA）			λ_P（nm）			2θ1/2（Age）			Ie（mw/sr）		
	Min	Typ	Max	Min	Typ	Max	Min	Typ	Max	Min	Typ	Max	Min	Typ	Max
L-316EIR1C		1.2 1.4	1.6			10		940			30		6	12	
TEST CONDITION	I_F=20mA IF=100mA			V_R= 5V			I_F= 20mA			I_F =20mA			I_F= 20mA		

BC=BLUE CLEAR

1. All dimension are in millimeter (inches).
2. Tolerance is ±0.25mm(0.01”)unless otherwise specified.

Appendix D

Source Code Listing

D.1 main.cpp

```
/**
 * @file main.cpp
 * @author Thomas Bradford (thomas.bradford91@gmail.com)
 *
 * Main function for Gaze Analyser
 * Controls program with text-driven menu and switch-case
    statements
 * To be used in conjunction with ITU Gaze Tracker
 *
 * @return 0 on program exit
 */

#include <iostream>
#include <sstream>
#include <string>
#include <windows.h>

#include "global.hpp"
#include "Session.hpp"

using namespace std;

int main()
{
    string input;
    int mode;

    res_width = GetSystemMetrics(SM_CXSCREEN);
    res_height = GetSystemMetrics(SM_CYSCREEN);

    CreateDirectory(".\\Data\\", NULL);

    cout << "Gaze Analyser 1.0a\n";
    cout << "------------------\n";

    while(TRUE) {
        cout << "\n";
        cout << "1: Start a new eye tracking session\n";
        cout << "2: Exit\n";
        cout << "Enter your selection: ";

        getline(cin, input);
        cout << "\n";
        istringstream ss(input);
        ss >> mode;

        switch(mode) {
        case 1: {       // New eye tracking session
            cout << "New eye tracking session started.\n";
```

```
            cout << "\n";
            Session session;
        } break;

        case 2:          // Exit program
            cout << "Press Enter to exit the program.";
            cin.get();
            return 0;

        default:
            cout << "Invalid input! Expected inputs: 1-2.\n";
            break;
        }
    }
}
```

D.2 global.hpp

```
/**
 * @file global.hpp
 * @author Thomas Bradford (thomas.bradford91@gmail.com)
 *
 * Global variable definition
 */

#ifndef GLOBAL_HPP
#define GLOBAL_HPP

extern int res_width;
extern int res_height;
extern bool track;
extern const int t_fix_min;

#endif // GLOBAL_HPP
```

D.3 global.cpp

```
/**
 * @file global.cpp
 * @author Thomas Bradford (thomas.bradford91@gmail.com)
 *
 * Global variable declaration
 */

#include "global.hpp"

int res_width;                  // Resolution width (px)
int res_height;                 // Resolution height (px)
bool track;                     // Track boolean for tracking
    thread
```

```
const int t_fix_min = 100;        // Minimum fixation duration
    constant
```

D.4 tcp_send.cpp

```
/**
 * @file tcp_send.cpp
 * @author Thomas Bradford (thomas.bradford91@gmail.com)
 *
 * Sends command via the TCP port defined in PORT
 *
 * @return 0 if successful, 1 if unsuccessful
 */

#include <iostream>
#include <string>
#include <windows.h>
#include <winsock2.h>

#define IP "127.0.0.1"
#define PORT 5555

using namespace std;

int tcp_send(string command)
{
    SOCKET s;
    SOCKADDR_IN server;
    int send_len;
    WSADATA wsa;

    // Initialise Winsock
    if (WSAStartup(MAKEWORD(2,2), &wsa) != 0) {
        cout << "Error initialising Winsock. Error code: " <<
            WSAGetLastError() << "\n";
        return 1;
    }

    // Create TCP socket
    s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (s == INVALID_SOCKET) {
        cout << "Error opening socket. Error code: " <<
            WSAGetLastError() << "\n";
        WSACleanup();
        return 1;
    }

    // Prepare SOCKADDR_IN structure
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = inet_addr(IP);
    server.sin_port = htons(PORT);
```

```
    // Connect socket
    if (connect(s, (sockaddr *) &server, sizeof(server)) ==
       SOCKET_ERROR) {
        cout << "Error connecting socket. Error code: " <<
            WSAGetLastError() << "\n";
        closesocket(s);
        WSACleanup();
        return 1;
    }

    // Send command
    send_len = send(s, command.c_str(), command.length(), 0);
    if (send_len == SOCKET_ERROR) {
        cout << "Error sending command. Error code: " <<
            WSAGetLastError() << "\n";
        closesocket(s);
        WSACleanup();
        return 1;
    }

    // Close socket and clean up
    closesocket(s);
    WSACleanup();
    return 0;
}
```

D.5 udp_receive.cpp

```
/**
 * @file udp_receive.cpp
 * @author Thomas Bradford (thomas.bradford91@gmail.com)
 *
 * Receives data sent from UDP port defined in PORT.
 * To be called in a new thread.
 * Receives while track boolean is true.
 *
 * @param *arg: a pointer to the ofstream to write to
 */

#include <fstream>
#include <iostream>
#include <windows.h>
#include <winsock2.h>

#include "global.hpp"

#define BUFLEN 512
#define PORT 6666

using namespace std;
```

```
void udp_receive(void *arg)
{
    SOCKET s;
    SOCKADDR_IN server;
    SOCKADDR_IN si_other;
    char buf[BUFLEN];
    int recv_len;
    int slen;
    WSADATA wsa;
    ofstream *file;

    slen = sizeof(si_other);

    // Initialise Winsock
    if (WSAStartup(MAKEWORD(2,2), &wsa) != 0) {
        cout << "Error initialising Winsock. Error code: " <<
            WSAGetLastError() << "\n";
        return;
    }

    // Create UDP socket
    s = socket(AF_INET, SOCK_DGRAM, 0);
    if (s == INVALID_SOCKET) {
        cout << "Error opening socket. Error code: " <<
            WSAGetLastError() << "\n";
        WSACleanup();
        return;
    }

    // Prepare SOCKADDR_IN structure
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons(PORT);

    // Bind socket
    if (bind(s, (SOCKADDR *) &server, sizeof(server)) ==
       SOCKET_ERROR){
        cout << "Error binding socket. Error code: " <<
            WSAGetLastError() << "\n";
        closesocket(s);
        WSACleanup();
        return;
    }

    file = (ofstream*)arg;

    // Listen for data
    while(track){
        fflush(stdout);
        memset(buf, '\0', BUFLEN);
```

```
        recv_len = recvfrom(s, buf, BUFLEN, 0, (SOCKADDR *) &
            si_other, &slen);
        if (recv_len == SOCKET_ERROR) {
            cout << "Error reading from socket. Error code: "
                << WSAGetLastError() << "\n";
            closesocket(s);
            WSACleanup();
            return;
        }

        *file << buf << "\n";
    }

    // Close socket and clean up
    closesocket(s);
    WSACleanup();
}
```

D.6 screenshot.cpp

```
/**
 * @file screenshot.cpp
 * @author Thomas Bradford (thomas.bradford91@gmail.com)
 *
 * Captures the current desktop screen to a device context
 *
 * @return hCaptureDC: device context of the desktop screen
 */

#include <string>
#include <windows.h>

#include "global.hpp"

using namespace std;

HDC screenshot()
{
    HWND hDesktopWnd;
    HDC hDesktopDC;
    HDC hCaptureDC;
    HBITMAP hCaptureBmp;

    // Capture desktop screen
    hDesktopWnd = GetDesktopWindow();
    hDesktopDC = GetDC(hDesktopWnd);
    hCaptureDC = CreateCompatibleDC(hDesktopDC);
    hCaptureBmp = CreateCompatibleBitmap(hDesktopDC, res_width
        , res_height);
    SelectObject(hCaptureDC, hCaptureBmp);
```

```
    BitBlt(hCaptureDC, 0, 0, res_width, res_height, hDesktopDC
        , 0, 0, SRCCOPY | CAPTUREBLT);

    // Clean up
    ReleaseDC(hDesktopWnd, hDesktopDC);
    DeleteObject(hCaptureBmp);

    return hCaptureDC;
}
```

D.7 Point.cpp

```
/**
 * @file Point.cpp
 * @author Thomas Bradford (thomas.bradford91@gmail.com)
 *
 * Point struct implementation
 */

#include "Point.hpp"

/**
 * Point constructor
 *
 * @param x: X coordinate of point
 * @param y: Y coordinate of point
 * @param t: Time stamp or duration of point
 */
Point::Point(double x, double y, long long t): x(x), y(y), t(t
    ) {}
```

D.8 Session.cpp

```
/**
 * @file Session.cpp
 * @author Thomas Bradford (thomas.bradford91@gmail.com)
 *
 * Session class implementation
 */

#include <cmath>
#include <ctime>
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>
#include <vector>
#include <process.h>
#include <windows.h>
```

```
#include "global.hpp"
#include "Session.hpp"
#include "Metrics.hpp"
#include "GazePlot.hpp"
#include "HeatMap.hpp"

using namespace std;

int tcp_send(string);
void udp_receive(void *);
HDC screenshot();

/**
 * Gets the details of the current session from user input
 */
void Session::get_details()
{
    string input;

    // Prompt for trial name and check for valid input
    do {
        cout << "Enter the name of the trial: ";
        getline(cin, trial);
        if (trial.length() == 0) {
            cout << "Trial name cannot be blank!\n";
        }
    } while (trial.length() == 0);

    // Create directory for trial
    CreateDirectory((".\\Data\\"+trial+"\\").c_str(), NULL);

    // Prompt for test subject name and check for valid input
    do {
        cout << "Enter the name of the test subject: ";
        getline(cin, subject);
        if (subject.length() == 0) {
            cout << "Test subject name cannot be blank!\n";
        }
        else if (ifstream((".\\Data\\"+trial+"\\"+subject+"
            _Results.txt").c_str())) {
            cout << "Test subject " << subject << " already
                exists for trial " << trial <<"!\n";
        }
    } while (subject.length() == 0 || ifstream((".\\Data\\"+
        trial+"\\"+subject+"_Results.txt").c_str()));

    // Prompt for fixation radius and check for valid input
    do {
        cout << "Enter the desired fixation radius in pixels:
            ";
        getline(cin, input);
        istringstream ss(input);
```

```
        ss >> fix_radius;
        if (fix_radius <= 0) {
            cout << "Invalid fixation radius!\n";
        }
    } while (fix_radius <= 0);
}

/**
 * Starts a calibration and confirms it with the user
 */
void Session::calibrate()
{
    string input;
    string wnd_title = "CalibrationWindow";
    HWND hCalWnd = NULL;

    // Prompt to calibrate eye tracker
    input = "n";
    do {
        if (input == "n") {
            cout << "Press Enter to calibrate the eye tracker.
                ";
            cin.get();
            while(tcp_send("CAL_START") == 1) {
                cout << "Error calibrating the eye tracker.
                    Ensure Gaze Tracker is running and press
                    Enter to try again.";
                cin.get();
            }
            while (hCalWnd == NULL) {
                hCalWnd = FindWindow(NULL, wnd_title.c_str());
            }
            SetForegroundWindow(hCalWnd);
        }
        else {
            cout << "Invalid input! Expected inputs: Y/n.\n";
        }
        cout << "Would you like to accept this calibration? (Y
            /n): ";
        getline(cin, input);
    } while (input != "Y");
    cout << "Calibration accepted.\n";
}

/**
 * Starts eye tracking on Ctrl+Alt+E hotkey press, opens file
    for writing and runs receive function in a thread.
 * Stops eye tracking on Ctrl+Alt+E hotkey press, waits for
    thread to complete and closes file.
 */
void Session::tracking()
{
```

```
MSG msg;
ofstream file;
time_t now;
HANDLE hThread;

// Wait for hotkey press to start eye tracking
cout << "Eye tracker ready. Press Ctrl+Alt+E in the
    desired application to begin eye tracking session.\n";
RegisterHotKey(NULL, 1, MOD_CONTROL | MOD_ALT, 0x45);
while(GetMessage(&msg, NULL, 0, 0) && msg.message !=
    WM_HOTKEY) {}

// Open file to write to
file.open((".\\Data\\"+trial+"\\"+subject+"_Data.txt").
    c_str());
file << "Trial:\t\t" << trial << "\n";
file << "Subject:\t" << subject << "\n\n";
file << "
    ----------------------------------------------------------------------------
    n";
time(&now);
file << "Eye tracking started " << ctime(&now);
file << "
    ----------------------------------------------------------------------------
    n";

// Start eye tracking
track = TRUE;
hThread = (HANDLE)_beginthread(udp_receive, 0, &file);
hCaptureDC = screenshot();

// Wait for hotkey press to stop eye tracking
cout << "Eye tracking session active. Press Ctrl+Alt+E to
    end eye tracking session.\n";
while(GetMessage(&msg, NULL, 0, 0) && msg.message !=
    WM_HOTKEY) {}

// Stop eye tracking
track = FALSE;
WaitForSingleObject(hThread, INFINITE);
CloseHandle(hThread);

// Close file
file << "
    ----------------------------------------------------------------------------
    n";
time(&now);
file << "Eye tracking ended " << ctime(&now);
file << "
    ----------------------------------------------------------------------------
    n";
file.close();
```

```
}

/**
 * Processes eye tracking data file, detects fixations in the
    coordinates and adds fixations to the fix_path vector
 *
 * @return Number of fixations identified
 */
unsigned int Session::process()
{
    ifstream file;
    string line;
    string temp;
    istringstream ss;
    unsigned long long t_start = 0;
    vector<Point> fix_points;
    Point current;

    file.open((".\\Data\\"+trial+"\\"+subject+"_Data.txt").
       c_str());

    // Process file data to determine fixations
    while(!file.eof()) {
        getline(file, line);
        if (line.find("STREAM_DATA")!=string::npos) {

            // Set string stream to current line and obtain
               current coordinates
            ss.clear();
            ss.str(line);
            ss >> temp >> current.t >> current.x >> current.y;

            // Initialise t_start
            if (t_start == 0) {
                t_start = current.t;
            }

            // I-DT fixation identification algorithm
            if (fix_points.size() == 0 || fix_points.back().t-
               fix_points.front().t < t_fix_min) {
                fix_points.push_back(Point(current.x, current.
                   y, current.t));
            }
            else if ((max.x-min.x)+(max.y-min.y) <= 2*
               fix_radius) {
                fix_points.push_back(Point(current.x, current.
                   y, current.t));
                get_limits(fix_points);
                if ((max.x-min.x)+(max.y-min.y) > 2*fix_radius
                   ) {
                    fix_points.pop_back();
                    add_fixation(fix_points);
```

```
                    fix_points.clear();
                    fix_points.push_back(Point(current.x,
                        current.y, current.t));
                }
            }
            else {
                fix_points.erase(fix_points.begin());
                fix_points.push_back(Point(current.x, current.
                    y, current.t));
            }
            get_limits(fix_points);
        }
    }
    file.close();

    // Add last fixation not identified by algorithm
    if (((max.x-min.x)+(max.y-min.y)) <= 2*fix_radius &&
        fix_points.back().t-fix_points.front().t >= t_fix_min)
        {
         add_fixation(fix_points);
    }

    // Calculate session time
    t_session = current.t-t_start;

    return fix_path.size();
}

/**
 * Get minimum and maximum coordinates of fixation points
    vector
 *
 * @param fix_points: Fixation points vector
 */
void Session::get_limits(vector<Point> fix_points)
{
    unsigned int i;

    min = Point(fix_points[1].x, fix_points[1].y);
    max = Point(fix_points[1].x, fix_points[1].y);

    for (i = 1; i < fix_points.size(); i++) {
        if (fix_points[i].x < min.x) {
            min.x = fix_points[i].x;
        }
        else if (fix_points[i].x > max.x) {
            max.x = fix_points[i].x;
        }
        if (fix_points[i].y < min.y) {
            min.y = fix_points[i].y;
        }
        else if (fix_points[i].y > max.y) {
```

```
            max.y = fix_points[i].y;
        }
    }
}

/**
 * Adds a fixation to the fix_path vector
 *
 * @param fix_points: Vector of the points within the
    potential fixation
 */
void Session::add_fixation(vector<Point> fix_points)
{
    unsigned int i;
    double x_sum = 0;
    double y_sum = 0;
    Point fix;

    // Calculate centroid of fixation
    for (i = 0; i < fix_points.size(); i++) {
        x_sum += fix_points[i].x;
        y_sum += fix_points[i].y;
    }
    fix.x = floor(x_sum/fix_points.size());
    fix.y = floor(y_sum/fix_points.size());

    // Calculate fixation duration
    fix.t = fix_points.back().t-fix_points.front().t;

    // Determine if fixation is on screen
    if (fix.x >= 0 && fix.x < res_width && fix.y >= 0 && fix.y
        < res_height) {
        fix.on_screen = TRUE;
    }
    else {
        fix.on_screen = FALSE;
    }

    fix_path.push_back(fix);
}

/**
  * Session constructor
  * Calls class functions
  * Declares Metrics, GazePlot and HeatMap objects if
     fixations are identified
  */
Session::Session()
{
    get_details();
    cout << "\n";
    calibrate();
```

```
    cout << "\n";
    tracking();

    if (process() > 0) {

        Metrics metrics(this);

        GazePlot gazeplot(this);

        HeatMap fix_count_heatmap(this);
        fix_count_heatmap.fix_count(this);

        HeatMap t_fix_heatmap(this);
        t_fix_heatmap.t_fix(this);

        HeatMap t_fix_mean_heatmap(this);
        t_fix_mean_heatmap.t_fix_mean(this);

        cout << "Eye tracking session complete. See the " <<
            trial << " directory for the results.\n";
    }
    else {
        cout << "No fixations were detected! Ensure the eye
            tracker is plugged in and calibrated in Gaze
            Tracker.\n";
    }
}

/**
 * Session destructor
 * Cleans up
 */
Session::~Session()
{
    DeleteObject(hCaptureDC);
}
```

D.9 Metrics.cpp

```
/**
 * @file Metrics.cpp
 * @author Thomas Bradford (thomas.bradford91@gmail.com)
 *
 * Metrics class implementation
 */

#include <cmath>
#include <ctime>
#include <fstream>
#include <iomanip>
```

```
#include "global.hpp"
#include "Session.hpp"
#include "Metrics.hpp"

using namespace std;

/**
 * Writes session details and metrics to file
 *
 * @param trial: Session trial name
 * @param subject: Session subject name
 * @param fix_radius: Fixation radius
 */
void Metrics::write_results(string trial, string subject, int
    fix_radius)
{
    ofstream file;
    time_t now;

    time(&now);
    tm *ltm = localtime(&now);

    // Write results to file
    file.open((".\\Data\\"+trial+"\\"+subject+"_Results.txt").
        c_str());
    file << fixed;
    file << "
        ------------------------------------------------------------
        n";
    file << "                              Gaze Analyser
        Results\n";
    file << "
        ------------------------------------------------------------
        n";
    file << "Session Details\n";
    file << "
        ------------------------------------------------------------
        n";
    file << "Trial:\t\t\t" << trial << "\n";
    file << "Subject:\t\t" << subject << "\n";
    file << "Date:\t\t\t" << ltm->tm_mday << "/" << setfill('0
        ') << setw(2) << ltm->tm_mon+1 << "/" << ltm->tm_year
        +1900 << "\n";
    file << "Screen resolution:\t" << res_width << "x" <<
        res_height << "\n";
    file << "Fixation radius:\t" << fix_radius << " px\n";
    file << "Session duration:\t" << t_session_mins << ":" <<
        setfill('0') << setw(2) << t_session_secs << " min\n";
    file << "\n";
    file << "
        ------------------------------------------------------------
        n";
```

```
    file << "Metrics\n";
    file << "
       ------------------------------------------------------------------------
       n";
    file << "On-screen fixations:\t\t\t" << on << " (" <<
       setprecision(1) << 100*(double)on/(on+off) << "%)\n";
    file << "Off-screen fixations:\t\t\t" << off << " (" <<
       setprecision(1) << 100*(double)off/(on+off) << "%)\n";
    file << "Overall fixation rate:\t\t\t" << setprecision(1)
       << fix_rate << " fixations/s\n";
    file << "Mean on-screen fixation duration:\t" <<
       setprecision(0) << t_fix_mean << " ms\n";
    file << "Mean on-screen saccade length:\t\t" <<
       setprecision(0);
    if (on == 1) {
        file << "0 pixels\n";
    }
    else {
        file << saccade_length_mean << " px\n";
    }
    file << "\n";
    file << "
       ------------------------------------------------------------------------
       n";
    file.close();
}

/**
 * Metrics constructor
 * Calculates eye tracking metrics for session
 *
 * @param session: Session class pointer
 */
Metrics::Metrics(Session *session)
{
    unsigned int i;
    unsigned int t_sum = 0;
    double saccade_length_sum = 0;
    int saccade_count = 0;

    on = 0;
    off = 0;

    t_session_mins = session->t_session/1000/60;
    t_session_secs = session->t_session/1000%60;

    // Calculate metrics
    for (i = 0; i < session->fix_path.size(); i++) {
        if (session->fix_path[i].on_screen) {
            on++;
            t_sum += session->fix_path[i].t;
```

```
            if (i < session->fix_path.size()-1 && session->
                fix_path[i+1].on_screen) {
                saccade_length_sum += (sqrt(pow(abs(session->
                    fix_path[i].x-session->fix_path[i+1].x), 2)
                    +pow(abs(session->fix_path[i].y-session->
                    fix_path[i+1].y), 2)));
                saccade_count++;
            }
        }
        else if (!session->fix_path[i].on_screen) {
            off++;
        }
    }

    fix_rate = 1000*session->fix_path.size()/(double)session->
        t_session;
    t_fix_mean = floor(t_sum/(double)on+0.5);
    saccade_length_mean = floor(saccade_length_sum/(double)
        saccade_count+0.5);

    write_results(session->trial, session->subject, session->
        fix_radius);
}
```

D.10 save_bitmap.cpp

```
/**
 * @file save_bitmap.cpp
 * @author Thomas Bradford (thomas.bradford91@gmail.com)
 *
 * Saves a bitmap to a file
 *
 * @param hDC: handle to the device context of the bitmap
 * @param hBmp: handle to the bitmap
 * @param filename: desired filename with extension
 */

#include <string>
#include <windows.h>

#include "global.hpp"

using namespace std;

void save_bitmap(HDC hDC, HBITMAP hBmp, string filename)
{
    BITMAP bmp;
    unsigned int bmp_size;
    char *bmp_bits;
    BITMAPFILEHEADER bmfh;
    BITMAPINFO bmi;
```

```
    HANDLE hFile;
    DWORD dwBytesWritten;

    // Initialise bitmap
    GetObject(hBmp, sizeof(BITMAP), &bmp);
    bmp_size = ((bmp.bmWidth*24+31)/32)*bmp.bmHeight*4;
    bmp_bits = new char[bmp_size];

    // Set bitmap file header struct
    bmfh.bfType = 0x4d42;
    bmfh.bfReserved1 = 0;
    bmfh.bfReserved2 = 0;
    bmfh.bfSize = sizeof(BITMAPFILEHEADER)+sizeof(
        BITMAPINFOHEADER)+bmp_size;
    bmfh.bfOffBits = (DWORD)sizeof(BITMAPFILEHEADER)+(DWORD)
        sizeof(BITMAPINFOHEADER);

    // Set bitmap info header struct
    bmi.bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
    bmi.bmiHeader.biWidth = bmp.bmWidth;
    bmi.bmiHeader.biHeight = bmp.bmHeight;
    bmi.bmiHeader.biPlanes = 1;
    bmi.bmiHeader.biBitCount = 24;
    bmi.bmiHeader.biCompression = BI_RGB;
    bmi.bmiHeader.biSizeImage = 0;
    bmi.bmiHeader.biXPelsPerMeter = 0;
    bmi.bmiHeader.biYPelsPerMeter = 0;
    bmi.bmiHeader.biClrUsed = 0;
    bmi.bmiHeader.biClrImportant = 0;

    // Get bitmap bits
    GetDIBits(hDC, hBmp, 0, bmp.bmHeight, bmp_bits, &bmi,
        DIB_RGB_COLORS);

    // Write file
    hFile = CreateFile (filename.c_str(), GENERIC_WRITE,
        FILE_SHARE_READ, NULL, CREATE_ALWAYS,
        FILE_ATTRIBUTE_NORMAL, NULL);
    WriteFile(hFile, &bmfh, sizeof(BITMAPFILEHEADER), &
        dwBytesWritten, NULL);
    WriteFile(hFile, &bmi, sizeof(BITMAPINFOHEADER), &
        dwBytesWritten, NULL);
    WriteFile(hFile, bmp_bits, bmp_size, &dwBytesWritten, NULL
        );
    CloseHandle(hFile);

    // Clean up
    delete[] bmp_bits;
}
```

D.11 GazePlot.cpp

```
/**
 * @file GazePlot.cpp
 * @author Thomas Bradford (thomas.bradford91@gmail.com)
 *
 * GazePlot class implementation
 */

#include <cmath>
#include <ctime>
#include <iomanip>
#include <sstream>
#include <string>
#include <vector>
#include <windows.h>

#include "global.hpp"
#include "GazePlot.hpp"

using namespace std;

void save_bitmap(HDC, HBITMAP, string);

/**
 * Gets upper limit of fixation durations
 *
 * @param fix_path: Fixation path vector
 */
void GazePlot::get_upper(vector<Point> fix_path)
{
    unsigned int i;

    // Start with upper equal to lower
    upper = t_fix_min;

    for (i = 0; i < fix_path.size(); i++) {
        if (fix_path[i].on_screen && fix_path[i].t > upper) {
            upper = fix_path[i].t;
        }
    }
}

/**
 * Draws scan path to device context
 *
 * @param fix_path: Fixation path vector
 * @param fix_radius: Fixation radius
 */
void GazePlot::draw(vector<Point> fix_path, int fix_radius)
{
```

```
    unsigned int i;
    unsigned int ellipse_radius;
    ostringstream ss;
    string text;
    RECT text_region;

    // Draw saccades as lines
    for (i = 0; i < fix_path.size(); i++) {
        if (i == 0 || !fix_path[i-1].on_screen) {
            MoveToEx(hGazePlotDC, fix_path[i].x, fix_path[i].y
                , NULL);
        }
        else if (fix_path[i].on_screen) {
            SelectObject(hGazePlotDC, hPurplePen);
            LineTo(hGazePlotDC, fix_path[i].x, fix_path[i].y);
        }
    }

    // Draw fixations as ellipses
    for (i = 0; i < fix_path.size(); i++) {
        if (fix_path[i].on_screen) {

            // First fixation
            if (i == 0) {
                SelectObject(hGazePlotDC, hGreenPen);
                SelectObject(hGazePlotDC, hLightGreenBrush);
            }

            // Last fixation
            else if (i == fix_path.size()-1) {
                SelectObject(hGazePlotDC, hRedPen);
                SelectObject(hGazePlotDC, hLightRedBrush);
            }

            // First on-screen fixation
            else if (!fix_path[i-1].on_screen) {
                SelectObject(hGazePlotDC, hBluePen);
                SelectObject(hGazePlotDC, hLightBlueBrush);
            }

            // Last on-screen fixation
            else if (!fix_path[i+1].on_screen) {
                SelectObject(hGazePlotDC, hOrangePen);
                SelectObject(hGazePlotDC, hLightOrangeBrush);
            }

            // Regular fixation
            else {
                SelectObject(hGazePlotDC, hPurplePen);
                SelectObject(hGazePlotDC, hLightPurpleBrush);
            }
```

```
            // Draw fixation ellipse
            ellipse_radius = get_radius(fix_path[i].t,
                fix_radius);
            Ellipse(hGazePlotDC, fix_path[i].x-ellipse_radius,
                fix_path[i].y-ellipse_radius, fix_path[i].x+
                ellipse_radius, fix_path[i].y+ellipse_radius);

            // Write fixation number
            text_region = {(long)fix_path[i].x, (long)fix_path
                [i].y, (long)fix_path[i].x, (long)fix_path[i].y
                };
            ss.str("");
            ss << i+1;
            text = ss.str();
            DrawText(hGazePlotDC, text.c_str(), -1, &
                text_region, DT_CENTER | DT_NOCLIP |
                DT_SINGLELINE | DT_VCENTER);
        }
    }
}

/**
 * Calculates the radius of the ellipse representing a
    fixation
 *
 * @param t: Fixation duration
 * @param fix_radius: Fixation radius
 * @return Ellipse radius in pixels
 */
int GazePlot::get_radius(int t, int fix_radius)
{
    double weight;

    if (upper == lower) {
        weight = 0;
    }
    else {
        weight = (double)(t-lower)/(upper-lower);
    }

    return floor(fix_radius*weight)+fix_radius;
}

/**
 * Draws footer to device context
 *
 * @param trial: Session trial name
 * @param subject: Session subject name
 * @param fix_radius: Fixation radius
 */
void GazePlot::draw_footer(string trial, string subject, int
    fix_radius)
```

```
{
    ostringstream ss;
    string text;
    time_t now;

    time(&now);
    tm *ltm = localtime(&now);

    // Draw footer background
    SelectObject(hGazePlotDC, hBlackPen);
    SelectObject(hGazePlotDC, hWhiteBrush);
    Rectangle(hGazePlotDC, 0, res_height+1, res_width,
        res_height+160);

    // Draw session details
    TextOut(hGazePlotDC, 20, res_height+10, "Gaze Plot", 9);
    TextOut(hGazePlotDC, 20, res_height+50, ("Trial: "+trial).
        c_str(), 7+trial.length());
    TextOut(hGazePlotDC, 20, res_height+75, ("Subject: "+
        subject).c_str(), 9+subject.length());
    ss << ltm->tm_mday << "/" << setfill('0') << setw(2) <<
        ltm->tm_mon+1 << "/" << ltm->tm_year+1900;
    text = ss.str();
    TextOut(hGazePlotDC, 20, res_height+100, ("Date: "+text).
        c_str(), 6+text.length());
    ss.str("");
    ss << fix_radius;
    text = ss.str();
    TextOut(hGazePlotDC, 20, res_height+125, ("Fixation radius
        : "+text+" px").c_str(), 20+text.length());

    // Draw legend
    TextOut(hGazePlotDC, res_width/2+20, res_height+10, "
        Legend", 6);
    SelectObject(hGazePlotDC, hGreenPen);
    SelectObject(hGazePlotDC, hLightGreenBrush);
    Ellipse(hGazePlotDC, res_width/2+20, res_height+50,
        res_width/2+60, res_height+90);
    TextOut(hGazePlotDC, res_width/2+70, res_height+60, "First
        fixation", 14);
    SelectObject(hGazePlotDC, hRedPen);
    SelectObject(hGazePlotDC, hLightRedBrush);
    Ellipse(hGazePlotDC, res_width/2+20, res_height+100,
        res_width/2+60, res_height+140);
    TextOut(hGazePlotDC, res_width/2+70, res_height+110, "Last
        fixation", 13);
    SelectObject(hGazePlotDC, hBluePen);
    SelectObject(hGazePlotDC, hLightBlueBrush);
    Ellipse(hGazePlotDC, res_width/2+260, res_height+50,
        res_width/2+300, res_height+90);
    TextOut(hGazePlotDC, res_width/2+310, res_height+60, "
        First on-screen fixation", 24);
```

```
    SelectObject(hGazePlotDC, hOrangePen);
    SelectObject(hGazePlotDC, hLightOrangeBrush);
    Ellipse(hGazePlotDC, res_width/2+260, res_height+100,
        res_width/2+300, res_height+140);
    TextOut(hGazePlotDC, res_width/2+310, res_height+110, "
        Last on-screen fixation", 23);
}

/**
 * GazePlot constructor
 * Initialises class
 * Calls class functions to draw gaze plot
 *
 * @param session: Session class pointer
 */
GazePlot::GazePlot(Session *session)
{
    // Initialise device context and bitmap
    hGazePlotDC = CreateCompatibleDC(session->hCaptureDC);
    hGazePlotBmp = CreateCompatibleBitmap(session->hCaptureDC,
        res_width, res_height+160);
    SelectObject(hGazePlotDC, hGazePlotBmp);
    BitBlt(hGazePlotDC, 0, 0, res_width, res_height, session->
        hCaptureDC, 0, 0, SRCCOPY);

    // Initialise graphics objects
    hFont = CreateFont(20, 0, 0, 0, FW_BOLD, FALSE, FALSE,
        FALSE, DEFAULT_CHARSET, OUT_DEFAULT_PRECIS,
        CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY, DEFAULT_PITCH |
        FF_DONTCARE, NULL);
    hPurplePen = CreatePen(PS_SOLID, 3, RGB(128, 0, 128));
    hGreenPen = CreatePen(PS_SOLID, 3, RGB(0, 128, 0));
    hBluePen = CreatePen(PS_SOLID, 3, RGB(0, 0, 128));
    hRedPen = CreatePen(PS_SOLID, 3, RGB(128, 0, 0));
    hOrangePen = CreatePen(PS_SOLID, 3, RGB(255 ,140, 0));
    hBlackPen = CreatePen(PS_SOLID, 3, RGB(0 ,0, 0));
    hLightPurpleBrush = CreateSolidBrush(RGB(221, 160, 221));
    hLightGreenBrush = CreateSolidBrush(RGB(144, 238, 144));
    hLightBlueBrush = CreateSolidBrush(RGB(173, 216, 230));
    hLightRedBrush = CreateSolidBrush(RGB(240, 128, 128));
    hLightOrangeBrush = CreateSolidBrush(RGB(255, 222, 173));
    hWhiteBrush = CreateSolidBrush(RGB(255, 255, 255));
    SelectObject(hGazePlotDC, hFont);
    SetBkMode(hGazePlotDC, TRANSPARENT);

    // Set lower to minimum fixation duration
    lower = t_fix_min;

    get_upper(session->fix_path);
    draw(session->fix_path, session->fix_radius);
    draw_footer(session->trial, session->subject, session->
        fix_radius);
```

```
    save_bitmap(hGazePlotDC, hGazePlotBmp, ".\\Data\\"+session
        ->trial+"\\"+session->subject+"_GazePlot.bmp");
}

/**
 * GazePlot Destructor
 * Cleans up
 */
GazePlot::~GazePlot()
{
    DeleteObject(hFont);
    DeleteObject(hPurplePen);
    DeleteObject(hGreenPen);
    DeleteObject(hBluePen);
    DeleteObject(hRedPen);
    DeleteObject(hOrangePen);
    DeleteObject(hBlackPen);
    DeleteObject(hLightPurpleBrush);
    DeleteObject(hLightGreenBrush);
    DeleteObject(hLightBlueBrush);
    DeleteObject(hLightRedBrush);
    DeleteObject(hLightOrangeBrush);
    DeleteObject(hWhiteBrush);
    DeleteObject(hGazePlotBmp);
    DeleteDC(hGazePlotDC);
}
```

D.12 HeatMap.cpp

```
/**
 * @file HeatMap.cpp
 * @author Thomas Bradford (thomas.bradford91@gmail.com)
 *
 * HeatMap class implementation
 */

#include <ctime>
#include <iomanip>
#include <sstream>
#include <string>
#include <vector>
#include <gdiplus.h>
#include <windows.h>

#include "global.hpp"
#include "HeatMap.hpp"

#define TRANSPARENCY 128

using namespace Gdiplus;
using namespace std;
```

```
void save_bitmap(HDC, HBITMAP, string);

/**
 * Gets upper limit of grid array
 */
void HeatMap::get_upper()
{
    unsigned int i;
    unsigned int j;

    upper = lower;

    for (j = 0; j < grid_height; j++) {
        for (i = 0; i < grid_width; i++) {
            if (grid[i][j] > upper) {
                upper = grid[i][j];
            }
        }
    }
}

/**
 * Draws heat map to device context
 */
void HeatMap::draw()
{
    unsigned int i;
    unsigned int j;
    unsigned int img_x;
    unsigned int img_y;
    double weight;
    unsigned int R;
    unsigned int G;
    unsigned int B;

    Graphics screen(hHeatMapDC);

    for (j = 0; j < grid_height; j++) {
        for (i = 0; i < grid_width; i++) {
            if (grid[i][j] > 0) {

                // Calculate normalised weight of current cell
                if (upper == lower) {
                    weight = 0;
                }
                else {
                    weight = (double)(grid[i][j]-lower)/(upper
                       -lower);
                }
```

```
                    // Colour is blue with increasing green as
                       weight approaches 0.25
                    if (weight <= 0.25) {
                        R = 0;
                        G = floor(255*weight/0.25);
                        B = 255;
                    }

                    // Colour is green with decreasing blue as
                       weight approaches 0.5
                    else if (weight <= 0.5) {
                        R = 0;
                        G = 255;
                        B = ceil(255*(1-weight/0.5));
                    }

                    // Colour is green with increasing red as
                       weight approaches 0.75
                    else if (weight <= 0.75) {
                        R = floor(255*weight/0.75);
                        G = 255;
                        B = 0;
                    }

                    // Colour is red with decreasing green as
                       weight approaches 1
                    else if (weight <= 1) {
                        R = 255;
                        G = ceil(255*(1-weight));
                        B = 0;
                    }

                    // Calculate corresponding pixel value of top
                       left corner of cell
                    img_x = res_width*i/grid_width;
                    img_y = res_height*j/grid_height;

                    // Set brush colour and fill the rectangle
                    SolidBrush colourBrush(Color(TRANSPARENCY, R,
                       G, B));
                    screen.FillRectangle(&colourBrush, img_x,
                       img_y, cell_size, cell_size);
                }
            }
        }
    }

/**
 * Draws footer to device context
 *
 * @param trial: Session trial name
 * @param subject: Session subject name
```

```
 * @param fix_radius: Fixation radius
 */
void HeatMap::draw_footer(string trial, string subject, int
    fix_radius)
{
    ostringstream ss;
    string text;
    time_t now;
    RECT text_region;

    time(&now);
    tm *ltm = localtime(&now);

    Graphics footer(hHeatMapDC);

    // Draw footer background
    Rectangle(hHeatMapDC, 0, res_height+1, res_width,
        res_height+160);

    // Draw session details
    TextOut(hHeatMapDC, 20, res_height+50, ("Trial: "+trial).
        c_str(), 7+trial.length());
    TextOut(hHeatMapDC, 20, res_height+75, ("Subject: "+
        subject).c_str(), 9+subject.length());
    ss << ltm->tm_mday << "/" << setfill('0') << setw(2) <<
        ltm->tm_mon+1 << "/" << ltm->tm_year+1900;
    text = ss.str();
    TextOut(hHeatMapDC, 20, res_height+100, ("Date: "+text).
        c_str(), 6+text.length());
    ss.str("");
    ss << fix_radius;
    text = ss.str();
    TextOut(hHeatMapDC, 20, res_height+125, ("Fixation radius:
        "+text+" px").c_str(), 20+text.length());

    // Draw legend
    TextOut(hHeatMapDC, res_width/2+20, res_height+10, "Legend
        ", 6);
    LinearGradientBrush linGrBrush1(Gdiplus::Point(res_width
        /2+20, 0), Gdiplus::Point(res_width/2+148, 0), Color
        (255, 0, 0, 255), Color(255, 0, 255, 255));
    LinearGradientBrush linGrBrush2(Gdiplus::Point(res_width
        /2+148, 0), Gdiplus::Point(res_width/2+276, 0), Color
        (255, 0, 255, 255), Color(255, 0, 255, 0));
    LinearGradientBrush linGrBrush3(Gdiplus::Point(res_width
        /2+276, 0), Gdiplus::Point(res_width/2+404, 0), Color
        (255, 0, 255, 0), Color(255, 255, 255, 0));
    LinearGradientBrush linGrBrush4(Gdiplus::Point(res_width
        /2+404, 0), Gdiplus::Point(res_width/2+532, 0), Color
        (255, 255, 255, 0), Color(255, 255, 0, 0));
    footer.FillRectangle(&linGrBrush1, res_width/2+20,
        res_height+50, 128, 70);
```

```
    footer.FillRectangle(&linGrBrush2, res_width/2+148,
        res_height+50, 128, 70);
    footer.FillRectangle(&linGrBrush3, res_width/2+276,
        res_height+50, 128, 70);
    footer.FillRectangle(&linGrBrush4, res_width/2+404,
        res_height+50, 128, 70);
    ss.str("");
    ss << lower;
    text = ss.str();
    text_region = {(int)res_width/2+20, (int)res_height+125, (
        int)res_width/2+20, (int)res_height+125};
    DrawText(hHeatMapDC, text.c_str(), -1, &text_region,
        DT_CENTER | DT_NOCLIP | DT_SINGLELINE);
    ss.str("");
    if (lower == upper) {
        ss << "Inf";
    }
    else {
        ss << upper;
    }
    text = ss.str();
    text_region = {(int)res_width/2+532, (int)res_height+125,
        (int)res_width/2+532, (int)res_height+125};
    DrawText(hHeatMapDC, text.c_str(), -1, &text_region,
        DT_CENTER | DT_NOCLIP | DT_SINGLELINE);
}

/**
 * HeatMap constructor
 * Initialises class
 *
 * @param session: Session class pointer
 */
HeatMap::HeatMap(Session *session)
{
    unsigned int i;
    GdiplusStartupInput gdiplusStartupInput;

    // Set grid size
    cell_size = 2*session->fix_radius;
    grid_width = ceil(res_width/cell_size);
    grid_height = ceil(res_height/cell_size);

    // Initialise grid
    grid = new unsigned int*[grid_width];
    memset(grid, 0, grid_width*sizeof(unsigned int));
    for (i = 0; i < grid_width; i++) {
        grid[i] = new unsigned int[grid_height];
        memset(grid[i], 0, grid_height*sizeof(unsigned int));
    }

    // Initialise device context and bitmap
```

```
    hHeatMapDC = CreateCompatibleDC(session->hCaptureDC);
    hHeatMapBmp = CreateCompatibleBitmap(session->hCaptureDC,
        res_width, res_height+160);
    SelectObject(hHeatMapDC, hHeatMapBmp);
    BitBlt(hHeatMapDC, 0, 0, res_width, res_height, session->
        hCaptureDC, 0, 0, SRCCOPY);

    // Initialise graphics objects
    hFont = CreateFont(20, 0, 0, 0, FW_BOLD, FALSE, FALSE,
        FALSE, DEFAULT_CHARSET, OUT_DEFAULT_PRECIS,
        CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY, DEFAULT_PITCH ||
        FF_DONTCARE, NULL);
    hBlackPen = CreatePen(PS_SOLID, 3, RGB(0 ,0, 0));
    hWhiteBrush = CreateSolidBrush(RGB(255, 255, 255));
    SelectObject(hHeatMapDC, hFont);
    SelectObject(hHeatMapDC, hBlackPen);
    SelectObject(hHeatMapDC, hWhiteBrush);

    // Initialise GDI+
    GdiplusStartup(&gdiplusToken, &gdiplusStartupInput, NULL);
}

/**
 * HeatMap destructor
 * Cleans up
 */
HeatMap::~HeatMap()
{
    unsigned int i;

    for (i = 0; i < grid_width; i++) {
        delete[] grid[i];
    }
    delete[] grid;

    DeleteObject(hBlackPen);
    DeleteObject(hWhiteBrush);
    DeleteObject(hHeatMapBmp);
    DeleteDC(hHeatMapDC);

    GdiplusShutdown(gdiplusToken);
}

/**
 * Fuction to draw a 'number of fixations' heat map
 *
 * @param session: Session class pointer
 */
void HeatMap::fix_count(Session *session)
{
    unsigned int i;
    unsigned int grid_x;
```

```
    unsigned int grid_y;
    RECT text_region;

    // Set grid values
    for (i = 0; i < session->fix_path.size(); i++) {
        if (session->fix_path[i].on_screen) {
            grid_x = floor(session->fix_path[i].x/cell_size);
            grid_y = floor(session->fix_path[i].y/cell_size);
            grid[grid_x][grid_y]++;
        }
    }

    // Set lower to the minimum number of fixations
    lower = 1;

    get_upper();
    draw();
    draw_footer(session->trial, session->subject, session->
        fix_radius);

    // Draw specific fixation count details to footer
    TextOut(hHeatMapDC, 20, res_height+10, "Heat Map (Number
        of Fixations)", 30);
    text_region = {(int)res_width/2+20, (int)res_height+125, (
        int)res_width/2+532, (int)res_height+125};
    DrawText(hHeatMapDC, "Number of fixations", -1, &
        text_region, DT_CENTER | DT_NOCLIP | DT_SINGLELINE);

    save_bitmap(hHeatMapDC, hHeatMapBmp, ".\\Data\\"+session->
        trial+"\\"+session->subject+"_HeatMap_Fixations.bmp");
}

/**
 * Fuction to draw a 'total fixation duration' heat map
 *
 * @param session: Session class pointer
 */
void HeatMap::t_fix(Session *session)
{
    unsigned int i;
    unsigned int grid_x;
    unsigned int grid_y;
    RECT text_region;

    // Set grid values
    for (i = 0; i < session->fix_path.size(); i++) {
        if (session->fix_path[i].on_screen) {
            grid_x = floor(session->fix_path[i].x/cell_size);
            grid_y = floor(session->fix_path[i].y/cell_size);
            grid[grid_x][grid_y] += session->fix_path[i].t;
        }
    }
```

```
    // Set lower to the minimum fixation duration
    lower = t_fix_min;

    get_upper();
    draw();
    draw_footer(session->trial, session->subject, session->
        fix_radius);

    // Draw specific total fixation duration details to footer
    TextOut(hHeatMapDC, 20, res_height+10, "Heat Map (Total
        Fixation Duration)", 34);
    text_region = {(int)res_width/2+20, (int)res_height+125, (
        int)res_width/2+532, (int)res_height+125};
    DrawText(hHeatMapDC, "Fixation duration (ms)", -1, &
        text_region, DT_CENTER | DT_NOCLIP | DT_SINGLELINE);

    save_bitmap(hHeatMapDC, hHeatMapBmp, ".\\Data\\"+session->
        trial+"\\"+session->subject+"_HeatMap_Duration.bmp");
}

/**
 * Fuction to draw a 'mean fixation duration' heat map
 *
 * @param session: Session class pointer
 */
void HeatMap::t_fix_mean(Session *session)
{
    unsigned int i;
    unsigned int j;
    unsigned int grid_x;
    unsigned int grid_y;
    RECT text_region;

    // Initialise two grids
    unsigned int fix_count_grid[grid_width][grid_height];
    memset(fix_count_grid, 0, grid_width*grid_height*sizeof(
        unsigned int));
    unsigned int t_fix_grid[grid_width][grid_height];
    memset(t_fix_grid, 0, grid_width*grid_height*sizeof(
        unsigned int));

    // Set grid values for number of fixations and total
        fixation duration
    for (i = 0; i < session->fix_path.size(); i++) {
        if (session->fix_path[i].on_screen) {
            grid_x = floor(session->fix_path[i].x/cell_size);
            grid_y = floor(session->fix_path[i].y/cell_size);
            fix_count_grid[grid_x][grid_y]++;
            t_fix_grid[grid_x][grid_y] += session->fix_path[i
                ].t;
        }
```

```
    }

    // Set grid values for mean fixation duration
    for (j = 0; j < grid_height; j++) {
        for (i = 0; i < grid_width; i++) {
            if (fix_count_grid[i][j] > 0) {
                grid[i][j] = floor(t_fix_grid[i][j]/
                    fix_count_grid[i][j]+0.5);
            }
        }
    }

    // Set lower to the minimum fixation duration
    lower = t_fix_min;

    get_upper();
    draw();
    draw_footer(session->trial, session->subject, session->
        fix_radius);

    // Draw specific mean fixation duration details to footer
    TextOut(hHeatMapDC, 20, res_height+10, "Heat Map (Mean
        Fixation Duration)", 33);
    text_region = {(int)res_width/2+20, (int)res_height+125, (
        int)res_width/2+532, (int)res_height+125};
    DrawText(hHeatMapDC, "Fixation duration (ms)", -1, &
        text_region, DT_CENTER | DT_NOCLIP | DT_SINGLELINE);

    save_bitmap(hHeatMapDC, hHeatMapBmp, ".\\Data\\"+session->
        trial+"\\"+session->subject+"_HeatMap_MeanDuration.bmp"
        );
}
```

Appendix E

Sample Output

```
------------------------------------------------------------------------
                        Gaze Analyser Results
------------------------------------------------------------------------
Session Details
------------------------------------------------------------------------
Trial:              Sample
Subject:            Thomas
Date:               29/10/2014
Screen resolution:  1366x768
Fixation radius:    20 px
Session duration:   0:30 min

------------------------------------------------------------------------
Metrics
------------------------------------------------------------------------
On-screen fixations:                54 (90.0%)
Off-screen fixations:               6 (10.0%)
Overall fixation rate:              1.9 fixations/s
Mean on-screen fixation duration:   485 ms
Mean on-screen saccade length:      139 px

------------------------------------------------------------------------
```

Figure E.1: Sample eye tracking results and metrics calculated using the low-cost eye tracking system.

No.	x	y	T	On-Screen	No.	x	y	T	On-Screen
1	122	174	1197	Yes	31	1232	284	649	Yes
2	168	166	171	Yes	32	1281	293	224	Yes
3	212	161	167	Yes	33	1322	298	294	Yes
4	411	159	585	Yes	34	1222	401	1270	Yes
5	673	149	1110	Yes	35	1252	419	251	Yes
6	129	266	292	Yes	36	1297	421	344	Yes
7	81	294	1463	Yes	37	1233	470	687	Yes
8	71	336	705	Yes	38	1280	481	301	Yes
9	85	367	536	Yes	39	1298	493	443	Yes
10	94	399	1125	Yes	40	1268	516	240	Yes
11	98	426	484	Yes	41	1247	544	408	Yes
12	297	281	504	Yes	42	1285	561	1080	Yes
13	330	285	534	Yes	43	1282	582	307	Yes
14	504	415	1007	Yes	44	1264	617	313	Yes
15	333	328	220	Yes	45	1282	651	735	Yes
16	614	328	199	Yes	46	1273	665	168	Yes
17	360	341	219	Yes	47	422	589	203	Yes
18	796	332	379	Yes	48	371	609	395	Yes
19	345	390	657	Yes	49	558	627	157	Yes
20	366	409	172	Yes	50	368	535	500	Yes
21	423	408	168	Yes	51	342	536	128	Yes
22	475	403	194	Yes	52	590	534	538	Yes
23	341	464	821	Yes	53	642	530	282	Yes
24	4636	-8400	203	No	54	677	524	841	Yes
25	2193	43	203	No	55	418	1333	243	No
26	2224	97	156	No	56	364	216	251	Yes
27	2235	142	579	No	57	318	200	252	Yes
28	2226	158	164	No	58	634	192	289	Yes
29	1101	374	153	Yes	59	697	192	666	Yes
30	1243	302	468	Yes	60	284	211	419	Yes

Table E.1: Fixations identified in the sample eye tracking session.

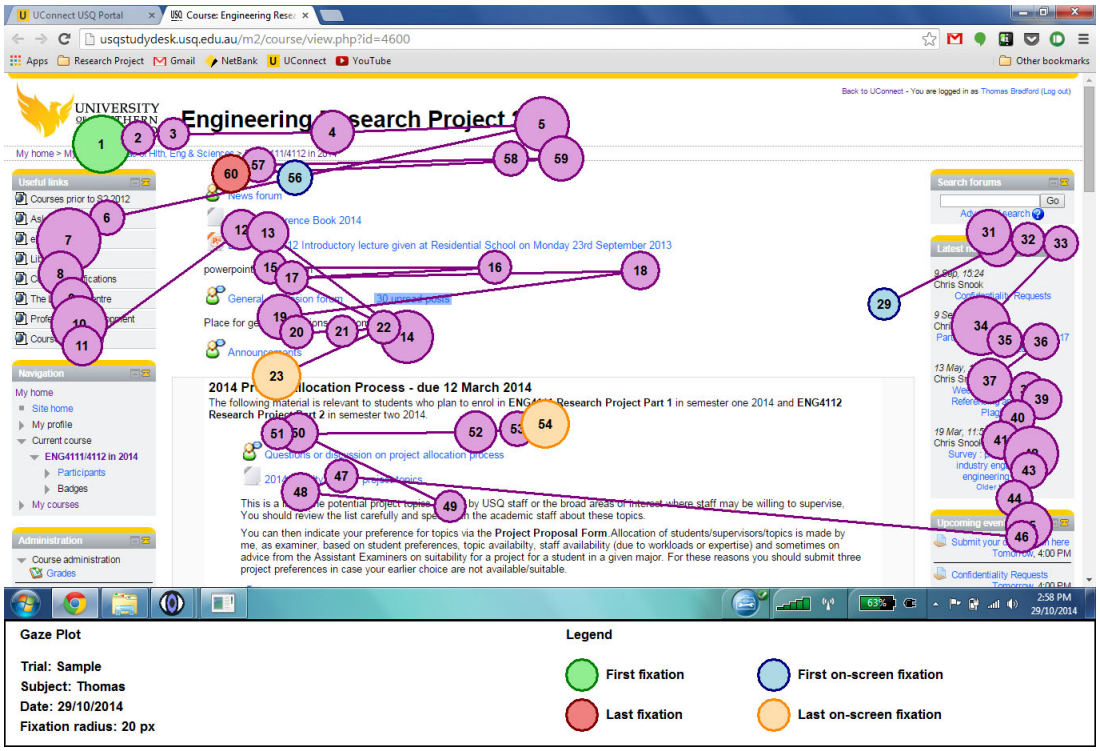


Figure E.2: Sample gaze plot generated using the low-cost eye tracking system.

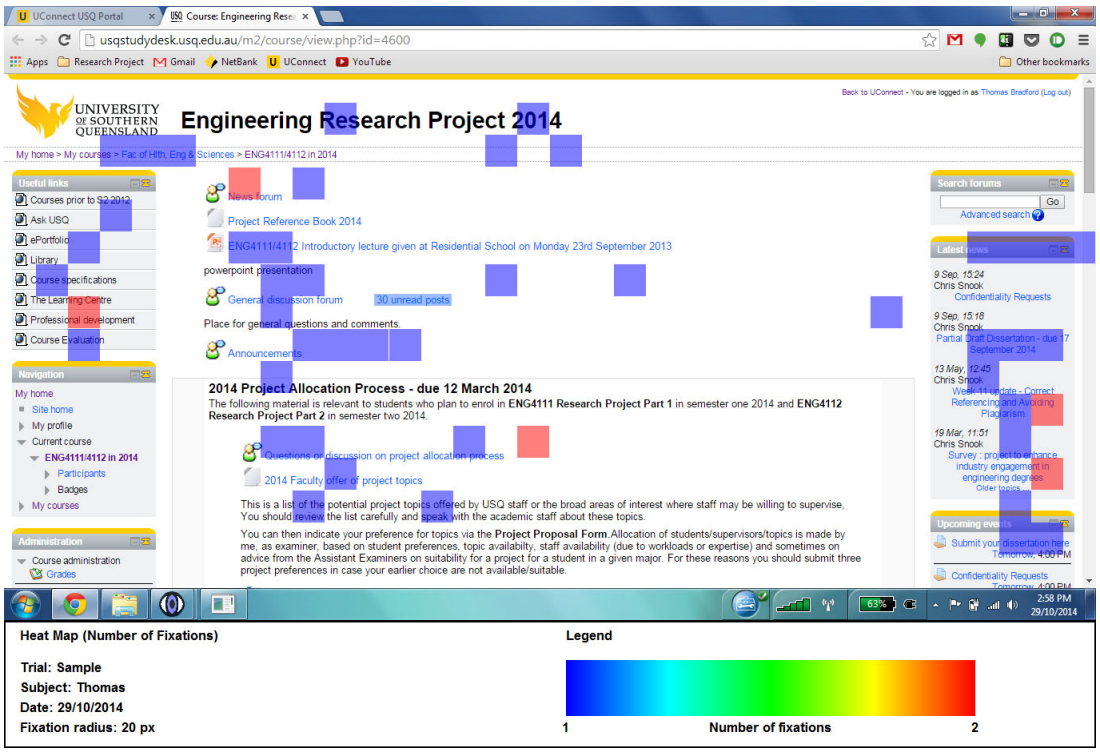


Figure E.3: Sample 'fixation count' heat map generated using the low-cost eye tracking system.

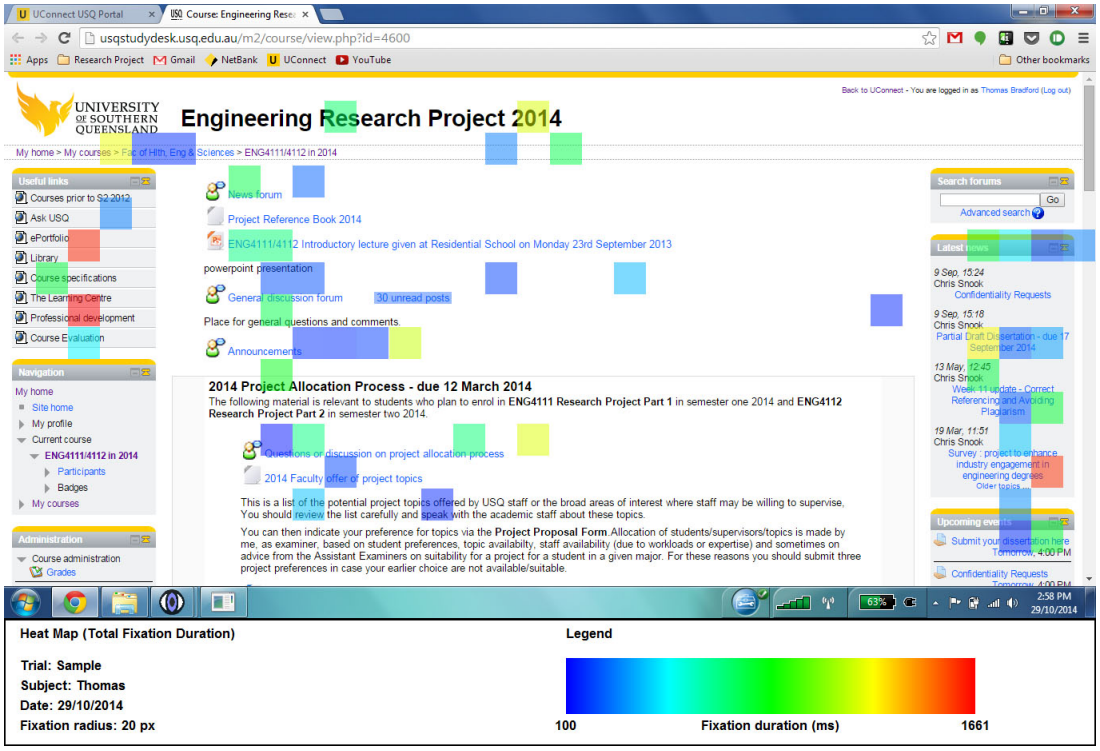


Figure E.4: Sample 'fixation duration' heat map generated using the low-cost eye tracking system.

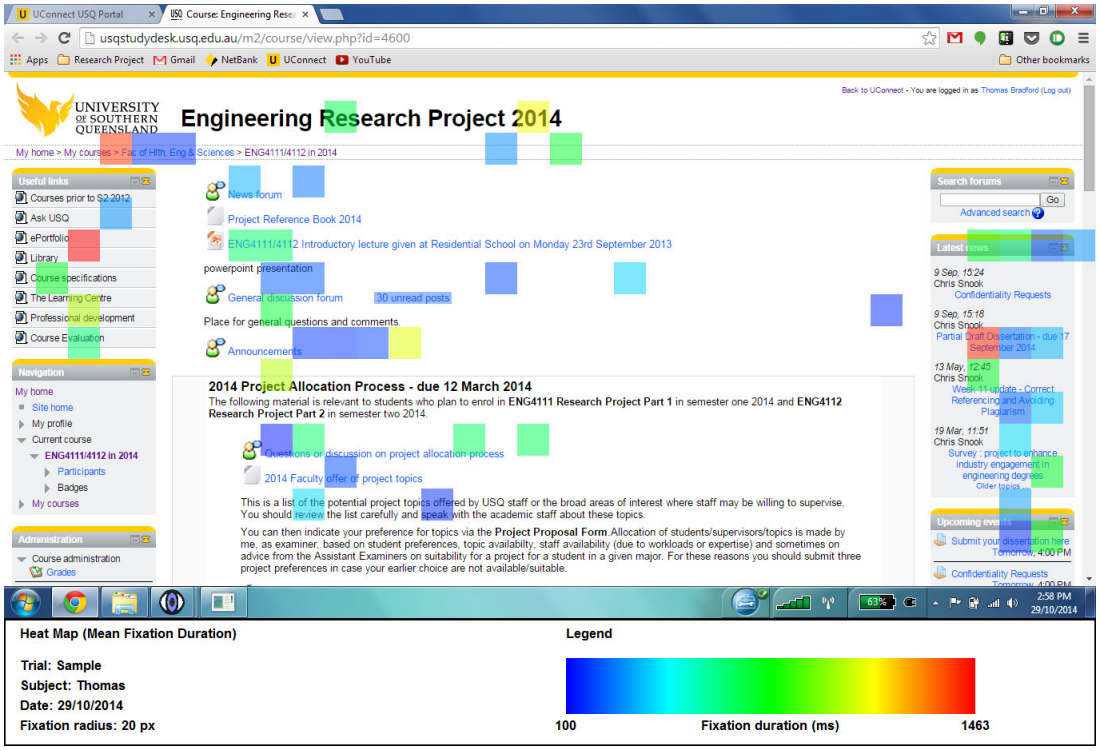


Figure E.5: Sample 'mean fixation duration' heat map generated using the low-cost eye tracking system.