CARDIFF UNIVERSITY

PRIFYSGOL CAERDYDD

Cardiff University
School of Computer Science and Informatics
BSc Computer Science (UFBSCMSA)

CM3203 - Final Report
Building a Prototype to Track Eye Gaze

Author: Gagandeep Malhotra
Supervisor: Yukun Lai
Moderator: Richard Booth
Date of Completion: 07/05/2023

Abstract

Eye contact is an important form of communication between people, which can convey where people’s attention lies and their emotions. These qualities struggle to extend to online interactions between people so eye gaze tracking can be used to retain this information as best as possible in a virtual environment. Recognising where people's attention is directed through eye gaze tracking can be useful for many applications, such as more natural human robot interaction, improving communication in virtual reality, and as a unique and intuitive input device.

This dissertation uses image processing (facial landmark detection) and machine learning techniques (Regression Convolutional Neural Network) to track eye gaze, using only a basic webcam. As well as displaying this information, the eye gaze direction is converted into a corresponding screen-coordinate using a linear regression model, which is then used to control the cursor’s position in real-time, as well as interact with a computer with left and right clicks, using only eye movements.

Furthermore, this dissertation further exemplifies that eye gaze tracking has the capabilities to be used as an intuitive tool to navigate a computer without the need of traditional hardware such as a mouse. Such a tool is extremely helpful for people with physical disabilities and those who otherwise struggle with traditional input devices. This problem of mapping a user’s eye gaze using a single webcam camera is a difficult challenge due to the many factors affecting this determination, including low camera/facial clarity, and a large variation in people’s pupil size and eye shape. This project will highlight the effectiveness of a machine learning approach to solve this problem and the choices made to determine the most accurate estimations for the eye gaze direction.

Acknowledgements

I would like to express my sincere gratitude to all those who have supported me throughout the completion of my Eye Gaze Tracking program.

I would like to thank my supervisor Yukun Lai for the project idea, guidance, and continual feedback throughout this process. Their insights have been instrumental in the creation of my work and helping me to understand and navigate the problem space.

I would also like to thank Cardiff University who have been incredibly helpful in providing access to the research materials I needed.

I would like to express my gratitude to my family and friends for their unwavering support and encouragement throughout the duration of this project and beyond. Their belief in me gave me the strength and motivation to persevere through any obstacles I faced.

Thank you all for your contributions and support, which have undoubtedly helped me greatly in my academic career.

Table of Contents

Abstract 2
Acknowledgements 3
Table of Figures 6
1. Introduction 7
1.1 Project Description 7
1.2 Project Aim 8
1.3 Project Audience 8
1.4 Project Scope 9
1.5 Approach used in Project 9
1.6 Assumptions of the Project 10
1.7 Summary of Important Outcomes 10
2. Background 12
2.1 Wider Context of Project 12
2.2 Problem Identified 14
2.3 Stakeholders 14
2.4 Theory of Problem Area 15
2.5 Constraints of Approach 15
2.6 Existing Solutions and Methods & Tools Used 16
3. Specification, Design and Implementation 20
3.1 Specification and Design 20
3.1.1 User Requirements 20
3.1.2 User Interface 20
3.1.3 Dynamic Behaviour 22
3.1.4 Data Flow 22
3.1.5 Algorithms 27
3.1.6 Architecture 27
3.2 Implementation 32
3.2.1 User Interface 32
3.2.2 Dynamic Behaviour 35
3.2.3 Algorithms 36
3.2.4 Problems 45
4. Results and Evaluation 49
5. Conclusions and Future Work 52

5.1 Conclusion 52
5.2 Future Work 53
6. Reflection 54
Appendices 55
References 63

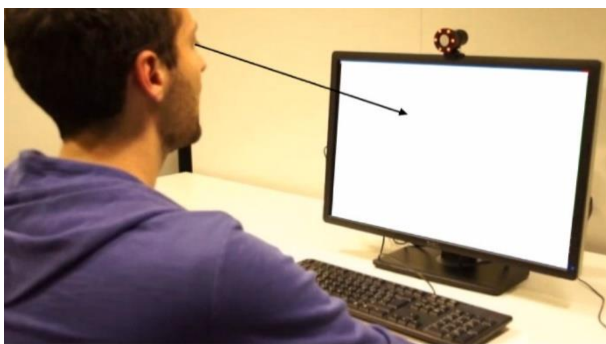
Table of Figures

Figure 1: Representation of Eye Gaze Tracking in Different Environments (Cazzato et al 2020)......... 7
Figure 2: Demonstration of SynthesEyes Database Images and their Collection Methods (Wood et al. 2020) 10
Figure 3: Two Images showing the iris of a user illuminated by infrared light, clearly showing the separation of pupil and iris in detail (Ole Baunbæk Jensen 2022)........ 13
Figure 4: (Left to right) Process for creating eye gaze estimation system with HCI capabilities (Gudi et al. 2020) 16
Figure 5: Graph of Screen Point Prediction Error (mm) against the number of calibration points for different implementations of converting gaze vector to screen coordinate (Gudi et al. 2020) 17
Figure 6: A graph showing how different input image crops for the machine learning model, affect the computation time and gaze-vector prediction error 18
Figure 7: Facial Landmark points using MediaPipe (Koushik and Chanda 2022) 19
Figure 8: A wireframe for the prototype's user interface 21
Figure 9: A wireframe for the prototype’s user interface if a new instruction is displayed........ 22
Figure 10: Pipeline for prototype (before ‘start’ is selected) 23
Figure 11: Pipeline for prototype (after 'start' is selected and continued from Figure 10)........ 23
Figure 12: Flowchart of Prototype........ 24
Figure 13: Flowchart of Calibration Sequence 25
Figure 14: Flowchart of Accessible Calibration Method........ 26
Figure 15: Menu GUI Shown on Program Launch........ 32
Figure 16: Instructions and Information Shown to User of Predicted Look Vector and Predicted Mouse Coordinate........ 33
Figure 17: Instruction Pop-Up of ‘Right Click’ at Top-Left of User's Device 33
Figure 18:Instruction Pop-Up of 'Mouse Disabled' at Top-Left of User's Device........ 33
Figure 19: Main Menu UI when Calibration Process is Ongoing........ 34
Figure 20: The Calibration Screen Which Displays 25 Calibration Points........ 34
Figure 21: Eye Gaze Vector Still Predicted When User is Located Far Away from Webcam........ 35
Figure 22: Eye Gaze Vector Still Predicted When User Has a Dark Background Area........ 35
Figure 23: Eye Gaze Vector Still Predicted When User's Face Is in Semi-Darkness........ 36
Figure 24: Eye Gaze Vector Still Predicted Even with One Eye Obscured........ 36
Figure 25: Detect Pupil Center using MediaPipe........ 37
Figure 26: Crop Eye Region 37
Figure 27: Two Functions that Plot Eye Gaze Vector on User's Webcam Feed........ 38
Figure 28: Training of Regression CNN Model to Predict Eye Gaze Vector from Image Input........ 40
Figure 29: Averages Both Predicted Gaze Vectors........ 40
Figure 30: Detects a Blink Based on Distance Between the Two Eyelids and Corners of the Eye........ 42
Figure 31: Linear Regression Model and R-Squared Score Calculation 42
Figure 32: Calibration Method and Storing Values Returned 44
Figure 33: Automatic Calibration Method Implementation........ 45
Figure 34: Conversion of SynthesEyes .pkl Files to .npz Files with Look Vectors Stored 46
Figure 35: Example of the Wollaston's effect (Wollaston 1824)........ 47

1. Introduction

1.1 Project Description

Eye contact is an important form of communication between humans, which can convey where people's attention lies as well as their emotions (Itier & Batty, 2009). These qualities struggle to extend to online interactions between people so eye gaze tracking can be used to retain this information as best as possible in a virtual environment. Eye gaze tracking estimates where a subject is looking using identifiable features such as face, eyes and pupils. These features can be detected by implementing infrared cameras or a head-mounted eye tracking device, and used to determine a user's eye gaze direction extremely accurately. However, these devices can be expensive, cumbersome to use, and seen as overly intrusive so have not been adopted by many. A method of eye gaze tracking without this barrier of entry, one which only utilises already existing hardware would be ideal for the mass adoption in forms of online communication. Therefore, this application only requires the user to have a normal webcam with their Microsoft Windows (2022) device, meaning this technology is accessible for a great deal more people. Eye gaze estimation is the process of predicting where a person's attention is directed towards using the current, size, shape, and direction of their facial and eye features (iris and pupil). There are many ways to refer to eye gaze tracking and there is an important distinction between this and just eye tracking. Gaze estimation concerns the reconstruction of the line of sight, and leverages different kinds of information (head pose, face geometry, and eye geometry) to achieve this estimation, eye tracking instead refers to analysing how the eye or its parts change their positions over time, which may be used to determine the eye gaze tracking estimation (Cazzato et al 2020).



(a) Constrained environments.

(b) Unconstrained environments.

Figure 1: Representation of Eye Gaze Tracking in Different Environments (Cazzato et al 2020).

In this dissertation, using Python a basic webcam is utilized to track eye gaze by employing facial landmark detection (image processing) and machine learning techniques (Regression Convolutional Neural Network (RCNN)). Additionally, a linear regression model is used to convert the eye gaze direction into a corresponding screen-coordinate, enabling real-time control of the cursor's position and interaction with a computer via left and right clicks through eye movements. The user can view their current and past predicted eye gaze vectors in the GUI for this program as well as a visual aid showing the live eye gaze direction from the user's eyes from the webcam feed. This GUI is intuitive and gives detailed information

and instructions on how to correctly calibrate and use the eye gaze tracking/mouse movement application.

Moreover, this dissertation demonstrates that eye gaze tracking has the potential to serve as an intuitive means of navigating a computer. Such an approach would be particularly advantageous for individuals with physical disabilities or those who experience difficulties using traditional input devices like a computer mouse. However, the task of accurately mapping a user's eye gaze using a single webcam camera poses significant challenges, including poor camera or facial clarity and considerable variations in pupil size and eye shape between users. This project will highlight the effectiveness of a machine learning approach to solve this problem and the choices made to determine the most accurate estimations for the eye gaze direction.

1.2 Project Aim

The primary motivation of this project is to build a robust prototype that can track eye gaze effectively with just a webcam using machine learning techniques. Furthermore, this project demonstrates that eye gaze tracking can be utilised in many ways, specifically showcasing the potential as an input device to navigate a computer. The detailed objectives within this overarching goal were to 'use machine learning methods such as Convolutional Neural Network (CNN) to determine the eye gaze of a user'. Another specific objective is that 'The eye gaze determination will only require a camera from the user which has a reasonable view of their face and eyes'. In addition to this, another objective is that the eye gaze tracking information will be utilised 'in the form of an application which allows the user to navigate their computer using their eye gaze information, where traditional inputs from the mouse would be instead be converted into inputs from the user's eyes'. This dissertation was designed from the beginning to be functional on devices using Microsoft Windows as an operating system as well as include a simple GUI interface for the user to interact with the program with. This dissertation builds beyond the ideas used to create applications with similar function, such as 'GazePointer' (2016), by utilising unique image processing techniques and a RCNN/Linear Regression model, as well as displaying wider array of detailed parameters to the user.

1.3 Project Audience

There are many different audiences which would find this dissertation useful both in terms of how the eye gaze tracking technology is implemented as well as the outcome of the project itself. For example, researches in the field of human computer interaction are demonstrated how eye gaze tracking can be used to benefit users who would want to navigate an electronic device more intuitively than other forms of input devices (Zander et al. 2010). These users include anyone from the general public, utilising this tool in online video meeting environments to show others where their attention is directed. This program is also beneficial to people who simply prefer using a hands-free method of human computer interaction for its comfortable usage. In addition to this, the implementation of moving the mouse cursor and navigating a device using eyes has tremendous potential in allowing disabled people a new form of hands-free communication (Solska and Kocejko 2022) and allows them to interact with people online. Eye gaze tracking technology from webcam can also be a great asset to

medical professionals who conduct consultations to detect conditions in patients such as autism, schizophrenia, and brain injury based on patterns detected in eye gaze movements (Shishido et al 2019).

1.4 Project Scope

The project specifically focuses on developing an image processing and machine learning-based eye gaze tracking system, using a basic webcam, with the ability of human computer interaction using the eye gaze values. Within the scope of this project is to develop a system that can accurately track eye gaze and convert it into corresponding screen-coordinates for controlling the cursor's position and interacting with a computer using only eye movements. Importantly, this dissertation demonstrates the potential of eye gaze tracking as an intuitive tool for navigating a computer without traditional hardware such as a mouse, particularly for people with physical disabilities or those who struggle with traditional input devices. In addition to this, this project acknowledges the challenges associated with eye gaze tracking using a single webcam camera and highlights the effectiveness of a machine learning approach to solve this problem. The calculation of the eye gaze coordinates and corresponding screen coordinate does not go beyond using image processing and machine learning techniques and does not cover using other methods such as geometric calculations or 3D modelling of eye region. Also, this dissertation does not develop a system for methods of improving communication in virtual reality or extensively evaluate the eyes as an input device against other forms of input devices.

1.5 Approach used in Project

The beginning of the creation of this project consisted of in-depth research about the subject area of eye-gaze tracking and how viable the brief was in being met under the time frame given, as well as if the intended approach was feasible. During the formation of the initial plan, a weekly plan was made that would detail the steps taken during week 1 to week 12 of the spring term in which the project was made. Along with informative weekly meetings with Professor Yukun Lai, the weekly plan was followed precisely to ensure that all milestones are met on time and to a high standard of quality. Once the machine learning methods to use were chosen based on past studies in this field, such as choosing to implement a linear regression model for optimal calibration efficiency (Gudi et al.2020), an evaluation of many databases with eye images and their corresponding look vectors was carried out. SynthesEyes (Wood et al. 2020) was chosen as the training database because it acknowledges the challenges involved in mapping a user's eye gaze using a single webcam camera, including factors such as low camera/facial clarity, and variation in people's pupil size and eye shape. The sheer number of eye images of different eye shapes and types (11,382) compounded with the precise associated coordinates of landmarks and three-dimensional eye gaze vector made this the ideal database to utilise. SynthesEyes also boasts a 90° of gaze variation which gives a large enough range to predict the gaze vector with

After the completion of the functionality of the program and once ensuring that all the individual elements were working correctly, the GUI of the project was created to call the functions made earlier. The user interface was made with a novice user in mind by including a single-menu interface which ensured that it was easy to navigate, with a plethora of

information and instructions displayed clearly to the user. From the beginning the project was planned to be written in python intended only for Microsoft Windows devices with a normal webcam attached.

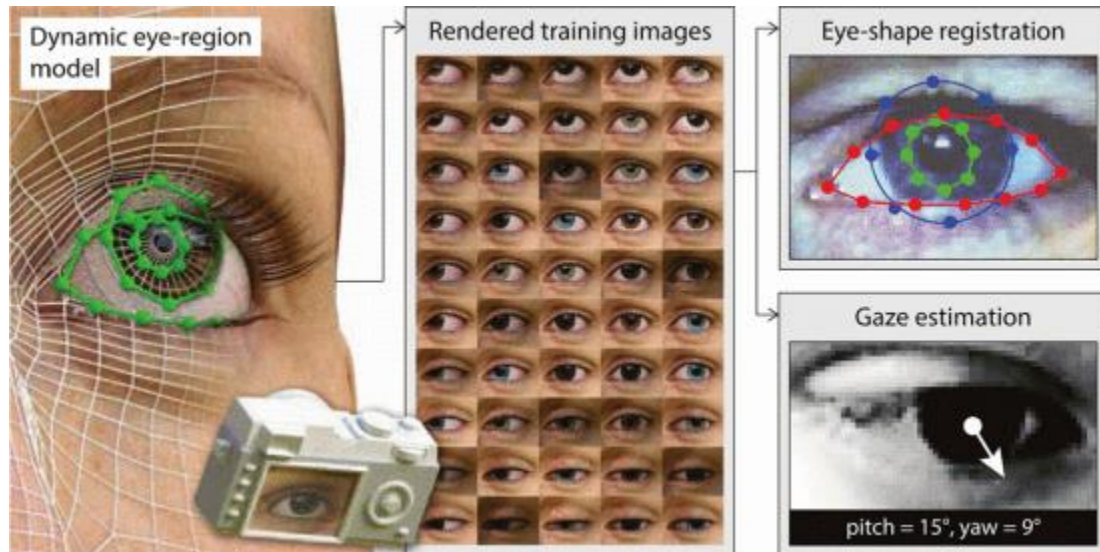


Figure 2: Demonstration of SynthesEyes Database Images and their Collection Methods (Wood et al. 2020)

1.6 Assumptions of the Project

A variety of assumptions are made both about the user and the system at the creation of this project. A salient assumption made is that the user has at least one functioning eye which has a clearly detectable pupil, as well as full motor control to indicate where they are looking at. Furthermore, this program assumes that the user has the hardware of a Microsoft Windows device, as well as an attached webcam, which is powerful enough to run this program which aims for 30 samples per seconds. Additionally, the entire program is written in English and relies on the user to follow calibration instructions correctly in order to have an accurate mouse cursor coordinate calculated. Moreover, the user will also need access to a keyboard to input the escape sequence of 'shift & F5' to exit the program if for any reason they cannot control the mouse correctly. Further assumptions are made at the beginning of the project that the two machine learning models that are used, will be accurate enough for the user to navigate their device with.

1.7 Summary of Important Outcomes

Several important outcomes are achieved through the creation of this dissertation for an eye gaze tracker with human computer interaction capabilities. Namely, clearly demonstrating that the unique machine learning model used is an effective method in which to track eye gaze from just a regular webcam. Also, showcasing the outstanding capabilities of using eyes as an input device with intuitive controls, as well as how to convert eye gaze into a mouse cursor coordinate and evaluating the different calibration techniques utilised in this project. The mapping of a user's eye gaze using a single webcam is a difficult challenge, but the

approach used in this project is effective in solving this problem and determining accurate estimations for the eye gaze direction.

2. Background

2.1 Wider Context of Project

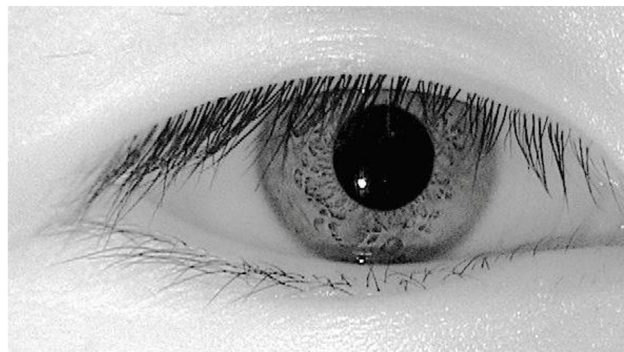
Eye contact plays an important role in social interaction and has the ability to capture and hold attention (Song et al. 2021). Through in-person interaction, eyes are commonly used to establish rapport between people, convey emotions, and effectively communicate with others. However, in online environments, it is found that the purposes of eye contact above are somewhat lost as they only occur with mutual, live eye contact and not in response to direct gaze pictures or when the observer believes that the live person cannot see them (Hietanen et al. 2020). Even in online video meetings, Ruth Wong (2020) found that relatedness (refers to an individual's desire and need to connect and create a bondage with people) was not present in online learning environments. Online live video environments, although they may show a broadcast of another person, it is not sufficient enough to replicate the communications and emotions expressed in-person. Several reasons for this include low quality of camera and therefore low facial/eye clarity, as well as the inability to see where someone's attention is directed through online videos. Since webcams, which display a video feed of the person on the other end, are typically placed above or alongside the user's device, it is very difficult to determine if someone is looking at their device or if their attention lies elsewhere in their own home environment. This is a problem leading to great inconsistencies in tone and approach when communicating online, making it problematic in creating social relationships through this medium. Even in a more interactive a three-dimensional virtual environment, such as those in virtual reality, VR cannot fully replicate the in-person effects that eye contact and eye gaze can express. As stated by Syrjämäki (2020), although artificial stimuli such as virtual avatars are helpful, but they cannot completely replace live stimuli when studying people's responses in live social interactions.

One method in which to bridge this gap is to use eye gaze tracking/eye gaze estimation to determine where your gaze is directed, and use this information to communicate to others where your attention is focused. This is more straight-forward to show in virtual reality where a three-dimensional vector may protrude from an avatar's eyes to convey where their attention lies and at what object they may be looking at. Clay (2019) finds that, you can easily match the eye gaze with the different objects in a virtual environment, making it much more effective in conveying and predicting the intention as well as the future actions of a user. Furthermore, present results suggest that virtual eye contact can evoke affective arousal responses, and thus implementing eye gaze information in VR could make virtual interactions more emotionally engaging (Syrjämäki et al. 2020). This proves that eye gaze tracking is an extremely useful tool to eliminate the shortcomings of online interactions that were stated earlier. In addition to this, virtual reality can also use gaze tracking effectively to implement foveated rendering which improves performance of a system by taking human eyes' inherent features and rendering different regions with different qualities without sacrificing perceived visual quality (Wang et al. 2023).

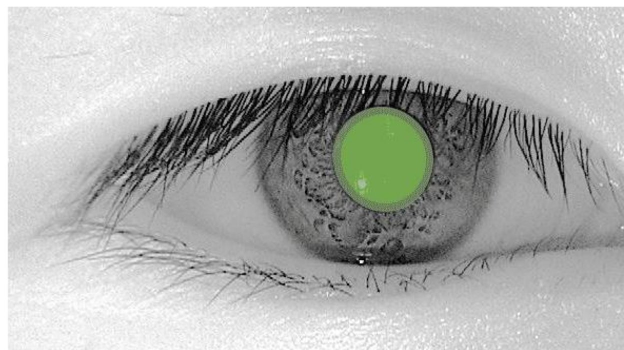
However, eye gaze tracking currently comes in many forms using a large variety of methods to determine where eye gaze lies. Typically, eye gaze trackers are either classified as intrusive, or non-intrusive, based on their level of physical discomfort for a user. Intrusive eye gaze trackers, although cumbersome for majority of users, are generally more accurate than their non-intrusive counterparts. Examples of intrusive gaze trackers, are

electrooculography (EOG), where sensors are placed on skin near the user's eyes to detect movements, and infrared oculography where custom glasses are used to detect infrared light reflected off the user's eyes (Modi et al., 2021).

Notwithstanding, infrared cameras are also a very common method to accurately predict eye gaze in a non-intrusive manner, ranging from products such as the Tobii Pro Nano (2022) to implementation in many current VR systems such as PSVR2 (2023) and Meta Quest Pro (2022). An example of how infrared cameras work is demonstrated in Figure 3, where the sensor typically interprets a greyscale image which easily separates the different features of an eye (pupil and iris), which will inform the device much more precisely of the location of these features to use for predictions. Normal webcams are unable to receive an image of the eye features as clear as the eye in Figure 3 because they only detect light in the visible spectrum. These work by using invisible near-infrared light and high-definition cameras to project light into the subject's eye, resulting in corneal reflections, which are recorded. Then advanced algorithms are applied, unique for each system, to convert this information into a gaze vector (Orduna-Hospital et al 2023). Many gaze tracking systems also take advantage of multiple cameras with different angles of the user's eye to get the most information to determine the current gaze vector. There also exists a recent and growing popularity in the use of normal webcams and machine learning algorithms to determine eye-gaze, which is what this project exemplifies.



Iris illuminated by infrared light.



Iris illuminated by infrared light, with iris highlighted (similar to how eye trackers follow eye movements).

Figure 3: Two Images showing the iris of a user illuminated by infrared light, clearly showing the separation of pupil and iris in detail (Ole Baunbæk Jensen 2022)

This project showcases one important use case of eye gaze estimation, which is as a method in which people can interact with their devices with. Many of these eye gaze estimation devices rely on specific hardware to function, whereas this project is focused on the software developed to make eye gaze estimation possible from a webcam, akin to applications such as 'GazePointer' (2016). Standard computer interfaces (e.g., mouse, keyboard), which require muscle movements during the Human-Computer Interaction (HCI), are mostly not suitable for users with severe motor disabilities (Cecilio et al. 2016). To overcome this, devices to help these people navigate devices disabilities are mechanical switches, proximity sensors, adapted joysticks, voice recognition solutions, head-trackers, and eye-trackers (Sumak et al. 2019). These devices are found to be very effective and results of the data analysis and comparison between non-disabled and disabled users showed that the disabled users performed very similarly to non-disabled users using an EPOC+ (headset with sensors to measure brain activity) solution, which demonstrates that it is possible for disabled people to use these devices with the right tools (Sumak et al. 2019).

2.2 Problem Identified

Disabled people can have problems with navigating devices using traditional input methods such as a mouse and keyboard. This inhibits them from feeling like part of a community and having access to opportunities which are otherwise available to abled-bodied people. To allow them to communicate with others and remain an active part of our society requires innovations in how people can interact with technology. All the intrusive and non-intrusive devices (listed in 2.1), those whose primary function is just to track eye gaze, have struggled greatly to see mass-adoption for a myriad of reasons. Some of the most prevalent reasons are because these specialised trackers are expensive and not scalable to mobile devices (Valliappan et al. 2020). The technology involved in the creation of infrared (IR) sensors as well as their high demand in consumer electronics and aerospace lead to these IR cameras being expensive. Therefore, the use of a normal webcam is because they are ubiquitous in current devices, including both front and back facing cameras on phones. This will make it easier for all people to have access to this software with no additional hardware purchases required, which is typical of gaze estimation mouse control programs.

Another problem identified is that using a laptop's keyboard and mouse results in awkward wrist placements while typing on a keyboard or using the touchpad, a lack of support for the arms, which in turn places stress on the upper back (Werth and Babski-Reeves 2012). This problem is exacerbated by the sharp increase of jobs which now require 'working from home', on a setup similar to that described above, after the COVID-19 outbreak (Bick et al. 2020). To address this problem, ergonomic input devices are effective, although they are still expensive like other solutions above. Therefore, an effective and cheap solution to solve this would be navigating the devices using an eye gaze tracking solution, reducing the risk for work related musculoskeletal disorders by avoiding the input devices that cause this harm.

2.3 Stakeholders

There are many likely stakeholders in the problem area of this dissertation, including researchers who are concerned with the topic of human-computer interaction, eye gaze

estimation, robot design, and machine learning. Also, users who have a physical disability or any motor impairment, as well as any user who would simply prefer to use this method of input device. A possible stakeholder could be people in the field of virtual reality, and the possibilities of using non-infrared cameras to reduce costs of headset production. Any other companies who design products for eye gaze estimation such as Tobii and GazePointer can also benefit from the research conducted and methods used in this project. Overall, this dissertation will be very relevant to a lot of researchers as well as users for many reasons concerned with using eye gaze tracking on a low-cost budget.

2.4 Theory of Problem Area

For the creation of this project, many underlying concepts and principles are used that guide the creation of the eye gaze tracking program. One of the most salient research areas is that of machine learning techniques and the resources on the benefits of regression CNN in the area of eye gaze tracking has greatly benefitted the effectiveness of the algorithm used in this program. The research involved in the purpose and effect of the different layers in a regression model has informed the choices used in this implementation. Computer vision techniques such as Haar Cascarde for object detection have been evaluated for implementation in this project and other methods of image processing like image conversions have been utilised heavily that the project relies on for working correctly. Research in the area of assistive technology and human computer interaction are cornerstones upon which the idea of the dissertation appeared, in solving the issues and deficiencies faced in these areas of study.

2.5 Constraints of Approach

Since the eye gaze tracking program purpose is to function with only the input from a low-cost webcam, the accuracy of the tracking, and consequently mouse movement, is dependent on many factors which depend on the hardware used as well as the user's actions. If the user is not in a clear position on the camera view, where both eyes are in view, the program's prediction will be inaccurate. The accuracy is also affected by a myriad of different reasons such as the camera quality not being clear enough or the user misinterpreting the calibration instructions and calibrating their eye gaze to the screen incorrectly. An additional difficult problem to overcome is if the intended device is positioned too far from the webcam, the linear regression algorithm to determine the mouse coordinate would struggle to be accurate. As this solution of using machine learning algorithms is chosen, if the user's hardware is not powerful enough to handle the model's image processing techniques and predictions, the program will become too slow to run and use effectively. Unfortunately, this is a problem as this dissertation's goal is to make eye-gaze tracking technology available on low-cost webcams and remain inexpensive, but would become inaccessible if the user does not have access to a powerful enough device to run this program. In conclusion, the main problem concerning this dissertation is many external factors negatively affecting both the performance and accuracy of this model.

2.6 Existing Solutions and Methods & Tools Used

There exist many solutions for eye gaze tracking (as referenced in 2.1), but for the purpose of human computer interaction using a regular webcam, not many solutions currently exist as it is a field that is only recently beginning to be heavily researched. In 2013, Ghani et al implemented an eye gaze tracking method for real-time mouse pointer control applications using a webcam. In this study, a feature extraction-based approach was used to detect the eyes and pupils, which is what is implemented into this program. Notwithstanding, many existing methods for webcam-based eye gaze tracking still lack robustness, meaning they will work for only a very clear view of the face. Similar programs to this project use a mixture of machine learning and geometric calculations for both the facial detection and point of gaze to screen coordinate conversion. For example, implementation of GazePointer uses a wide range of approaches to detect facial features, from Haar-features based approach for the face, and using Hough Circle Transform (HCT) for the pupils (Ghani et al. 2013). However, this implementation method is not as effective as using current machine learning models and datasets which have been developed in the time since this paper was published. The two research papers that greatly influenced the design process of this project are by Koushik Roy and Dibaloke Chanda (2022) as well as Amogh Gudi et al (2020).

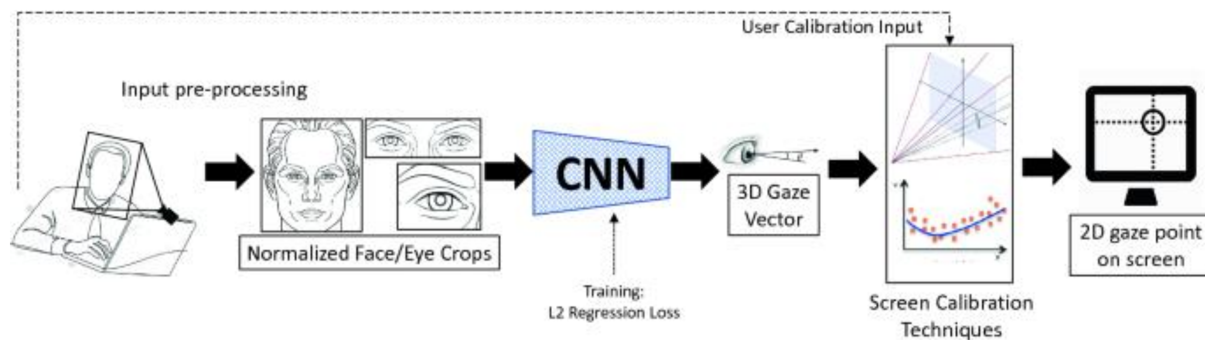


Figure 4: (Left to right) Process for creating eye gaze estimation system with HCI capabilities (Gudi et al. 2020)

An overview of the camera-to-screen gaze tracking pipeline. Images captured by the webcam are first pre-processed to create a normalized image of face and eyes which are used for training a convolutional neural network to predict the 3D gaze vector. With the cooperation of the user, the predicted gaze vectors can finally be projected on to the screen where he/she is looking using proposed screen calibration techniques. (Gudi et al. 2020)

The pipeline as illustrated in Figure 4 gives a straightforward design process for the system, without being overbearing in the description of the methods used which is useful as a reference tool to plan the project's methodology. The approach of pre-processing the image is extremely vital in ensuring that all inputs to the machine learning model are normalised and as compact in quality as can be whilst remaining accurate and the program running at a high-performance level. However, this solution does not consider moving the mouse cursor at the 2D gaze point on screen so the process must be adjusted taking this into account for this dissertation's goals to be met. Using a regression CNN model is also very apt for this

purpose, where an inputted image can be used to predict a continuous value such as a three-dimensional look vector, making this a great model to build upon in this dissertation. Once the gaze vector is predicted, using a calibration method (explored further in Figure 5) to map this to a two-dimensional point on the device screen for which the mouse will move to is a great concept of the processes involved in creating this application.

An important problem that must be overcome in the creation of this project is choosing the method in which to map to map eye gaze to screen-coordinate, as well as how to collect this data for calibration. The best method of calibration requires a large number of points from a wide variety of positions on the screen. However, as many implementations showed, collecting these calibration points were cumbersome to the user and eventually a plateau would be hit for the accuracy of the mapping, and the model would no longer improve with an increase in calibration points. Therefore, an optimum number of calibration points must be found that maximises accuracy whilst minimising time spent by the user. Of all the resources, Gudi's study was the most effective in evaluating the different methods of mapping an eye gaze vector to a screen coordinate. As is evident from Figure 5, the findings of this mapping process were tested with three implementations: a purely machine learning approach, a purely geometric approach, or a hybrid of the two implementations. From Figure 5, it is surmised that above around 10 calibration points the machine learning model and the hybrid model improves vastly, whilst the pure geometric model remains stagnant. The hybrid model then outperforms the ML model in terms of accuracy until after 40 calibration points where the ML becomes the best mapping method with the lowest screen point prediction error out of all the methods. This shows me that a large number of calibration points must be chosen, that is around 50 for this case, for machine learning methods to generally become superior in completing this task accurately.

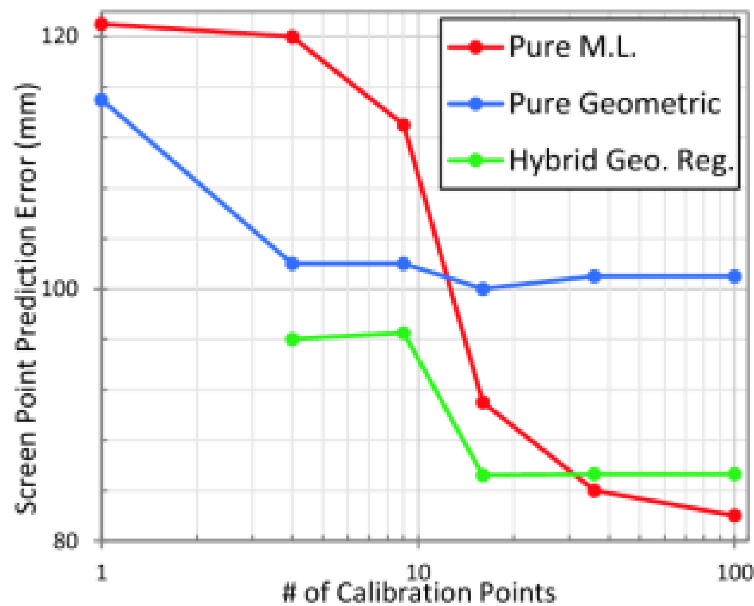


Figure 5: Graph of Screen Point Prediction Error (mm) against the number of calibration points for different implementations of converting gaze vector to screen coordinate (Gudi et al. 2020)

Learning curves of the calibration techniques showing that a purely geometric method performs better than ML method with a low number of calibration data. However, the pure geometric method does not improve further when more calibration data is given. The ML method improves greatly when calibration data becomes abundant.

Furthermore, the study by Gudi evaluated the efficiency of real-time webcam gaze tracking applications using a cropped image of a whole face, against two eyes, or just one eye. From the results in Figure 6, the crop of two eyes was used to have the lowest gaze-vector error prediction, followed by a facial crop, and lastly a single eye crop had the highest error values. Since the two eyes crop was found to take less than half the computation time of the full-face crop, this showed that cropping two eyes from the initial webcam image is the best method for maximising accuracy and minimising computation time. This particular study is very reliable in the data provided as it is stated that, 'in order to obtain a reliable error metric, we perform 5-fold cross-validation training'. This means that the model is trained on 4 of the folds and validated on the last fold. This process is repeated 5 times, with each fold serving as the validation set once. The performance of the model is then evaluated by averaging the results across the 5 validation sets to ensure a fair representation of the true value returned.

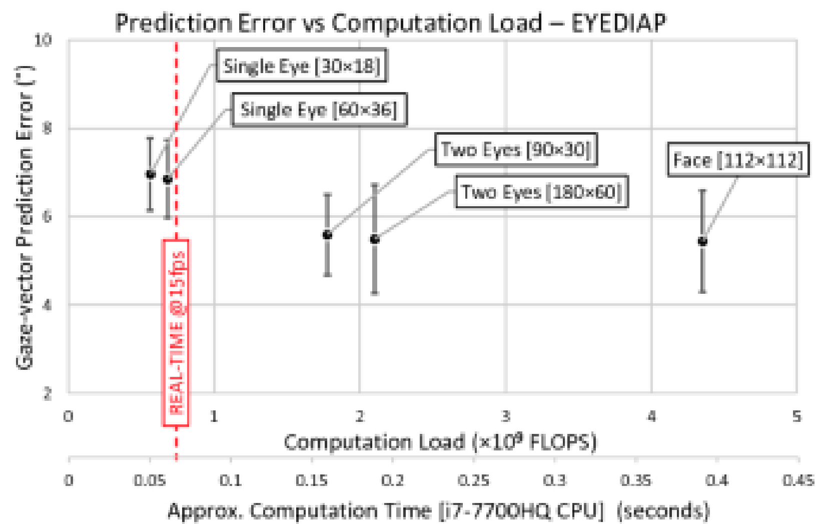


Figure 6: A graph showing how different input image crops for the machine learning model, affect the computation time and gaze-vector prediction error

This graph shows how the gaze-vector prediction error and average computational time is affected using a cropped image of a whole face, against two eyes, or just one eye. There are error bars included to accurately represent the range of values collected.

Koushik's study found that the use of Google's MediaPipe framework is a remarkable tool in creating applications that analyse media inputs, while remaining very high in performance. This framework also has many uses beyond this application, such as being excellent in performing human pose and figure detection (Singh et al. 2022). MediaPipe is used in Koushik's study to detect facial landmarks and crop out the eye region to be processed in the machine learning model. MediaPipe's high level of accuracy is attributed to its immense

training dataset with over 100,000 three-dimensional facial scans, as well as using methods such as weight decay to avoid overfitting. As seen in Figure 7, this framework is capable of detecting up to 468 unique landmarks on a person's face. This figure shows that these landmarks can be represented as green dots as well as an outline of the woman's face. MediaPipe is exceptionally accurate and maintains its accuracy in low-clarity situations which may obscure facial features of the user. MediaPipe is used extremely effectively by Koushik in conjunction with OpenCV (2021) to crop out the eye region for use in their CNN model to predict eye gaze.

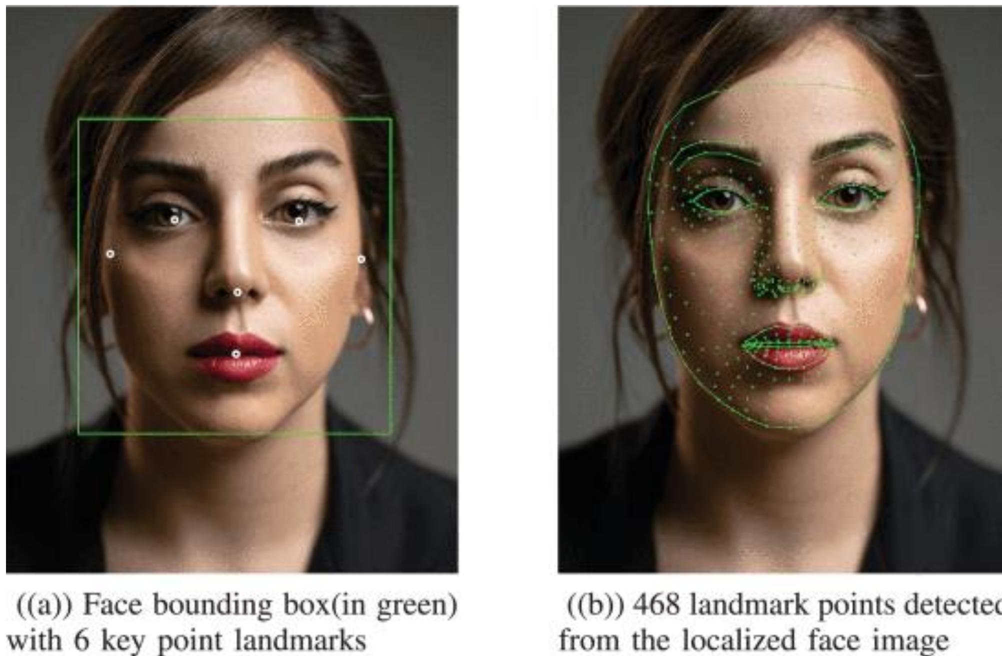


Figure 7: Facial Landmark points using MediaPipe (Koushik and Chanda 2022)

Google's MediaPipe can detect up to 468 unique landmarks on a person's face. This figure shows that these landmarks can be represented as green dots as well as an outline of the woman's face. MediaPipe is exceptionally accurate and can be useable even in dark or obscured environments

The aim for this project is to build a fully functional prototype that tracks eye gaze, using only a basic webcam. As well as displaying this information, the eye gaze direction is then used to control the cursor's position in real-time, as well as interact with a computer with left and right clicks, using only eye movements. In order to demonstrate the achievement of the stated aim, this prototype will get a live feed of the user's webcam and crop out both eye regions to be inputs for a machine learning model which will predict the look-vector. After a calibration process this information can be used to predict where the user is looking on the screen using another machine learning model. The user can then interact with the device using the predictions above in the robust prototype.

3. Specification, Design and Implementation

3.1 Specification and Design

3.1.1 User Requirements

In this program, there are several important requirements that would need to be met for this prototype to be effective for end users. The following requirements are chosen, ensuring that they are feasible, complete, and testable.

Functional requirements:

- The user interface must minimise the number of menus to make the application remain accessible

- The user must be able to view their real-time eye gaze as well as past eye gaze vectors

- The user must be able to see the webcam feed as well as cropped eye regions

- The user must be able to calibrate the eye gaze vector to device screen at their own pace

- The user must be able to exit the mouse movement process at any time with a key press

- The user must be able to choose the information that is being displayed on the user interface

- The user must be able to see if they have performed any actions with the mouse through an alert

These functional requirements are chosen because they fully encompass the scope of my project and satisfy all the criteria that needs to be met for the user to successfully use the program to fulfil its purpose.

Non-functional requirements:

- The user must have good performance, given the user has sufficient hardware

- The user interface should be a professional colour scheme

- The user should not experience any crashes/freezing during use

- The user should be able to navigate to every process through just one click

These non-functional requirements are chosen as they ensure that the application is easy for the user to navigate through, and find the option that they want. In addition to this, the prototype will perform well and consistently on the user's device to minimise unexpected processes/scenarios.

3.1.2 User Interface

The graphical user interface of this application is designed with the aim of being simple, intuitive, and informative for a user. The wireframe for the user interface is chosen to be one menu, to ensure that, since the purpose of this application is beneficial for many disabled

users, keeping the interface simple and minimising the number of menus is beneficial as they can navigate with just one click. The wireframe for the application's used interface is shown in Figure 8, where all processes and information can be accessed from one menu. At the top of the menu, it will be a feed of the webcam and the cropped section of each eye displayed to the user. This is so the user is aware of how they appear on the webcam, and if their eyes are being tracked correctly. Below this will be a checkbox for if the gaze-estimation should be drawn on the webcam feed or if the user prefers not to show this information. A checkbox is chosen which is clear and easy to understand for the user that they have a binary choice, and gives immediate visual feedback. Two boxes will then be displayed, the left box with instructions on how to work through the program, and the right box with information displayed concerning the predicted look vector as well as the screen-coordinate. At the very bottom of the UI will be two large buttons, one to 'calibrate' the application to the user's unique scenario, and another 'start' button that will become available once calibration is complete.

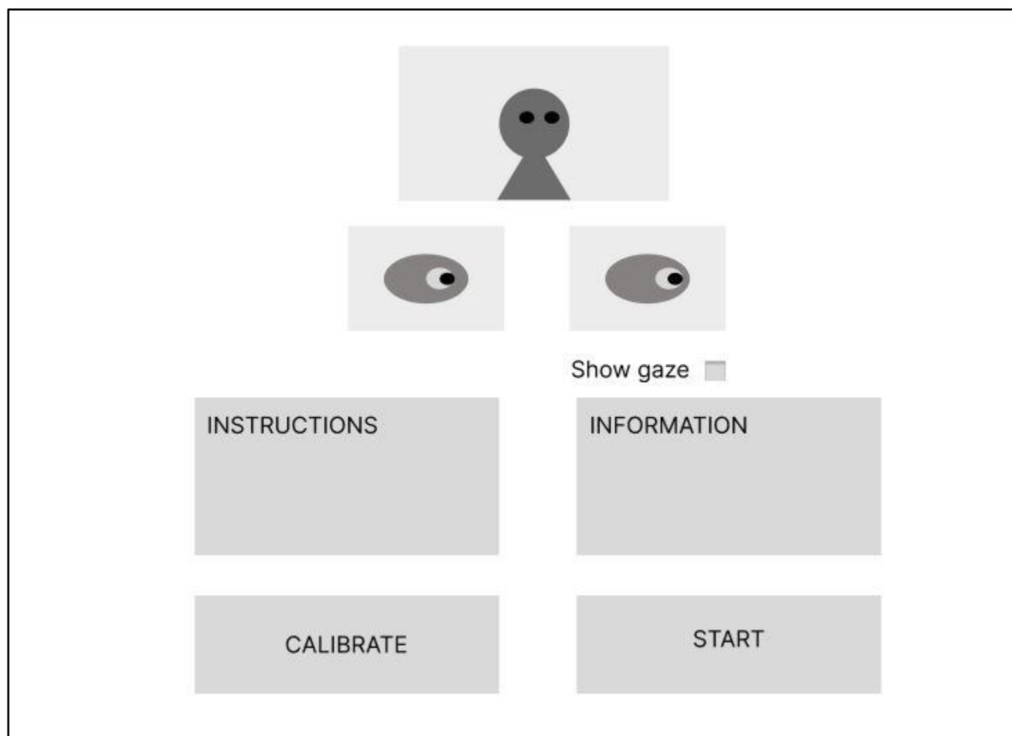


Figure 8: A wireframe for the prototype's user interface

The user will already be familiar with their mouse cursor, which will move on their device based on where the user is predicted to be currently looking. To alert the user of the current instruction that is shown, a pop-up box will appear above all other windows at the top-left of the screen to keep the user informed at all times about the processes that are being carried out, such as 'left click', 'right click', 'mouse disabled' as well as general instructions (Figure 9).

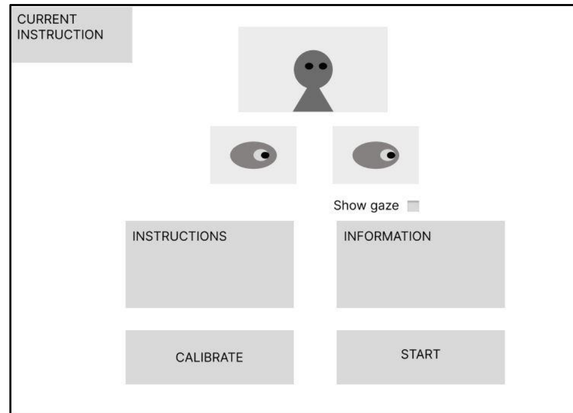


Figure 9: A wireframe for the prototype's user interface if a new instruction is displayed

3.1.3 Dynamic Behaviour

One of the most important elements of this application is to be efficient as to maintain its real-time tracking capabilities. By using a relatively lightweight framework such as MediaPipe this goal is much more possible when considering how the prototype is designed. Calibration is a vital feature for this prototype to accurately predict the screen-coordinate that the user is looking at an contributes greatly to how the system behaves to unique user circumstances. Furthermore, dynamic behaviour of this project entails that the screen-coordinate conversion be adaptable to the user's head position, although difficult as the user moves, introducing a recalibration feature into the design process will satisfy this condition of dynamic behaviour.

Through the use of regression CNN and MediaPipe, the program will remain robust under a large variety of user conditions, even if their face is partially obscured, predictions from the covered eye will still be made. In addition to this, the user interface of the program will adjust what information it shows based on the user's selected checkboxes. Another example of dynamic behaviour is that instructions will be displayed in the instruction box, as well as at the top-left of the screen to ensure the user is informed about the current state their prototype is in (Figure 9). Also, the 'start' button, as shown in Figures 8 & 9, will be greyed out and disabled until the 'calibration' is completed fully, so the user cannot start the program with a prediction model that is not suited to their current environment.

3.1.4 Data Flow

The pipeline for this prototype has taken inspiration from Figure 4, which succinctly deconstructed the main processes that would need to be carried out for this program to meet its requirements. As seen in Figure 10, the pipeline of this program has been broken down into relatively simple steps. First, the webcam feed will be received, which will then have the detected eye regions cropped out. Then these cropped images will be fed into the Regression CNN to predict a corresponding 3D gaze vector.

The program will perform all the tasks displayed in the pipeline of Figure 10 from the moment the program is launched, until it is closed. Once the calibration is completed by the user and the 'start' button is clicked, the pipeline will continue, as illustrated in Figure 11. This 'start' button will signify the start of the program predicting the "D screen-coordinate based on the user's calibration and the predict gaze vector. The final step will be to move the

mouse cursor to the predicted 2D screen-coordinate, which will then have the entire pipeline repeated until the user stops the program or exits the mouse movement process.

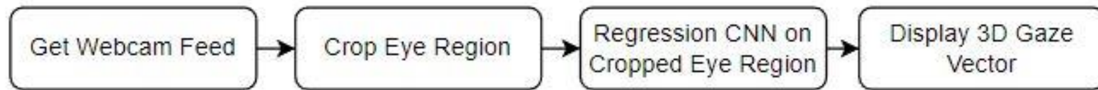


Figure 10: Pipeline for prototype (before 'start' is selected)

A flowchart of the main processes in this project is shown in Figure 12. This flowchart plan was created to help inform the design of the program and help identify any logical flaws in the order of processes. First, a webcam feed would be received, which would then be cropped into an eye region to feed into the Regression CNN model to predict a 3D gaze vector. Then, if the calibration is complete, the start button to begin moving the mouse will be displayed to the user. If this button is detected as being clicked, the program will then predict a corresponding 2D screen-coordinate to the gaze vector and then move the mouse cursor to this coordinate. Finally, the program will check to see if the user has selected to 'show the relevant information' in the GUI, and display this information if the checkbox has been selected. This is when the processes begin again until the user stops the program. The flow will immediately go to 'Is Show 'Gaze and Data' Selected?' in Figure 12 if any of the two conditional symbols before are evaluated as being false.

Another flowchart is demonstrated in Figure 13, which describes the processes that are carried out if the user begins the calibration of their system. When the 'calibrate' button is clicked, the calibration points will be displayed around the user's screen. If a user clicks on a calibration point, the current gaze vector as well as coordinate of the point itself is stored, and that particular calibration point will be removed. This will be repeated until no calibration points remain, and then the 'start' button to move the mouse cursor will be displayed. This is important because the user should not be able to begin moving the mouse cursor as it will be inaccurate without completing the calibration process beforehand.

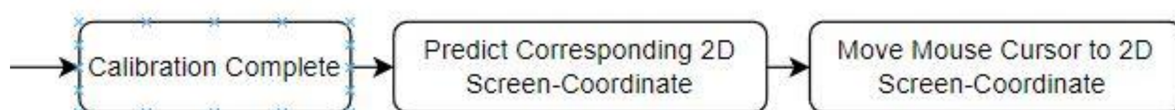


Figure 11: Pipeline for prototype (after 'start' is selected and continued from Figure 10)

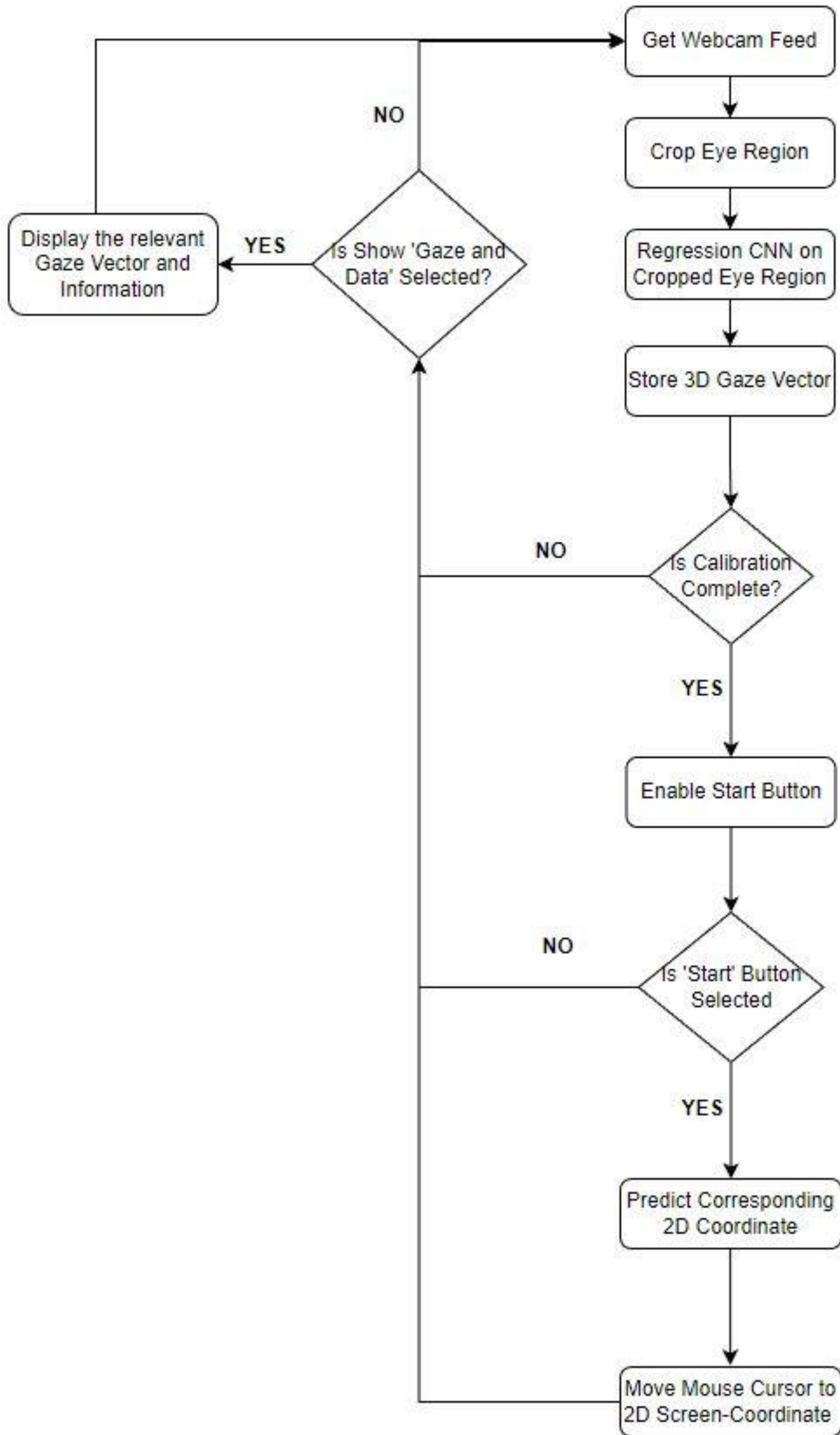


Figure 12: Flowchart of Prototype

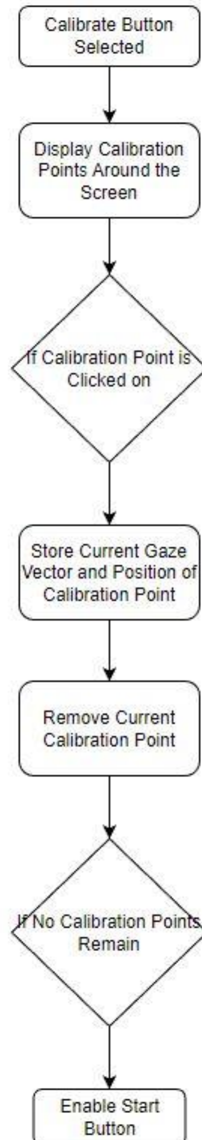


Figure 13: Flowchart of Calibration Sequence

Throughout the program there is a large variety of data types that are used to store important variables that are vital to the processes shown in Figures 10, 11, 12, 13. For example, the 3D gaze vector which is predicted by the machine learning model is aptly stored as a three-dimensional list, with a ‘x’, ‘y’, and ‘z’ value. The predicted 2D coordinate is stored as a two-dimensional list in the form of ‘x’ and ‘y’, which would correlate to a coordinate on the user’s screen.

In this dissertation, an additional experimental calibration feature is designed (Figure 14) which will allow the user to calibrate the gaze vector to screen coordinate process without the need of clicking on every individual calibration point. It is necessary, since these applications functionalities provide many benefits for disabled users, that the program should remain as accessible as possible by reducing the number of clicks required, with a traditional mouse, in the setup process. This experimental calibration setup is unique compared to all other

calibration implementations in webcam eye trackers such as Tobii and GazePointer. In this process, a square will appear at the top left of the screen and traverse the screen in an alternating pattern until all points have been accounted for. Every gaze prediction made during this time will be collected whilst the user is looking at the square. This will return a large variety of values, much larger than the tradition calibration method shown in Figure 13, to train the linear regression model on and determine accurate mouse cursor movements. The deliberate choice was made to allow the user to choose between both calibration methods depending on their preferences, allowing them to calibrate in the method that they are most comfortable with, because the experimental calibration will generally take longer than the normal calibration.

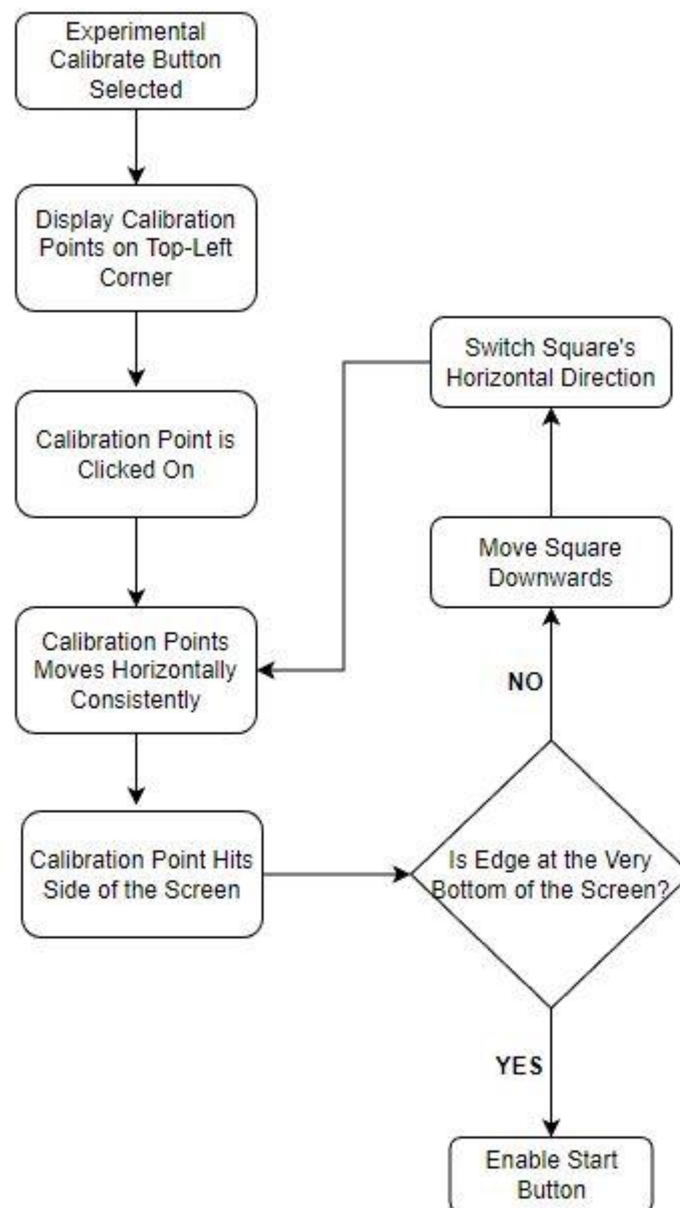


Figure 14: Flowchart of Accessible Calibration Method

3.1.5 Algorithms

Many algorithms are needed in the design of this program for the requirements to all be met sufficiently. Some of the main algorithms include the algorithm which detects and crops the eye regions, the regression convolutional neural network model, the linear regression model, mapping the eye gaze vector to the webcam, and showing the calibration points to the user.

The eye region crop is an important algorithm so an accurate image, like those in the SynthesEyes Database, can be fed into the regression CNN model which has its own unique implementation in this project that has been optimised to minimise loss. To map the eye gaze vector to the webcam as a simple visualisation for the user, this is done by converting the vector into pitch and yaw, using a magnitude value that accurately represents the z value of the eye gaze vector. All these algorithms are explored further in (3.2.3)

3.1.6 Architecture

All the code needed for the creation of this project has been partitioned into three separate modules, each of which have a distinct purpose. The three modules were cnn_regression, eye_tracker, and Tkinter_GUI.

First module of 'cnn_regression' defines the machine learning model that is used to predict the three-dimensional gaze vector from an image of the user's eye. This module trains the machine learning model using the SynthesEyes database (Wood et al. 2020). Below are all the modules and some of the more important variables required for the program to meet its functional and non-functional requirements as seen in 3.1.1.

Cnn_regression.py:

Function: **get_x_y**
- Variable: **df** (DataFrame)

Function: **get_grid_search**
- Variables: **X_train**, **y_train**, **X_val**, **y_val** (NumPy arrays), **model** (KerasRegressor), **batch_size** (list), **epochs** (list), **param_grid** (dictionary), **grid** (GridSearchCV), **grid_result** (GridSearchCV result object)

Function: **get_prediction**
- Variables: **npz_file_path** (string), **model** (Keras model), **npz_file** (NumPy file), **pic_data** (NumPy array), **prediction** (NumPy array)

Function: **compute_difference**
- Variables: **row** (DataFrame row), **x** (NumPy array), **pred** (NumPy array), **abs_diff** (NumPy array)

Function: **get_load_model**
- Variables: **df** (DataFrame), **input_pic** (NumPy array), **loaded_model** (Keras model), **test_predictions** (NumPy array), **test_preds_series** (pandas Series)

Function: **get_average_difference**
- Variables: **df** (DataFrame), **xyz_values** (list of NumPy arrays), **avg_xyz** (NumPy array)

Function: **create_model**

- Variables: **X_train**, **y_train**, **X_val**, **y_val** (NumPy arrays), **K** (backend module from Keras), **model** (Keras model), **cp** (ModelCheckpoint object)

Function: **get_loss**
- Variables: **model** (Keras model), **X_test**, **y_test** (NumPy arrays), **loss** (float)

Function: **get_all_vectors**
- Variables: **names_df** (DataFrame), **df** (DataFrame)

Function: **drop_columns**
- Variables: **df** (DataFrame), **column_list** (list)

Function: **load_npz_dataset**

Function: **create_dataframes**
- Variables: **df** (DataFrame), **train_size** (integer), **val_size** (integer), **train_df**, **val_df**, **test_df** (DataFrames)

Function: **load_pkl_dataset**

Function: **open_file**
- Variables: **file_name** (string), **f** (file object), **data** (pickled object), **image_file_path** (string), **file_look_vector_x**, **file_look_vector_y**, **file_look_vector_z** (floats)

The second module called 'eye_tracker' performs all the image processing required to get the eye region from the webcam feed using the landmarks that are detected. In addition to this, this module detects if the eye blinks and plots the visualisation of the three-dimensional gaze vector for the user to see.

<u>Eye_tracker.py:</u>

Function: **main**
- Variables: **averaged_look_vector** (list), **selected_display** (integer)

Function: **landmarksDetection**
- Variables: **img** (image), **results** (object), **draw** (boolean)

Function: **euclideanDistance**
- Variables: **point** (tuple), **point1** (tuple)

Function: **blinkRatio**
- Variables: **img** (NumPy array), **landmarks** (list), **right_indices** (list), **left_indices** (list)

Function: **get_face_center**
- Variables: **frame** (image), **results_mesh** (object)

Function: **get_eye_keypoints**
- Variables: **chosen_eye** (string), **frame** (image), **detection** (object), **mp_face_detection** (module)

Function: **get_eyes_dimensions**
- Variables: **left_eye** (tuple), **right_eye** (tuple)

Function: **get_eye_region**
- Variables: **eye** (tuple), **eye_width** (int), **eye_height** (integer), **frame** (Numpy array)

Function: **get_pupil_center**
- Variables: **chosen_iris** (list), **frame** (NumPy array), **results_mesh** (object)

Function: **get_look_vector_prediction**

• Variables: **eye_pic_data** (ndarray)
Function: **read_window**
• Variables: **eye_region** (NumPy array)
Function: **display_window**
• Variables: **eye_region** (NumPy Array), **window_name** (string)
Function: **read_image**
• Variables: **image** (NumPy array)
Function: **get_gaze_values**
• Variables: **pupil_center** (tuple), **look_vector** (list), **length** (int)
Function: **plot_gaze**
• Variables: **pupil_center** (tuple), **arrow_end** (tuple), **frame** (NumPy array)
Function: **exit_program**
• Variables: **cap** (object)
Function: read_frame
• Variables: **cap** (object), **face_mesh** (module)
Variables (defined initially outside of functions):
• **df** (DataFrame)
• **mp_face_detection** (module)
• **mp_drawing** (module)
• **face_detection** (object)
• **mp_face_mesh** (module)
• **LEFT_IRIS_ID** (list)
• **RIGHT_IRIS_ID** (list)
• **LEFT_EYE** (list)
• **RIGHT_EYE** (list)
• **FACE_CENTER_ID** (list)
• **cap** (object)
• **model** (object)

The final module is called 'TkinterGUI' and controls the user interface of the program and how the user is presented the information calculated by this and the other two modules. In addition to this, this module uses linear regression to convert the gaze vector to a predicted screen coordinate. The calibration methods are also defined in this module so the linear regression can be applied correctly.

TkinterGUI.py:

Initialised at the beginning of the module:

- **frame**: a frame object
- **self.look_vector**: a list of floats with three elements
- **left_eye_region**: a numpy array
- **right_eye_region**: a numpy array
- **reRatio**: a float
- **leRatio**: a float
- **self.screen_width**: an integer
- **self.screen_height**: an integer
- **self.look_vector_list**: an empty list

- **self.window_coordinate_list**: an empty list
- **self.gaze_history**: an empty list
- **self.spacing**: an integer
- **self.num_windows**: an integer
- **self.start**: None
- **self.move_mouse**: a Boolean

Widgets that are created for the user interface using Tkinter

- **self.title_label**: a label object
- **self.video_label**: a label object
- **video_labels_frame**: a frame object
- **self.left_eye_video_label**: a label object
- **self.right_eye_video_label**: a label object
- **container_frame**: a frame object
- **plot_checkbox_frame**: a frame object
- **self.selected_display**: an integer
- **self.plot_eyes_radiobutton**: a radiobutton object
- **self.plot_face_radiobutton**: a radiobutton object
- **self.plot_both_radiobutton**: a radiobutton object
- **info_checkbox_frame**: a frame object
- **self.look_vector_checked**: a Boolean
- **self.look_vector_checkbox**: a checkbutton object
- **self.mouse_coord_checked**: a Boolean
- **self.mouse_coord_checkbox**: a checkbutton object
- **frame1**: a frame object
- **instructions_frame**: a frame object
- **instructions_lbl**: a label object
- **self.instructions**: a text object
- **information_frame**: a frame object
- **information_lbl**: a label object
- **self.information**: a text object

Function that shows the user the webcam feed and performs linear regression

- **frame**: a numpy array
- **video_feed**: a numpy array
- **left_eye_image**: an Image object
- **left_eye_image_tk**: an ImageTk object
- **right_eye_image**: an Image object
- **right_eye_image_tk**: an ImageTk object
- **gaze**: a list of floats with three elements
- **window_coordinate**: a tuple with two integers
- **current_time**: a float
- **diff**: a float
- **face_location**: a tuple with four integers
- **is_look_vector_checked**: a Boolean

- **is_mouse_coord_checked**: a Boolean
- **predicted_coord**: a tuple with two integers

This program is split into these three distinct modules as they all serve a unique purpose, as well as to achieve modularity, so each of these important processes can be tested separately, independent from one another. In turn, this will lead to easier and more effective troubleshooting in the development of this dissertation's requirements.

3.2 Implementation

3.2.1 User Interface

For the implementation of the user interface (Figure 15), the program follows the original wireframe design that was created very closely. In particular, the UI was made to be simple, intuitive, and informative (see 3.1.2) for the user, which has been achieved through the use of clear headings and grouping similar information together, such as the eye gaze and webcam feed. The colour scheme chosen was a professional and sleek grey background with white boxes for the instructions and information to be displayed clearly. There is also a logo icon of an eye so the user is easily able to identify this app in taskbars (Appendix A4). This all was completed using the Tkinter library in Python because of its simplicity in use and large community that surround this library giving plenty of resources to learn solutions from. As seen in Appendix A1, the implementation of radio buttons is chosen for the user to select one of three mutually exclusive display methods for the calculated eye gaze vector. Appendix A1 also showcases how simple the Tkinter library is in creating the radio buttons by simply assigning each selection a unique value which will be checked in an 'if statement' to see which selection is current chosen. First default option of 'Eye Gaze' displays the vector protruding from the user's eyes (Figure 15), the second option (Appendix A2) displays a single gaze vector from the user's face, whilst Appendix A3 shows the option to 'display both' gaze vector. However, these changes are entirely visual and cosmetic, having no effect on the actual function of predictions, instead implemented only to communicate clearly to the user what their current gaze vector is predicted to look like. As is evident in Figure 15, The 'Start Controlling Mouse' button is disabled, this is to signal to the user that they must complete the calibration process before accessing this button (stated in instructions box).

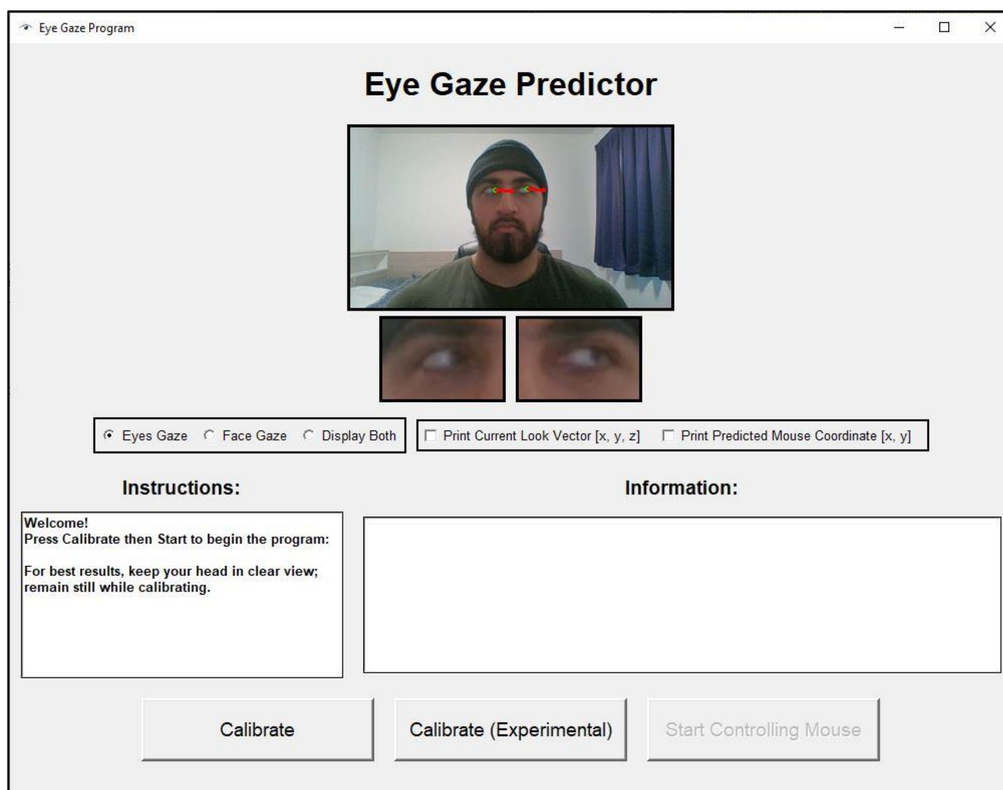


Figure 15: Menu GUI Shown on Program Launch

The functionality has also been implemented to show the user the current look vector as a three-dimensional coordinate as well as the predicted mouse coordinate two-dimensional vector, if the user selects the checkboxes shown in Figure 15. This information is shown in the 'information' box, as showcased in Figure 16, to provide more information to the user about the predictions that are made. In addition to this, the instructions also display the clicks that the user has performed with the mouse controlled by eye movement. These instructions also appear as pop-ups on the top-left corner of the user's device to alert them of any important inputs and information, illustrated in Figures 17 and 18 and as specified in the design wireframes. The 'mouse disabled' message in Figure 18 is displayed when the user inputs "Shift + F5" which exits the mouse control feature of this program. Figure 14 is also in the process of controlling the mouse, therefore the 'start' button has been replaced by a disabled 'Running' button which indicates clearly to the user that the process of controlling the mouse has begun.

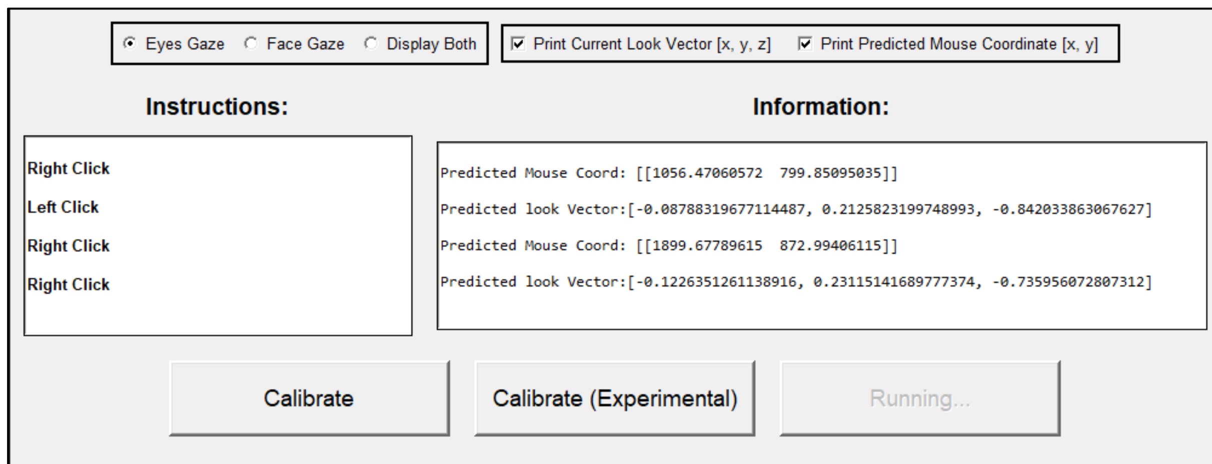


Figure 16: Instructions and Information Shown to User of Predicted Look Vector and Predicted Mouse Coordinate

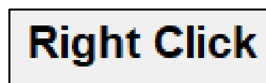


Figure 17: Instruction Pop-Up of 'Right Click' at Top-Left of User's Device

Figure 18:Instruction Pop-Up of 'Mouse Disabled' at Top-Left of User's Device

Similar to Figure 16's 'Running' button, Figure 19 shows that as the calibration process is being complete (Figure 20), the buttons will all replace themselves to show 'Calibrating…' that clearly indicates to the user that the calibration process must be completed before proceeding in the application.

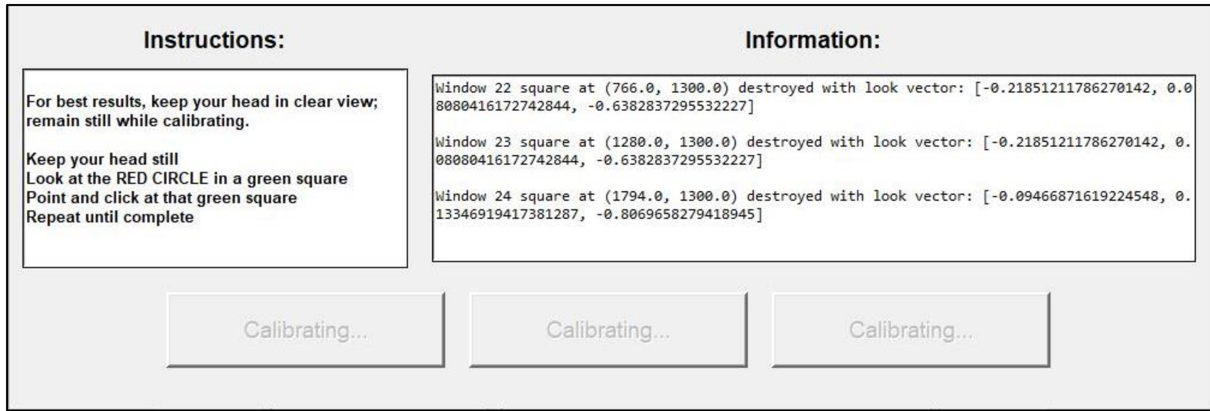


Figure 19: Main Menu UI when Calibration Process is Ongoing

Figure 20 shows what the user is presented with once the ‘Calibrate’ button is selected. Twenty-five equally spaced squares will spawn around the user’s screen. The user is instructed to look at the red dot points and then click on the associated green square, which will store the gaze vector and relevant point coordinates as well as remove that specific point. The window number that has been removed as well as the associated look vector is printed into the information box as seen in Figure 19. This process will be repeated until all points have been removed and the ‘start’ button will become accessible, as initially designed in Figure 13. The green square around each red dot allows the user to easily click on each calibration point without having to manoeuvre the mouse precisely on the red dot point. Overall, this method of calibration is very robust and collects enough points for the Linear Regression to make accurate predictions upon.

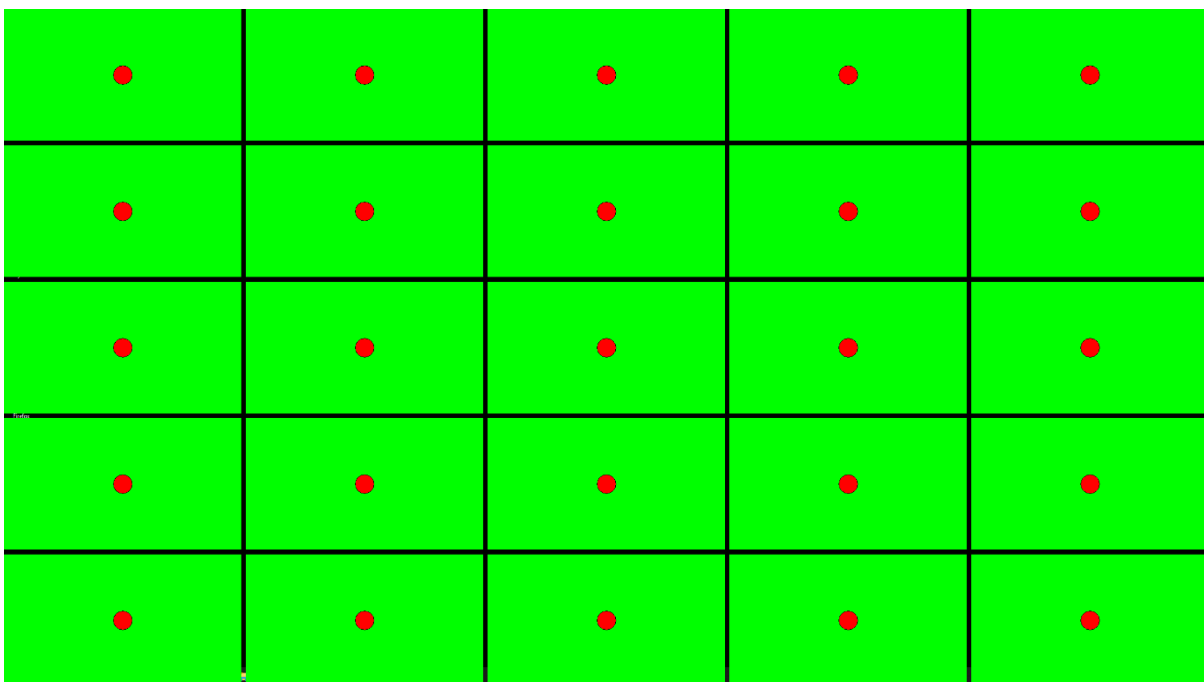


Figure 20: The Calibration Screen Which Displays 25 Calibration Points

3.2.2 Dynamic Behaviour

In the creation process of this application, it was important to maintain functionality of the program in less-than-ideal circumstances, such as those highlighted in 3.1.3. As demonstrated in Figure 21, 22, 23, and 24, the MediaPipe library is still able to detect eye regions even in circumstances where the user's face is either unclear or obscured. Figure 21 shows the user sitting more than a metre away from the webcam and the eye detection algorithm, although somewhat inaccurate at these extended distances, it is still able to crop out the estimated eye regions as inputs to the machine learning model. Figure 22 showcases that the eye regions can be accurately detected even with a high contrast between the brightness of the user's face and the background. Likewise, the eye gaze estimations are able to be made in environments where the user's face and background is dark, such as in Figure 23. Figure 24 showcases the robust nature of the application which will estimate the position of the obscured eye landmarks and still make predictions, albeit less accurate as data from only one eye can be used effectively. Overall, the eye gaze tracker is still robust under many conditions, less so the further away a user is, or if the user's face is partially covered, yet still functional especially in darker environments.

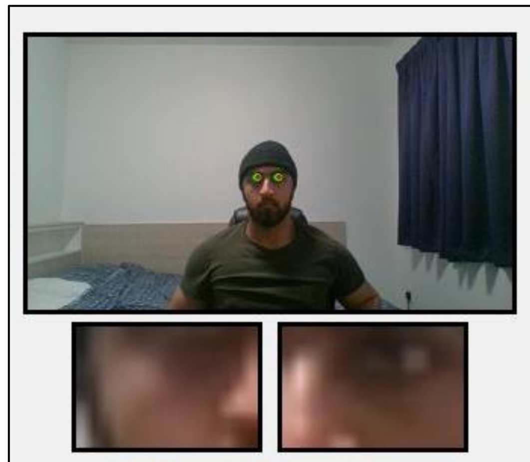


Figure 21: Eye Gaze Vector Still Predicted When User is Located Far Away from Webcam

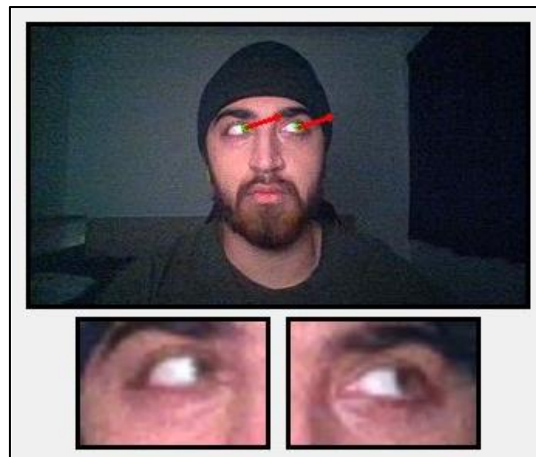


Figure 22: Eye Gaze Vector Still Predicted When User Has a Dark Background Area

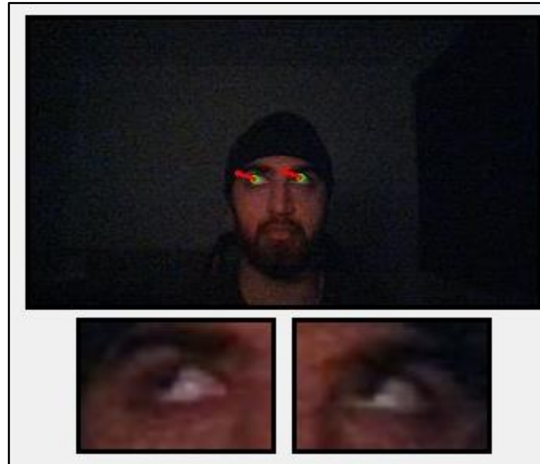


Figure 23: Eye Gaze Vector Still Predicted When User's Face Is in Semi-Darkness

Figure 24: Eye Gaze Vector Still Predicted Even with One Eye Obscured

3.2.3 Algorithms

These are a wide variety of unique and vital algorithms which have been implemented into to this program to meet all the requirements successfully, this section will cover the main algorithms implemented. First is the algorithm that detects the pupil centre as shown below in Figure 25. This function utilises the MediaPipe library which has a coordinate system of mesh points that is called with the variable of chosen iris that stores the indexes of all 4 iris points detected by MediaPipe. Then 'mesh_points' is used to store all of these four points, and '(cx, cy)' calculates the average of these four iris points which is the pupil centre. This is a very innovative way to utilise the MediaPipe library which can not natively detect the pupil itself, so using an average of all four corners of the iris to determine the pupil centre was a lightweight and effective methodology. Popular libraries such as OpenCV, which can detect the largest contours for the pupil, were also considered for this role, but in implementation they were much less accurate and robust in detecting the pupil position than the MediaPipe library.

```
def get_pupil_center (chosen_iris, frame, results_mesh):

    #Stores all x and y coordinates of facial landmarks
    mesh_points = np.array([np.multiply([p.x, p.y], frame.shape[:2][::-
1]).astype(int)
                        for p in
results_mesh.multi_face_landmarks[0].landmark])
    #Define and plot the pupil center position
    (cx, cy), radius = cv2.minEnclosingCircle(mesh_points[chosen_iris])
    pupil_center = np.array([cx, cy], dtype=np.int32)
    return pupil_center
```

Figure 25: Detect Pupil Center using MediaPipe

Another algorithm vital for this program's function is the cropping and normalisation of the eye regions. Figure 26 illustrates this code, where 'eye_x = int(eye[0] - eye_width * 0.5)' calculates the x-coordinate of the top-left corner of the eye region. It takes the x-coordinate of the center of the eye 'eye[0]' and subtracts half of the eye width 'eye_width * 0.5'. Similarly,' eye_y = int(eye[1] - eye_height * 0.65)' calculates the y-coordinate of the top-left corner of the eye region. It takes the y-coordinate of the center of the eye 'eye[1]' and subtracts 65% of the eye height (eye_height * 0.65). This is because the eye height is roughly 70% of eye width, so these values chosen will be able to show the full eye region which is cropped into 120x80 dimensions (in line with SynthesEyes database). The image is also converted into RGB (red green blue) format from the default BGR (blue green red) format so it can be fed into the Regression CNN model.

```
def get_eye_region(eye, eye_width, eye_height, frame):
    eye_x = int(eye[0] - eye_width * 0.5)
    eye_y = int(eye[1] - eye_height * 0.65)

    eye_region = frame[eye_y:eye_y+eye_height, eye_x:eye_x+eye_width]
    return eye_region

cropped_eye_region = cv2.resize(eye_region, (120, 80))

pic_rgb_arr = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
```

Figure 26: Crop Eye Region

Figure 27 showcases the implementation of plotting the eye gaze vector onto the user's webcam feed. Firstly, the pitch and yaw values are calculated from the three dimensional predicted 'look_vector' which is stored as [x, y, z]. A scaling factor 'z_scale' is calculated based on the z-coordinate and the length defined. This scaling factor is used to adjust the length of the arrow based on the depth of the gaze. These pitch and yaw angles are then converted into coordinates 'dx, dy' which is used to find the two-dimensional coordinates on the webcam feed that would accurately represent the z-value (depth) and its magnitude in relation to the pupil centre that was calculated earlier. These are then plotted on the webcam

frame using the OpenCV functionalities ‘circle’ and ‘arrowedLine’. Although initially difficult to solve the problem of representing magnitude of a three-dimensional vector in a two-dimensional environment, this algorithms implementation successfully addresses this issue completely.

```
def get_gaze_values(pupil_center, look_vector, length):
    #Convert look vector to pitch and yaw
    pitch = math.asin(look_vector[1])
    yaw = math.atan2(look_vector[0], -look_vector[2])

    #Define and apply ascaling factor based on the z coordinate and arrow
length
    dz = length * math.cos(pitch) * math.cos(yaw)
    z_scale = max(0.1, dz / length)
    length *= z_scale

    #Convert pitch and yaw angles to position
    dx = length * math.cos(pitch) * math.sin(yaw)
    dy = length * math.sin(pitch)
    dz = length * math.cos(pitch) * math.cos(yaw)

    #Get end point in relation to pupil center
    end_x = int(pupil_center[0] + dx)
    end_y = int(pupil_center[1] + dy)
    #end_z = int(dz)

    return end_x, end_y

def plot_gaze(pupil_center, arrow_end, frame):
    #Plot the pupil center
    cv2.circle(frame, pupil_center, 1, (0,255,0), 2, cv2.LINE_AA)
    #Plot the gaze line
    cv2.arrowedLine(frame, pupil_center, arrow_end[:2], (0, 0, 255),
thickness=2)
```

Figure 27: Two Functions that Plot Eye Gaze Vector on User's Webcam Feed

In Figure 28, the Regression CNN Model is shown, in which the training of this model is completed using Keras in Tensorflow. Keras has abundant documentation and plenty of resources to learn from, and is an interface for Tensorflow. To ensure that the data is a healthy mix of eye types, the order of the dataframe with the look vector and image data has been shuffled, with 80% being the training dataset, 10% being validation data, and the final 10% being test data. This split was chosen to get a large enough sample of random training data, which will greatly help the model in reducing its loss function. In addition to this, it is very important to get a large enough validation set to prevent common machine learning pitfalls such as overfitting as well as evaluate the model’s performance. The testing dataset

was shuffled to ensure that there was no bias in the results and to assess how effective the model is in making predictions from unseen data.

In this function (Figure 28), x_train is the image training data and y_train is the gaze vector training data. In the final line of the model, a custom activation function is also called, in which all vector values beyond the range of -1 to 1 are clipped to equal the closest of these two values. This is done to ensure that all predicted values are within the expected range that the mathematical calculations can later be performed correctly.

This model first creates a stack of layers which allows the creation of defining and adding more layers to the model. Next, A two-dimensional convolutional layer is added which has 32 filters of size 5x5, using the 'relu' activation function. The effect of increasing the number of filters will allow the model to capture more complex patterns in the data, however, it also increases the computational complexity and memory requirements of the model. 32 filters with the size of 5x5 is chosen to get an accurate representation of the input image data, whilst not being overbearing. Also, inputs of only the image shape (120,80) are able to be input into this model. The popular 'relu' function has been implemented because of its functionality that learns complex patterns and make non-linear predictions whilst also being computationally efficient to compute compared to some other activation functions. The next step in this Regression CNN model is to add a max pooling layer which reduces the spatial dimensions of the previous layer's output by taking the maximum value within each 2x2 region. 'MaxPooling2D' is very effective in feature extraction, where the most important features are extracted from the 5x5 dimensions, which in turn reduces the computational complexity of the model. The next line adds another convolutional layer with 64 filters of size 5x5 and 'relu' activation again. This additional convolutional layer allows the model to learn more complex and higher-level features from the down sampled representations in the line before. Again, the representation is down sampled and then through 128 filters with a size of 3x3.

The model is then flattened which reshapes the previous layer into one dimension, preparing it to connect the convolutional layers to the fully connected layers. A very important part of this model is the 'Dropout' layer which prevents overfitting of the model by setting 50% of the input values to 0 which forces the model to improve at generalising and predicting from new unseen image data. Finally, a fully connected dense layer is added, which is then used to return three values that correspond to the [x,y,z] values of the predicted gaze vector. Overall, this model is extremely effective in taking an input image and returning a predicted three-dimensional vector based on the features of the image.

The method in which to evaluate loss is measured using 'mean squared error' which calculates the average of the squared differences between each predicted and true value. As is evident in the code in Figure 28, the popular Adam optimiser is utilised because of its efficacy in regression models which is due to the Adam optimiser's learning rate which can change and adapt to a wide range of values, such as is used in the image data. The best version of this model is then saved using the validation dataset to evaluate the loss value in an unbiased manner. The model was trained using 50 epochs and a batch-size of 8, which was determined as the best values for reducing the loss value using the grid search method (explored further in 4.). Regression CNN was used because of its ability to return a continuous value from image data by detecting patterns within images. Furthermore, it is the most popular and well-documented method in which to make predictions from images, as

CNN are typically used for categorical data. As seen in detail under Appendix E1, this model deals with 815,939 parameters that are updated during this model to optimise its performance.

```
def create_model(X_train, y_train, X_val, y_val):
    from keras import backend as K

    def custom_activation(x):
        return K.clip(x, -0.1, 0.1)

    #Define the CNN architecture
    model = models.Sequential()
    model.add(layers.Conv2D(32, (5, 5), activation='relu',
input_shape=X_train.shape[1:]))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(64, (5, 5), activation='relu'))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(128, (3, 3), activation='relu'))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Flatten())
    model.add(layers.Dropout(0.5))
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(3, activation=custom_activation))

    #Compile the model with an appropriate loss function and optimizer
    model.compile(loss='mean_squared_error', optimizer='Adam')

    cp = ModelCheckpoint('model/', monitor='val_loss', mode='min',
save_best_only=True)

    #Train the model on the dataset
    model.fit(X_train, y_train, epochs=50, batch_size=8,
validation_data=(X_val, y_val), callbacks=[cp])
    return model
```

Figure 28: Training of Regression CNN Model to Predict Eye Gaze Vector from Image Input

The model above (Figure 28) is called twice and is executed on both cropped eye regions, as were seen as the most effective method to do so in Figure 6. Both predicted values are then averaged together and stored to have an overall eye gaze vector which takes in account both eye regions (Figure 29).

```
averaged_look_vector = [(x + y) / 2 for x, y in zip(left_look_vector,
right_look_vector)]
```

Figure 29: Averages Both Predicted Gaze Vectors

Another important algorithm vital for the requirements to be met is the functionality of blinking with the left eye to left-click and the right-eye to right-click. This unique implementation to this application provides an intuitive input method for the user to interact with their device using their eye movements, in this case detecting a wink of one eye to perform the function above. This innovative implementation is seen in Figure 30, where a blinking ratio is calculated (that represents the extent to which a user's eye is closed) for each eye by calculating the distance between the detected landmarks of the eye lids between each other using MediaPipe. The detected landmarks are the top and bottom middle of each eye as well as the corners of the eyes. If the distance between the two points is detected as being below a threshold, the extent to which a user's eyes is detected as being closed increases, and code is executed to left click or right click based on the closed eye.

```
def blinkRatio(img, landmarks, right_indices, left_indices):
    #Right Eyes
    #horizontal line
    rh_right = landmarks[right_indices[0]]
    rh_left = landmarks[right_indices[8]]
    #vertical line
    rv_top = landmarks[right_indices[12]]
    rv_bottom = landmarks[right_indices[4]]

    #Left Eyes
    #horizontal line
    lh_right = landmarks[left_indices[0]]
    lh_left = landmarks[left_indices[8]]

    #vertical line
    lv_top = landmarks[left_indices[12]]
    lv_bottom = landmarks[left_indices[4]]

    rhDistance = euclaideanDistance(rh_right, rh_left)
    rvDistance = euclaideanDistance(rv_top, rv_bottom)

    lvDistance = euclaideanDistance(lv_top, lv_bottom)
    lhDistance = euclaideanDistance(lh_right, lh_left)

    if rvDistance != 0:
        reRatio = rhDistance/rvDistance
    else:
        reRatio = 0

    if lvDistance != 0:
        leRatio = lhDistance/lvDistance
    else:
        leRatio = 0

    return reRatio, leRatio
```

```
def euclaideanDistance(point, point1):
    x, y = point
    x1, y1 = point1
    distance = math.sqrt((x1 - x)**2 + (y1 - y)**2)
    return distance
```

Figure 30: Detects a Blink Based on Distance Between the Two Eyelids and Corners of the Eye

The model that is chosen to predict the screen-coordinate of the cursor based on the eye gaze vector is shown in Figure 31, which is a linear regression model. This is clearly the most appropriate model for this use case because of its simplicity and efficiency, as the relationship between an eye movement, such as a saccade, and the screen coordinate is close to, if not, linear. This mapping may vary from person to person but with the calibration process and linear regression, and accurate prediction can easily be made, evaluated by the 'R-squared score'. This score represents how well a model fits the data between -1 and 1, where 1 is a perfect fit. Through my research and testing, a R-squared score above 0.7 is needed for the program to move the mouse to an accurate screen coordinate based on the eye gaze vector. This program alerts the user if their R-squared score from the linear regression model is below 0.7 (Appendix F1) and will notify the user if the R-squared score is sufficient enough for use (Appendix F2).

```
def create_screen_model(self):
        #Split the data
        X_train, X_test, y_train, y_test = 
train_test_split(self.look_vector_list, self.window_coordinate_list, 
test_size=0.2)

        #Define the model
        screen_model = LinearRegression()

        #Train the model
        screen_model.fit(X_train, y_train)

        #Evaluate the model
        score = screen_model.score(X_test, y_test)
        self.print_to_text(self.information, 'R-squared score:', score)
        if score < 0.7:
            self.show_message("Recommended R-squared score is above 0.7\nYour 
R-squared score is: {}\nPlease calibrate again keeping your head still\nOnly 
move your eyes to the square".format(score))
        else:
            self.show_message("Recommended R-squared score is above 0.7\nYour 
R-squared score is: {}\nPress Start to begin".format(score))
        return screen_model
```

Figure 31: Linear Regression Model and R-Squared Score Calculation

Another main algorithm implemented which is paramount to the function of this application is the calibration method in which user's map their eye gaze to a screen coordinate. The code shown in Figure 32 below first uses the input of 25 squares to equally space these squares out on the user's screen with a slight padding between them to differentiate the calibration squares. Each calibration point is stored as its own window and is coloured 'lime' with a 'red' circle in the middle to represent the specific position of thew calibration point. Once a click has been detected on a calibration point, the point will be destroyed and the associated gaze vector at the time and the points coordinate are stored in two separate lists that are later mapped to each other in the training of the linear regression model.

```
def set_windows(self):
        num_windows = 25
        rows = 5  #number of rows
        cols = 5  #number of columns
        x_start = 0  #starting x-coordinate for first column
        y_start = 0  #starting y-coordinate for first row
        x_step = (self.screen_width - self.spacing * (cols - 1)) /
cols  #distance between columns
        y_step = (self.screen_height - self.spacing * (rows - 1)) /
rows  #distance between rows
        for i in range(num_windows):
            row = int(i / cols)  #calculate which row the window should be in
            col = i % cols  #calculate which column the window should be in
            x1 = x_start + (col * (x_step + self.spacing))
            y1 = y_start + (row * (y_step + self.spacing))
            x2 = x1 + x_step
            y2 = y1 + y_step
            self.create_square_window("Window {}".format(i+1), x1, y1, x2, y2)

def create_square_window(self, name, x1, y1, x2, y2):
        #Adjust window position if it exceeds maximum coordinates
        if x2 > self.screen_width:
            x1 -= (x2 - self.screen_width)
            x2 = self.screen_width
        if y2 > self.screen_height:
            y1 -= (y2 - self.screen_height)
            y2 = self.screen_height

        square = tk.Canvas(window, width=int(x2 - x1), height=int(y2 - y1),
bg="lime", highlightthickness=0)
        square.pack(fill="both", expand=True)
        # calculate the centre of the canvas
        center_x, center_y = (x2 - x1) / 2, (y2 - y1) / 2
        # draw a red circle in the center of the canvas
        radius = 20
        square.create_oval(center_x - radius, center_y - radius, center_x +
radius, center_y + radius, fill="red")
```

```
        square.bind("<Button-1>", lambda event, lv=self.look_vector:
self.on_square_destroyed(name, window))  #destroys the window on left mouse
click and prints the mouse position
        return window

self.print_to_text(self.information, f"{name} square at {center_x,center_y}
destroyed with look vector: ", self.look_vector)
self.look_vector_list.append(self.look_vector)
self.window_coordinate_list.append((x,y))
```

Figure 32: Calibration Method and Storing Values Returned

An ‘experimental calibration method was also implemented (see 3.1.4) in which the user would not have to click on any calibration points, but instead just have their eyes follow a moving square on the device. Shown in Figure 33, the functionality follows the processes defined in the 3.1.4 so the user can calibrate their eye gaze without any inputs necessary. The calibration points traverses the user’s device going left, right, and down, until the whole screen has been traversed by this point in which case the point is destroyed and the start button will become available. The x direction is flipped when the point touches the edge of the device and makes steps of 50 pixels every 0.01 seconds. The time library in python is utilised to ensure that the movements of this calibration point is consistent, regardless of the other processes occurring concurrently.

```
def move_window(x_direction):
            calibration_complete = False
            x, y = window.winfo_x(), window.winfo_y()
            window_width, window_height = window.winfo_width(),
window.winfo_height()
            if x + window_width >= screen_width:
                x = screen_width - window_width
                y += window_height

                if (y + window_height >= screen_height and x == screen_width -
window_width) or (y + window_height >= screen_height and x == screen_width +
window_width):
                    calibration_complete = True
                    window.destroy()
                    self.define_model()
                    self.start_button.config(text="Start", state="normal")
                x_direction = -50  #Flip the x direction
            elif x <= 0:
                x_direction = 50  #Flip the x direction
                y += window_height
            x += 1 * x_direction
            if calibration_complete == False:
                self.window_coordinate_list.append((x+25,y+25))
                window.geometry("+{}+{}".format(x, y))
                self.look_vector_list.append(self.look_vector)
```

```
                #Calculate the time interval since the last call to
move_window
                time_interval = time.perf_counter() -
move_window.last_call_time if hasattr(move_window, 'last_call_time') else 0

                #Adjust the time interval to maintain a consistent speed
                adjusted_time_interval = max(0.01, 0.01 - time_interval)
                #Schedule the next call to move_window with the adjusted time
interval
                window.after(int(adjusted_time_interval * 1000), move_window,
x_direction)
                #Update the last_call_time attribute
                move_window.last_call_time = time.perf_counter()
```

Figure 33: Automatic Calibration Method Implementation

3.2.4 Problems

Some problems were encountered during the creation of this application which hindered some implementation methods and results. For Example, by default SynthesEyes stores all of its data as a .png and an associated .pkl file. All these pkl files have been converted into npz files with the relevant data needed, removing all unnecessary values stored by the database which is not utilised in this program. In Figure 34, a data frame in the Pandas library has been created to store the relevant information for every image in the database, which is then saved as npz files. This has been carried out because there was an abundance of data for the model to process, so to shorten the processing time, npz files were created in which compression is performed using the ZIP compression algorithm, which reduces the file size and allows for efficient storage and retrieval of the data. This problem had taken a substantial amount of time to solve and look for another method in which to represent the database's pkl files relevant information for fast retrieval.

```
def load_pkl_dataset():
    images = np.array([])
    look_vector_x = np.array([])
    look_vector_y = np.array([])
    look_vector_z = np.array([])
    npz_paths = glob.glob('SynthEyes_data/**/*.pkl', recursive=True)

    for name in npz_paths:
        image_file_path, file_look_vector_x, file_look_vector_y,
file_look_vector_z = open_file(name)

        images = np.append(images, image_file_path)
        look_vector_x = np.append(look_vector_x, file_look_vector_x)
        look_vector_y = np.append(look_vector_y, file_look_vector_y)
        look_vector_z = np.append(look_vector_z, file_look_vector_z)

    names_df = pd.DataFrame({'name':[y for y in images],
```

```
                              'x':[x for x in look_vector_x],
                              'y':[y for y in look_vector_y],
                              'z':[z for z in look_vector_z]})

    npz_paths = []
    for i, row in names_df.iterrows():
        picture_path = row['name']
        npz_path = picture_path[:-4] + '.npz'
        npz_paths.append(npz_path)

        pic_bgr_arr = cv2.imread(picture_path)
        pic_rgb_arr = cv2.cvtColor(pic_bgr_arr, cv2.COLOR_BGR2RGB)

        vector_in = np.array([row['x'], row['y'], row['z']])

        np.savez_compressed(npz_path, pic_data=pic_rgb_arr,
vector_in=vector_in)

    names_df['npz_path'] = pd.Series(npz_paths)
    return names_df
```

Figure 34: Conversion of SynthesEyes .pkl Files to .npz Files with Look Vectors Stored

To read the .npz files another algorithm was utilised (Appendix B1), an to validate that the data was being stored correctly in the .npz files, NPZViewer was installed to check the format of these files (Appendix B2).

One frustrating setback was the lack of documentation and surrounding community around the MediaPipe library, which is attributed to its recent creation. Although OpenCV is much more well documented, the accuracy of the eye detection algorithms were not up to the standard needed for this application to function effectively. Therefore, once the capabilities of MediaPipe in face and object detection were demonstrated, it was clear that this was the best implementation. However, MediaPipe not being able to detect pupils, and having scare information on eye detection purposes, much experimentation was carried out with the landmark numbers and methods in order to get the pupil centre detected accurately. The solution was found by averaging out the coordinates of the iris which would return the correct pupil centre.

Another problem encountered in determining eye gaze vector, is since the eye regions are the only parts of the face being fed to the machine learning model, the Wollaston's effect is not taken into account at all (Hecht et al. 2020). This is a famous optical illusion in which the position of a person's head can influence a viewer's perception of the direction of their gaze. The same eye region overlaid on two different head orientations can appear to look at the viewer in one orientation, and at somewhere to the side in another. As seen in Figure 35, This also lends credence to the fact that appearance-based estimation techniques are not as accurate as model-based techniques (Appendix C1)

Figure 35: Example of the Wollaston's effect (Wollaston 1824)

The eye gaze to mouse movement proves slightly difficult to navigate effectively with the jitter of the mouse. Attempts have been made to normalise the movements of the mouse using PyAutoGUI's 'easeInOutQuad' function (Appendix C2), yet this problem has not been fully eradicated. Another problem evident is that the question would be presented that if the user uses the left eye to left-click and right eye to right-click, what happens if they have to blink? This has been taken into account, and if the user's eyes are both detected as closing at the same time, the program will neither perform a left-click nor a right-click. This is to prevent users from mis-clicking every time they have to blink (Appendix C3).

In the initial plan for this project, an idea was included to have functionalities for a keyboard as well as the mouse using eye inputs. This functionality was attempted using Tkinter and the keyboard library to implement an on-screen keyboard, which can take inputs from the user's mouse. However, once this keyboard was created, it was extremely difficult to get an accurate click on any of the keys included as the on-screen keys themselves were too small, and to display them any larger would hinder the visibility of other application of the user's device and would not make for a straightforward user experience. Therefore, the decision was made to remove this feature and instead solely replace the mouse's functionalities with the eye gaze.

There can be instances when the user has started the mouse control portion of the program, but their r-squared score is low (below 0.7) meaning that accurate predictions are not being made. This can make it very difficult to control the mouse effectively, therefore the escape functionality of pressing "Shift + F5" was implemented to exit this process. (Appendix D1) Also, if no eye gaze is detected, the mouse will no longer be moved by eye gaze and the user is able to control the cursor as normal with the computer mouse. This functionality was implemented to avoid situations where the cursor is out to the user's control and they can easily regain control of the cursor with the mouse in this scenario. Also, the eye gaze predictor does not make another prediction if the predicted look vector is the same as the previous look vector, this will save computation time and computer resources making the program run smoother. To improve the performance of the model, which may be a limiting factor on user's devices that are not equipped with sufficient hardware, the quality of webcam camera input has been reduced greatly (Appendix D3). The limit of 30 frames per second has been applied as well as to give the model sufficient time to carry out calculations for each frame detected.

A sizeable amount of time was spent tuning the Regression CNN model to return the most accurate values by minimising the loss function. This process included adding and removing layers such as the ‘dropout layer’ (Figure 28) to minimise the loss value and finding the best method to predict a three-dimensional vector from an image. Also, the idea was explored to create three separate models, one each for the x, y, and z values in a three-dimensional vector. However, since the relation between each one of these values in a look vector is not zero, they would have to be predicted within the same model to detect the hidden links and patterns that lie between the x, y, and z values.

One issue which affected the user’s view of the program and not the functionality is that the initial default webcam view would be flipped when the user would be viewing the feed, so to make the left eye and right eye section appear more intuitively by mirroring the user’s face, the webcam feed is flipped in the final application (Appendix D4).

An issue with the calibration process is that the calibration points would sometimes appear below other programs which the user is using, making them difficult to see and to click on. Therefore, code which pushes all windows to the front of the user’s device, as well as removing their presence in the taskbar to ensure that the user is able to easily interact and identify all the points (Figure 32).

4. Results and Evaluation

This project can be effectively evaluated based on it meeting the functional and non-functional requirements that were set at the beginning of this dissertation (3.1.1).

Functional requirements evaluation:

- The user interface must minimise the number of menus to make the application remain accessible

This requirement has been met with all the entire GUI of the application being in one menu which is launched at the beginning of the program, which increases the level of accessibility by allowing the user to access each process through just one click on this menu.

- The user must be able to view their real-time eye gaze as well as past eye gaze vectors

This requirement has also been met as well as exceeded, because the user is able to view their current eye gaze vector as emanating from their eyes, the middle of their face, or both. The past gaze vectors are all printed in the information box which allows them to access their past data.

- The user must be able to see the webcam feed as well as cropped eye regions

The webcam feed is clearly displayed at the top of the application with the two cropped eye regions positioned directly below, therefore this requirement has been successfully met.

- The user must be able to calibrate the eye gaze vector to device screen at their own pace

There are two calibration methods available for the user in this program. As per this requirement, the first calibration method displays 25 points around the user's screen and there is no time limit for the user to look at and select these calibration points, thus the requirement is met.

- The user must be able to exit the mouse movement process at any time with a key press

This requirement has been met by allowing the user to simply close the program or press "Shift+F5" to exit the process of mouse movement at any time.

- The user must be able to choose the information that is being displayed on the user interface

There are two checkboxes, one for the predicted look vector to be shown and the other is the predicted mouse coordinate. If the user does not wish to see these pieces of information, they can uncheck the boxes so no more predictions are displayed textually on the UI, proving that this requirement is met.

- The user must be able to see if they have performed any actions with the mouse through an alert

A pop-up is displayed on the top-left of the user's device for every left-click or right-click that the user performs, in addition to writing the clicks in the information box on the main GUI. As a result, this requirement has been successfully met.

Non-functional requirements:

- The user must have good performance, given the user has sufficient hardware

This requirement has been partially met, as this program may be taxing on weaker devices, yet it will still run and remain functional on these devices, just not maintain the maximum of 30 predictions per second.

- The user interface should be a professional colour scheme

The user interface was chosen as a sleek grey and white to not cause eye strain to users and remain professional in its presentation, which meets this requirement.

- The user should not experience any crashes/freezing during use

For all the time this program has been run, there are error checks, yet the program has never frozen or crashed in the testing and evaluation process. Therefore, this requirement has been met successfully.

- The user should be able to navigate to every process through just one click

As stated in the functional requirements above, this requirement has been met through the use of the unique calibration method as well as the one-menu system which makes navigating the program extremely trivial.

The functional requirements are important as they allow the user to successfully use the program to fulfil its purpose and the non-functional requirements are chosen as they ensure that the application perform well and is intuitive in navigation. With all these functional and non-functional requirements being successfully met, with some implementations going beyond the brief, it is fair to conclude that this application is successful in achieving its goals of having a machine learning model-based eye gaze tracker with human computer interaction capabilities.

The methodology chosen which involved designing the processes and flowchart before the programming had begun for this project was successful as it allowed me to avoid errors and missteps in logic that would have been more likely if there was not a plan for the order of creation. Python and the main libraries used (Tkinter, TensorFlow, Keras, Sklearn, MediaPipe) were mostly well documented and had many different possible approaches available in addressing the requirements that were defined. Many unique approaches and algorithms had to be implemented for the greatest accuracy and function of the application (see 3.2.3).

Many more tests could be carried out on how to make a more lightweight Regression CNN model that runs on less powerful hardware, as well as removing the need to make two predictions (one for each eye) and instead just predict the three-dimensional gaze vector on just one eye image and still remain accurate.

Many models were created during the attempts to make an accurate eye gaze tracker, which was settled upon using a batch size of 8 and 50 epochs for the regression CNN model. This determination was made using grid search (Appendix G1) which is a technique used to find the best parameters to use in the machine learning model for batch size and epochs. This is completed by exhaustively searching through all the combinations of hyperparameters to determine the best combination. The grid search I had completed took a full 24 hours to complete (Appendix G1) yet was worth the computation power required to determine the

very best combination of hyperparameter to minimise mean squared error with 50 epochs and 8 batch size. The library sklearn was utilised for this process by importing GridSearchCV. This was a very import test to carry out in evaluating at which batch size demonstrates optimal convergence for this use case.

For the linear regression model to determine how accurate the model is a R-squared score is calculated based on the accuracy of the user's calculation as well as the regression CNN model. For the R-squared score, above 0.7 is required for the program to move the cursor to an accurate screen coordinate based on the predicted eye gaze vector. This value represents how well a model fits the data between -1 and 1, where 1 is an ideal fit, and is calculated using the sklearn library in python (Appendix H1). This value is largely dependent on the user's calibration technique and how close they follow the instructions of looking at the calibration point.

The loss value for the Regression model was 0.07 after it was run for 50 epochs on the SynthesEyes database (stored as .npz files and Pandas data frames), with a random 80% of the database as training, 10% as validation, and final 10% as testing data. This value for mean squared error is incredible in this context and lends heavy credence to the fact that accurate gaze estimation can be completed using machine learning and a normal webcam. Albeit, concerns of overfitting and over generalisation are present in such a low loss value, tremendous precautions have been taken to prevent overfitting such as dropouts, shuffled database, and large enough training data. Therefore, it is fair to evaluate that the model is extremely effective in its purpose of estimating gaze direction from an input image of an eye region.

Showcased in Appendix H3 is a graph showing the frequency of each predicted look vector value. As is evident, many of the values are between -1 and 0.5 for [x,y,z] showing that there is a definite trend in the dataset to look vector with those values in a specific direction. Appendix H4 shows a table that is printed which displays each image name in the dataset alongside the predictions made for each one and the actual values. An example of what the training process looks like in the terminal is shown in Appendix H5, where the time estimated to complete training the model and current loss function for a new model is shown.The predictions from Appendix H4 are averaged out in Appendix H6 which shows that the average difference for [x,y,z] from the actual value to the predicted value (using the trained model) is 'Avg_XYZ: [0.05931443 0.05654137 0.04200025]'. Evidently, this model works phenomenally on just a webcam feed to make prediction as the average predicted value for x, y, and z is on average within only 0.06 of the actual value making the model extremely accurate.

5. Conclusions and Future Work

5.1 Conclusion

In review, in just 12 weeks I was effectively able to plan, build, deploy, and test a prototype that tracks eye gaze in real-time using a normal webcam. This was taken further then the initial brief by showcasing and implementing a use case for the eye gaze tracker as an input device which replaces the functionalities of a mouse. The reason for this implementation was that I aimed to provide an inexpensive and accessible method in which to navigate devices for those with disabilities and motor impairments. I have successfully managed to reach a high level of accuracy in the eye gaze estimation with minimal loss, even in environments where the user's face is obscured or in non-ideal lighting conditions. All of the project's aims have been met (1.2) as well as the requirements, which means that the approach to this project was made correctly, with detailed planning and preparation before completing the programming aspect.

Evaluating the many libraries for functionalities such as image processing as well as their quality of documentation was completed in order to select the important libraries require in Python such as MediaPipe, OpenCV, Keras, and Tkinter. I have found in my program, that accurate low-cost webcam eye gaze tracking is entirely possible now with the presence of large databases, such as SynthesEyes, to train a Regression CNN model effectively. Since this process was fully completed in only 12 weeks, the possibilities this application could have and improvements in both accuracy and performance will be amazing to see given enough time and resources.

More importantly, this dissertation is an attempt at bridging the gaps and unjust forms of discrimination that disabled people face as the world improves the technological devices available, yet neglect viable methods for everyone, regardless of ability and level of wealth, to use them. In conclusion, this project demonstrates how breakthroughs in machine learning methods can be used to create low-cost solutions to address inequalities in our society, not the least of which is allowing disabled people to use modern devices using eye gaze tracking in conjunction with human computer interaction.

5.2 Future Work

There are many future applications for this dissertation, in which many features can be added with the goal to optimise and improve the usefulness and efficacy of low-cost webcam eye gaze tracking. For example, this project can be extended with purposes in employee monitoring software, where during meetings or work hours, companies can view if their employees are concentrating on their workstation or focused elsewhere. However, ethical issues of over intrusiveness and lack of privacy are raised from these kinds of bossware which is why there was no further research into this area in this project.

The implementation of an on-screen keyboard was discussed and evaluated in this dissertation as well as the initial plan. However, many issues were present in this idea as discussed in 3.2.4. Yet I believe functionality for an on-screen virtual keyboard can be implemented if the keyboard is a pop-up whenever the user enters a textbox, similar to how mobile devices work with their keyboards. The main limiting factor hindering this from becoming a reality is that the accuracy of the eye gaze tracker and the subsequent cursor must be extremely accurate to select an intended letter on a 26-letter keyboard. With the current r-squared error, this is not feasible to implement into the project at this current state.

If I had a further 12 weeks to complete this dissertation, I would also work on further improving the moving calibration system which I had implemented to be more accessible. This would be changed by having a passive calibration system in the background instead of an explicit one where the user would have to follow points. For example, assumptions can be taken into account that if the user is moving the mouse and clicking with the cursor, there is a high likelihood that they are looking at the cursor while they complete these actions. So passive calibration values can be taken just while the user is using the interface and device instead of completing an entire dedicated calibration process. An interesting research branch would be to see if this method is at all effective at calibration if more time was afforded. I would also like to stretch the limits of MediaPipe to see how far the users can be placed away from the webcam and how exactly certain environment changes affect the accuracy of the eye detection functionality.

Another implementation idea which could easily be extended into this application is the ability to view heatmaps of your eye gaze, on both three-dimensional world space, as well as on the device screen itself. This would allow users to simply view where they spend the most time looking and this functionality could have further functionalities such as if the eye gaze vector is detected as downwards for extended periods of time, the application could remind the user to fix their posture and such. This feature would lend greatly to the goal of this project which is to aid people with disabilities to remain comfortable in using these devices. I would also look for a way to smooth the mouse movements beyond ‘tween=pyautogui.easeInOutQuad’ which did not improve the jitter in the mouse’s movements substantially. Multithreading would also be implemented for the concurrent processes that occur in this program, such as predictions and automatic calibration methods, to ensure that they run more effectively with higher performance on lower-end devices.

6. Reflection

In reflection, this project has allowed me to develop my skills of planning, implementing, testing, and then acting upon my evaluation to produce a completed application that has successfully met the project aim and user requirements. Throughout the development of this dissertation, I have learnt critical skills in how importantly honesty and reflection is upon my own work, in order to assess whether a method to complete a process is the best way requires an introspective look into my own thought process to evaluate with sincerity. My familiarity with libraries in Python such as Keras, Tkinter, and MediaPipe has tremendously improved and I have now achieved a greater complete understanding of the boundaries which each one of these entails and how to overcome them through devising unique solutions.

Furthermore, I have gained a greater sense of perspective on how ostracised people can become due to them not being able to interact with a popular devices and software. Eye gaze tracking is an incredible feature with outstanding applications in a wide variety of fields such as virtual reality, gaming, and productivity. The highlighted importance of eyes and the active role they play in our society in communicating emotions and attention would not be possible without the extent of research and background reading that I have conducted. Utilising the wealth of resources and very recent research conducted in this very field has illustrated to me that eye gaze tracking is a rapidly developing field in which the benefits to the technology in our everyday lives is exciting. However, I have recognised that a pitfall of mine is being overambitious in the introduction of extra features in this program such as keyboards, which may have impacted the results of the final application, yet granted me a greater understanding for the limits my implementation may have.

Overall, through great introspection and by challenging my own knowledge of topics such as eye gaze tracking, I was able to learn much more about the huge applications and prospects that this technology possesses. Throughout this process, I had been afforded the opportunity to be able to make assumptions and decisions on how to solve the problem of eye gaze tracking through my own approach. As a result of this, I am able to assess if the methodologies chosen were accurate and learn from these experiences both encountered problems and triumphs, to make more informed decisions in the future and plan ahead more effectively.

Appendices

Appendix A:

Image A1:

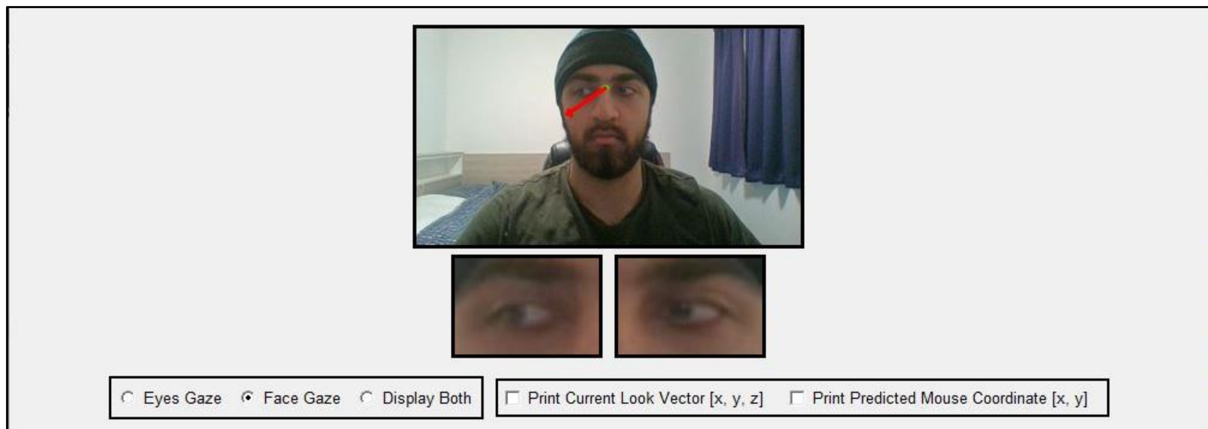
```
self.selected_display = tk.IntVar(value=1)
self.plot_eyes_radiobutton = tk.Radiobutton(plot_checkbox_frame, text="Eyes
Gaze", font=("Helvetica", 10), bd=0, variable=self.selected_display, value=1)
self.plot_eyes_radiobutton.pack(side=tk.LEFT, pady=5, padx=5)

self.plot_face_radiobutton = tk.Radiobutton(plot_checkbox_frame, text="Face
Gaze", font=("Helvetica", 10), bd=0, variable=self.selected_display, value=2)
self.plot_face_radiobutton.pack(side=tk.LEFT, pady=5, padx=5)

self.plot_both_radiobutton = tk.Radiobutton(plot_checkbox_frame, text="Display
Both", font=("Helvetica", 10), bd=0, variable=self.selected_display, value=3)
self.plot_both_radiobutton.pack(side=tk.LEFT, pady=5, padx=5)
```

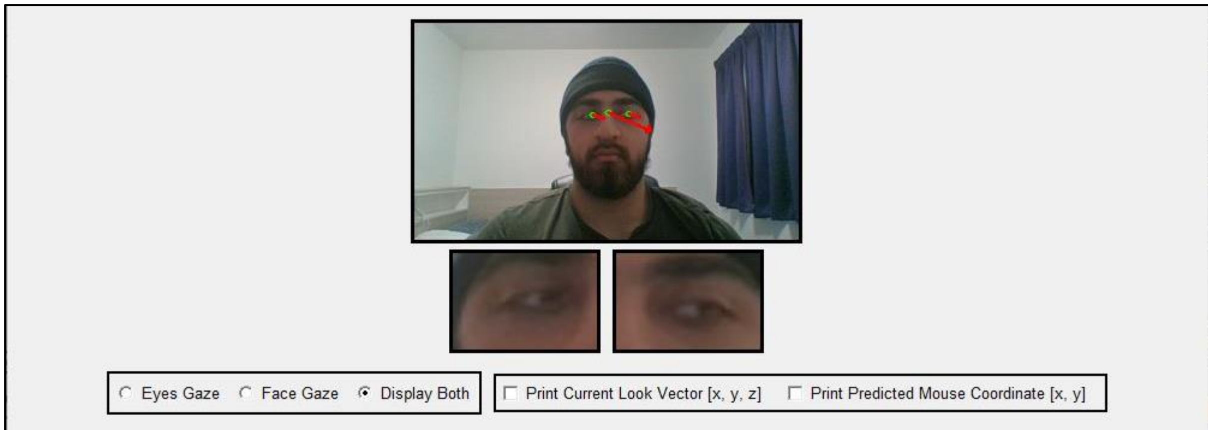
Radio button Implementation for Displayed Gaze

Image A2:



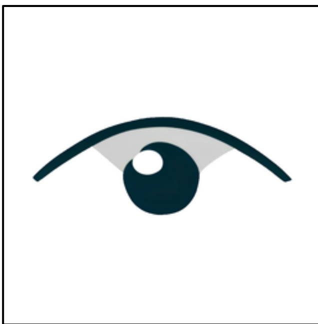
Radio Button 'Face Gaze' Selected Displaying Predicted Eye Gaze Vector Visualisation from User's Center of Face

Image A3



Radio Button 'Display Both' Selected Displaying Predicted Eye Gaze Vector Visualisation from User's Center of Face and Eyes

Image A4



Appendix B:

Image B1

```
def get_x_y(df):
    x = []
    y = []
    for name in df['name']:
        loaded_npz = np.load(name)

        pic = loaded_npz['pic_data']
        x.append(pic)

        vector_x, vector_y, vector_z = loaded_npz['vector_in']
        y.append([vector_x, vector_y, vector_z])

    x = np.array(x)
    y = np.array(y)
    return x, y
```

Image B2

```
pythonw

Load .npz file

keys are :['pic_data', 'vector_in']
--- *** ---
pic_data - shape: (80, 120, 3) - :
[[[ 97  56  26]
  [110  55  27]
  [103  59  29]
  ...
  [  0   0   0]
  [  0   0   0]
  [  0   0   0]]

 [[103  55  29]
  [110  56  27]
  [102  64  39]
```

Appendix C:

Image C1 (Modi and Singh, 2021)

Evaluation criteria	Model-based	Appearance-based	Hybrid
Setup complexity	High	Low	Low
System calibration	Fully calibrated	✗	Intermediate
Hardware (camera) requirements	Two or more	ordinary camera	Ordinary web camera
Implicit robustness to head movements	medium–high	Low	Low
Implicit robustness to varying illumination	medium–high	Low	Low
Gaze estimation accuracy error	low (<1°)	(>2°) high	(1–3°) high

Image C2

```
def move_mouse_to_point(self, point):
```

```
        pyautogui.moveTo(point[0][0], point[0][1],
tween=pyautogui.easeInOutQuad)
```

Image C3

```
if reRatio > 6.5 and leRatio < 6.5:
                    pyautogui.rightClick()
                    self.show_message("Right Click")
                    #print("RIGHT EYE BLINKING")
                elif leRatio > 6.5 and reRatio < 6.5:
                    pyautogui.leftClick()
                    self.show_message("Left Click")
                    #print("LEFT EYE BLINKING")
```

Appendix D:

Image D1

```
keyboard.add_hotkey('shift+f5', self.disable_mouse)
```

Image D2

```
old_look_vector = self.look_vector
if old_look_vector != self.look_vector:
      #make predictions
```

Image D3

```
cap = cv2.VideoCapture(0)
cap.set(cv2.CAP_PROP_FPS, 30)
cap.set(3, 64)
cap.set(4, 48)
```

Image D4

```
averaged_look_vector = [(x + y) / 2 for x, y in zip(left_look_vector,
right_look_vector)]
```

Appendix E:

Image E1

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 76, 116, 32)       2432

 max_pooling2d (MaxPooling2D  (None, 38, 58, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 34, 54, 64)        51264

 max_pooling2d_1 (MaxPooling  (None, 17, 27, 64)       0
 2D)

 conv2d_2 (Conv2D)           (None, 15, 25, 128)       73856

 max_pooling2d_2 (MaxPooling  (None, 7, 12, 128)       0
 2D)

 flatten (Flatten)           (None, 10752)             0

 dropout (Dropout)           (None, 10752)             0

 dense (Dense)               (None, 64)                688192

 dense_1 (Dense)             (None, 3)                 195

=================================================================
Total params: 815,939
Trainable params: 815,939
Non-trainable params: 0
```

Appendix F:

Image F1

Recommended R-squared score is above 0.7
Your R-squared score is: 0.6468598984318104
Please calibrate again keeping your head still
Only move your eyes to the square

Image F2

Recommended R-squared score is above 0.7
Your R-squared score is: 0.774047400020127
Press Start to begin

Appendix G:

Image G1

```
from sklearn.model_selection import GridSearchCV

def get_grid_search(X_train, y_train, X_val, y_val):
    model = KerasRegressor(build_fn=create_model, X_train=X_train,
y_train=y_train, X_val=X_val, y_val=y_val, verbose=1)

    #Define the grid search parameters
    batch_size = [4, 8, 16]
    epochs = [10, 20, 50]
    param_grid = dict(batch_size=batch_size, epochs=epochs)

    #Perform the grid search
    grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1,
cv=3)
    grid_result = grid.fit(X_train, y_train)

    #Print the best parameters
    print("Best: %f using %s" % (grid_result.best_score_,
grid_result.best_params_))

    return grid_result
```

Appendix H:

Image H1

```
screen_model = LinearRegression()

#Train the model
screen_model.fit(X_train, y_train)

#Evaluate the model (get r-squared score)
score = screen_model.score(X_test, y_test)
```

Image H2

```
                                           name         x         y         z                                       npz_path
0       SynthEyes_data\f01\f01_1002_0.1963_0.3927.png -0.399155 -0.147319 -0.904971   SynthEyes_data\f01\f01_1002_0.1963_0.3927.npz
1       SynthEyes_data\f01\f01_1003_0.1963_0.1963.png -0.212617 -0.155279 -0.964719   SynthEyes_data\f01\f01_1003_0.1963_0.1963.npz
2      SynthEyes_data\f01\f01_1004_0.1963_-0.0000.png -0.017909 -0.157271 -0.987393  SynthEyes_data\f01\f01_1004_0.1963_-0.0000.npz
3      SynthEyes_data\f01\f01_1005_0.1963_-0.1963.png  0.177488 -0.153221 -0.972122  SynthEyes_data\f01\f01_1005_0.1963_-0.1963.npz
4      SynthEyes_data\f01\f01_1010_-0.0000_0.5890.png -0.571396  0.029006 -0.820162  SynthEyes_data\f01\f01_1010_-0.0000_0.5890.npz
...                                               ...       ...       ...       ...                                            ...
11377   SynthEyes_data\m05\m05_947_-0.3927_0.5890.png -0.558585  0.351681 -0.751201   SynthEyes_data\m05\m05_947_-0.3927_0.5890.npz
11378   SynthEyes_data\m05\m05_949_-0.3927_0.1963.png -0.198649  0.415299 -0.887730   SynthEyes_data\m05\m05_949_-0.3927_0.1963.npz
11379  SynthEyes_data\m05\m05_950_-0.3927_-0.0000.png -0.003631  0.423636 -0.905825  SynthEyes_data\m05\m05_950_-0.3927_-0.0000.npz
11380  SynthEyes_data\m05\m05_952_-0.3927_-0.3927.png  0.379327  0.391774 -0.838227  SynthEyes_data\m05\m05_952_-0.3927_-0.3927.npz
11381  SynthEyes_data\m05\m05_959_-0.5890_-0.0000.png -0.003447  0.592215 -0.805773  SynthEyes_data\m05\m05_959_-0.5890_-0.0000.npz

[11382 rows x 5 columns]
```

Image H3

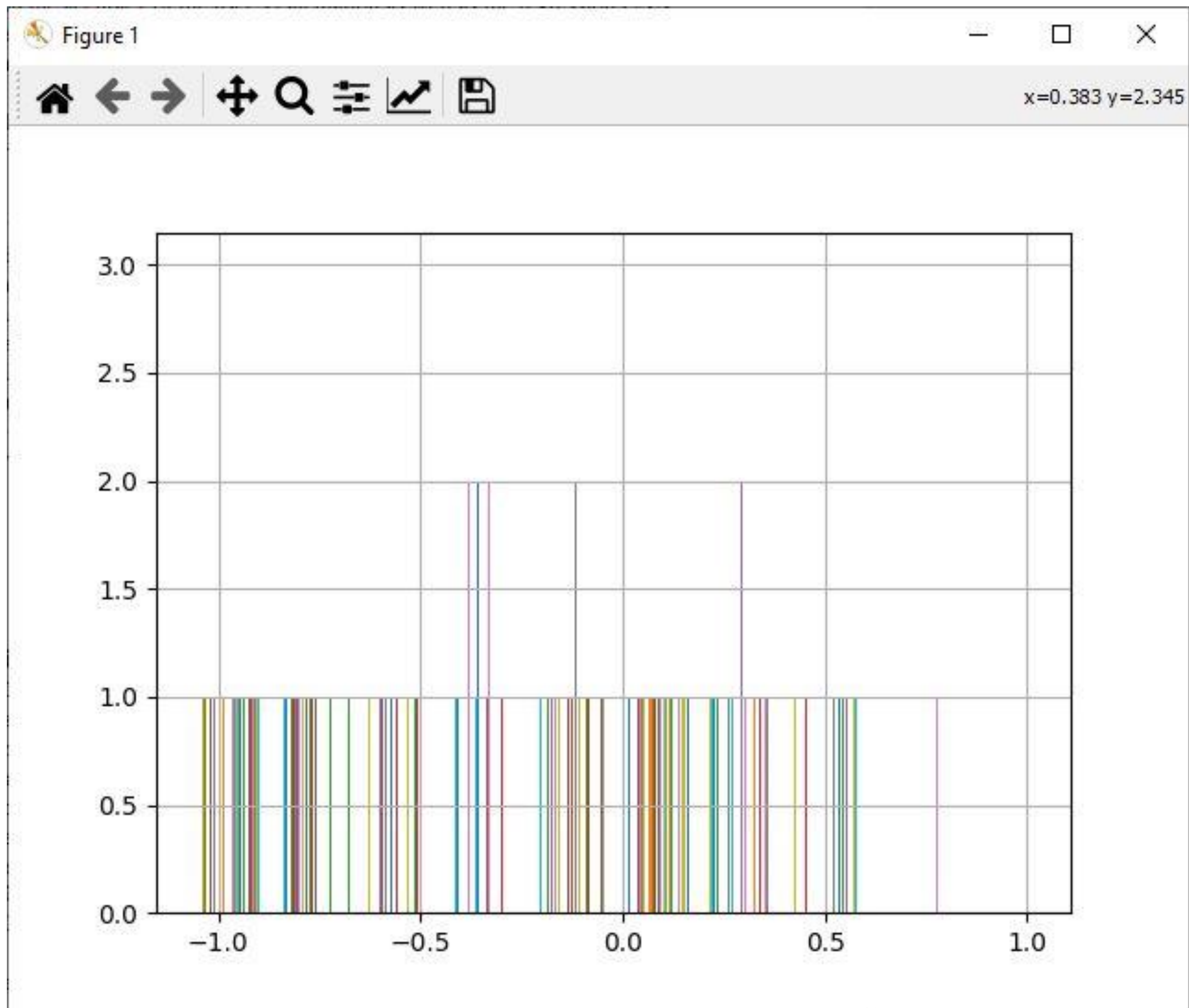


Image H4

	name	x	y	z	Predicted look_vector	difference
0	SynthEyes_data\m01\m01_611_-0.0000_-0.5890.npz	0.603761	0.032942	-0.796485	[0.5099607706069946, -0.0011753290891647339, -0.8402971029281616]	[0.09380001 0.03411733 0.04381251]
1	SynthEyes_data\f01\f01_3012_0.5890_-0.1963.npz	0.222877	-0.563136	-0.795741	[0.14215245842933655, -0.5769340991973877, -0.7850813269615173]	[0.08072427 0.0137983 0.01065987]
2	SynthEyes_data\f02\f02_2082_-0.3927_0.3927.npz	-0.389078	0.317233	-0.864859	[-0.41574400663375854, 0.1902363896369934, -0.8525264263153076]	[0.02666584 0.12699646 0.0123328]
3	SynthEyes_data\m03\m03_229_-0.5890_0.1963.npz	-0.167904	0.585676	-0.792964	[-0.1468977928161621, 0.5540487766265869, -0.8952593803405762]	[0.02100661 0.0316273 0.10229552]
4	SynthEyes_data\m03\m03_2870_0.1963_-0.5890.npz	0.604847	-0.178454	-0.776089	[0.569503128528595, -0.10041120648384094, -0.7638197541236877]	[0.03534406 0.07804245 0.0122692]
5	SynthEyes_data\m02\m02_3607_-0.0000_-0.3927.npz	0.414413	-0.0231566	-0.909794	[0.5055444836616516, -0.0016301348805427551, -0.8425006866455078]	[0.09113175 0.02152649 0.0672937]
6	SynthEyes_data\f04\f04_844_0.1963_-0.3927.npz	0.412023	-0.138088	-0.900649	[0.5003182291984558, -0.10335476696491241, -0.8001965880393982]	[0.08829561 0.03473283 0.10045266]
7	SynthEyes_data\f04\f04_2967_-0.1963_-0.1963.npz	0.256867	0.179872	-0.949561	[0.20673975348472595, 0.15126168727874756, -0.9469684362411499]	[0.05012751 0.0286105 0.00259215]
8	SynthEyes_data\f05\f05_2447_0.5890_-0.5890.npz	0.574908	-0.470639	-0.669313	[0.4775581657886505, -0.5324438810348511, -0.6221411228179932]	[0.09734967 0.06180525 0.04717213]
9	SynthEyes_data\m05\m05_796_-0.5890_0.1963.npz	-0.163895	0.581642	-0.796763	[-0.25771456956863403, 0.49509239196777344, -0.8618947267532349]	[0.09381939 0.08654946 0.06513184]
10	SynthEyes_data\f01\f01_2536_0.3927_-0.3927.npz	0.394647	-0.360769	-0.845044	[0.354897141456604, -0.26085445284843445, -0.8094972372055054]	[0.03974959 0.09991473 0.03554684]
11	SynthEyes_data\f03\f03_1167_0.1963_-0.1963.npz	0.262075	-0.174416	-0.949155	[0.28292331099510193, -0.1758890151977539, -0.8864117860794067]	[0.02084854 0.00147338 0.0627436]
12	SynthEyes_data\f03\f03_1976_0.1963_-0.0000.npz	0.0180387	-0.191608	-0.981306	[-0.09882207959890366, -0.18112343549728394, -0.9645154476165771]	[0.11686078 0.01048414 0.01679039]
13	SynthEyes_data\m02\m02_392_-0.5890_-0.0000.npz	0.000332542	0.61029	-0.792178	[0.10542985796928406, 0.6286625266075134, -0.8659512400627136]	[0.10509732 0.01837224 0.07377344]
14	SynthEyes_data\f03\f03_3849_-0.0000_-0.1963.npz	0.175481	-0.0523259	-0.983091	[0.21846386790275574, -0.09370262920856476, -0.9270375370979309]	[0.04298262 0.04137677 0.05605364]
15	SynthEyes_data\f02\f02_1741_-0.0000_0.1963.npz	-0.134432	-0.0413257	-0.990061	[-0.058907635509967804, -0.029717370867729187, -1.0271269083023071]	[0.07552438 0.01160829 0.03706622]
16	SynthEyes_data\f03\f03_2624_0.1963_-0.0000.npz	-0.00068404	-0.210323	-0.977632	[0.0017991997301578522, -0.049756474792957306, -1.0541503429412842]	[0.00248324 0.16056667 0.0765186]
17	SynthEyes_data\f02\f02_2398_-0.1963_0.1963.npz	-0.152327	0.1337	-0.979245	[-0.13038551807403564, 0.17536139488220215, -0.9649810791015625]	[0.02194159 0.0416614 0.01426393]
18	SynthEyes_data\m03\m03_2631_-0.0000_0.3927.npz	-0.38991	-0.0163125	-0.920709	[-0.4760841131210327, 0.027012109756469727, -0.8549546599388123]	[0.08617452 0.04332459 0.065754]
19	SynthEyes_data\f02\f02_233_-0.5890_-0.5890.npz	0.579723	0.464076	-0.669743	[0.7693682909011841, 0.43606483936309814, -0.6111143827438354]	[0.18964553 0.0280112 0.05862814]
20	SynthEyes_data\m03\m03_2463_0.1963_-0.1963.npz	0.221119	-0.207656	-0.952883	[0.08440768718719482, -0.210953027009964, -0.9789751172065735]	[0.13671111 0.00329727 0.02609229]
21	SynthEyes_data\f03\f03_3830_0.3927_-0.0000.npz	-0.0212066	-0.43005	-0.902556	[-0.04389079660177231, -0.3669206500053406, -0.9509004354476929]	[0.02268423 0.06312984 0.04834473]
22	SynthEyes_data\m01\m01_1094_-0.0000_-0.0000.npz	-0.0535785	0.0517803	-0.99722	[-0.17078307271003723, 0.00024759024381637573, -0.9003993272781372]	[0.11720458 0.05153267 0.09682089]
23	SynthEyes_data\f03\f03_3836_0.1963_0.5890.npz	-0.572882	-0.199612	-0.79496	[-0.5732141733169556, -0.13592448830604553, -0.8470707535743713]	[0.00033259 0.06368735 0.05211055]
24	SynthEyes_data\f01\f01_303_-0.3927_-0.1963.npz	0.209164	0.422155	-0.882064	[0.25361618399620056, 0.4046393930912018, -0.863187313079834]	[0.04445262 0.01751527 0.01887625]
25	SynthEyes_data\f01\f01_1029_-0.3927_0.3927.npz	-0.400572	0.384054	-0.831892	[-0.28149768710136414, 0.2743024230003357, -0.88951575756073]	[0.11907414 0.10975185 0.05762362]
26	SynthEyes_data\f03\f03_3747_0.3927_0.3927.npz	-0.383335	-0.397742	-0.83358	[-0.4490416944026947, -0.42234334349632263, -0.745064914226532]	[0.06570667 0.02460119 0.08851498]
27	SynthEyes_data\f04\f04_2215_0.1963_0.7854.npz	-0.721175	-0.126898	-0.681031	[-0.6261352300643921, -0.13803116977214813, -0.7644368410110474]	[0.09503955 0.01113282 0.08340544]
28	SynthEyes_data\f03\f03_63_-0.3927_-0.7854.npz	0.75942	0.281749	-0.586429	[0.8540953397750854, 0.43616271018981934, -0.5709347724914551]	[0.09467542 0.15441334 0.01549399]

Image H5

```
Epoch 1/5
1121/1139 [============================>.] - ETA: 1s - loss: 0.2968
```

Image H6

```
Avg_XYZ:  [0.05931443 0.05654137 0.04200025]
```

References

Bick, A., Arizona State University, Blandin, A., Mertens, K., Virginia Commonwealth University and Federal Reserve Bank of Dallas 2020. Work from home after the COVID-19 outbreak. Federal Reserve Bank of Dallas, Working Papers 2020(2017). Available at: http://dx.doi.org/10.24149/wp2017

Cazzato, D., Leo, M., Distante, C. and Voos, H. 2020. When I look into your eyes: A survey on computer vision contributions for human gaze estimation and tracking. Sensors (Basel, Switzerland) 20(13), p. 3739. Available at: http://dx.doi.org/10.3390/s20133739.

Cecílio, J., Andrade, J., Martins, P., Castelo-Branco, M. and Furtado, P. 2016. BCI framework based on games to teach people with cognitive and motor limitations. Procedia computer science 83, pp. 74–81. Available at: http://dx.doi.org/10.1016/j.procs.2016.04.101.
Clay, V., König, P. and König, S. 2019. Eye tracking in Virtual Reality. Journal of eye movement research 12(1). Available at: http://dx.doi.org/10.16910/jemr.12.1.3.

Easy to use, small, portable eye tracker - Tobii Pro Nano. Available at: https://www.tobii.com/products/eye-trackers/screen-based/tobii-pro-nano [Accessed: 7 April 2023]

GazePointer. 2016. Available at: https://gazerecorder.com/gazepointer/

Ghani, M.U., Chaudhry, S., Sohail, M. and Geelani, M.N. 2013. GazePointer: A real time mouse pointer control implementation based on eye gaze tracking. In: INMIC. IEEE

Gudi, A., Li, X. and van Gemert, J. 2020. Efficiency in real-time webcam gaze tracking. In: Computer Vision – ECCV 2020 Workshops. Cham: Springer International Publishing, pp. 529–543.

Hecht, H., Siebrand, S. and Thönes, S. 2020. Quantifying the Wollaston illusion. Perception 49(5), pp. 588–599. Available at: http://dx.doi.org/10.1177/0301006620915421.

Hietanen, J.O., Peltola, M.J. and Hietanen, J.K. 2020. Psychophysiological responses to eye contact in a live interaction and in video call. Psychophysiology 57(6), p. e13587. Available at: http://dx.doi.org/10.1111/psyp.13587

Ole Baunbæk Jensen. (2022). Webcam eye tracking vs. an eye tracker: iMotions. https://imotions.com/blog/learning/best-practice/webcam-eye-tracking-vs-an-eye-tracker/

Itier, R.J. and Batty, M. 2009. Neural bases of eye and gaze processing: the core of social cognition. Neuroscience and biobehavioral reviews 33(6), pp. 843–863. Available at: http://dx.doi.org/10.1016/j.neubiorev.2009.02.004.

Keras: Deep learning for humans. Available at: https://keras.io/

MediaPipe. Available at: https://developers.google.com/mediapipe

Meta Quest Pro. Available at: https://www.meta.com/gb/quest/quest-pro/

Microsoft 2022. Discover new features in Windows 11. Available at: https://www.microsoft.com/en-gb/windows

Modi, N. and Singh, J. 2021. A review of various state of art eye gaze estimation techniques. In: Advances in Intelligent Systems and Computing. Singapore: Springer Singapore, pp. 501–510.

Npzviewer. Available at: https://pypi.org/project/npzviewer/

PSVR2. Available at: https://direct.playstation.com/en-gb/playstation-vr2

OpenCV Library 2021. Home. Available at: https://opencv.org/

Orduna-Hospital, E., Navarro-Marqués, A., López-de-la-Fuente, C. and Sanchez-Cano, A. 2023. Eye-tracker study of the Developmental Eye Movement test in young people without binocular dysfunctions. Life (Basel, Switzerland) 13(3). Available at: http://dx.doi.org/10.3390/life13030773.

Python. Available at: https://www.python.org/

Roy, K. and Chanda, D. 2022. A robust webcam-based eye gaze estimation system for human-computer interaction. In: 2022 International Conference on Innovations in Science, Engineering and Technology (ICISET). IEEE

Shishido, E., Ogawa, S., Miyata, S., Yamamoto, M., Inada, T. and Ozaki, N. 2019. Application of eye trackers for understanding mental disorders: Cases for schizophrenia and autism spectrum disorder. Neuropsychopharmacology reports 39(2), pp. 72–77. Available at: http://dx.doi.org/10.1002/npr2.12046.

Singh, A.K., Kumbhare, V.A. and Arthi, K. 2022. Real-time human pose detection and recognition using MediaPipe. In: Advances in Intelligent Systems and Computing. Singapore: Springer Singapore, pp. 145–154.

Solska, K. and Kocejko, T. 2022. Eye-tracking everywhere - software supporting disabled people in interaction with computers. In: 2022 15th International Conference on Human System Interaction (HSI). IEEE

Song, F., Zhou, S., Gao, Y., Hu, S., Zhang, T., Kong, F. and Zhao, J. 2021. Are you looking at me? Impact of eye contact on object-based attention. Journal of experimental psychology. Human perception and performance 47(6), pp. 765–773. Available at: http://dx.doi.org/10.1037/xhp0000913.

Šumak, B., Špindler, M., Debeljak, M., Heričko, M. and Pušnik, M. 2019. An empirical evaluation of a hands-free computer interaction for users with motor disabilities. Journal of biomedical informatics 96(103249), p. 103249. Available at: http://dx.doi.org/10.1016/j.jbi.2019.103249.

Syrjämäki, A.H., Isokoski, P., Surakka, V., Pasanen, T.P. and Hietanen, J.K. 2020. Eye contact in virtual reality – A psychophysiological study. Computers in human behavior 112(106454), p. 106454. Available at: http://dx.doi.org/10.1016/j.chb.2020.106454.

Valliappan, N. et al. 2020. Accelerating eye movement research via accurate and affordable smartphone eye tracking. Nature communications 11(1), p. 4553. Available at: http://dx.doi.org/10.1038/s41467-020-18360-5.

Wang, L., Shi, X. and Liu, Y. 2023. Foveated rendering: A state-of-the-art survey. Computational visual media 9(2), pp. 195–228. Available at: http://dx.doi.org/10.1007/s41095-022-0306-4.

Werth, A.J. and Babski-Reeves, K. 2012. Assessing posture while typing on portable computing devices in traditional work environments and at home. Proceedings of the Human Factors and Ergonomics Society ... Annual Meeting. Human Factors and Ergonomics Society. Annual Meeting 56(1), pp. 1258–1262. Available at: http://dx.doi.org/10.1177/1071181312561223.

Wollaston, W.H. 1824. XIII. On the apparent direction of eyes in a portrait. Philosophical transactions of the Royal Society of London 114(0), pp. 247–256. Available at: http://dx.doi.org/10.1098/rstl.1824.0016.

Wong, R. 2020. When no one can go to school: does online learning meet students' basic learning needs? Interactive learning environments 31(1), pp. 434–450. Available at: http://dx.doi.org/10.1080/10494820.2020.1789672

Wood, E., Baltrusaitis, T., Zhang, Z., Sugano, Y., Bulling, A. and Robinson, P. 2020. SynthesEyes Dataset.

Zander, T.O., Gaertner, M., Kothe, C. and Vilimek, R. 2010. Combining eye gaze input with a brain–computer interface for touchless human–computer interaction. International journal of human-computer interaction 27(1), pp. 38–51. Available at: http://dx.doi.org/10.1080/10447318.2011.535752.