

GAZE ESTIMATION PROJECT:

Summary: built a Python-based eye-gaze detection system with OpenCV/dlib that enabled users to type using a virtual keyboard via blink and gaze tracking. Designed for accessibility and low-cost hardware (standard webcam).

Analysis

Problem identification

An eye gaze communication frame is a transparent board with pictures or letters placed around a hole in the center of the board. It acts as means of communications for those who struggle with other forms of communication due to a physical/mental impairment. Observing which letter or picture a person is looking at, through the hole in the center of the frame, is how it is used. This low-tech approach usually takes some time and mistakes can easily be made, which could be very frustrating for both the carer and the person trying to communicate.

Therefore, many tech companies such as Tobii and eye gaze Inc. have created an assistive technology that uses an eye tracking camera, to follow the movement of a person's eyes around a keyboard on a screen. By computationally solving this issue it could enable someone to communicate much faster, and to a consistent, and hopefully higher, degree of accuracy (as they wouldn't have to rely on another to figure what they are trying to communicate).

Although these current computational approaches are extremely beneficial to those that require such assistance to communicate, the technology is very expensive when sold commercially and requires specific hardware. Therefore, the aim of my project, in its simplest form, is to create an application that uses a web cam to detect when a user blinks enabling them to select an item/letter and follow the movement of the user's pupil to indicate whereabouts, the keyboard displayed onscreen, they are looking.

Computational problem

Although there is an alternative method to this problem, there are still many reasons why it should be solved computationally. As compared to humans, computers tend to be much more accurate and consistent in their performance. It doesn't require extensive training or months/years of experience to get good at a task and remain good at it for computers. Therefore, this problem is suited to a computational solution as from the point the person starts using the application, the code will perform as it should. So, this will hopefully enable the user to quickly and accurately communicate what they want to say without relying on other people to understand them, whose performance and levels of experience may vary.

As well as the problem being more suited to a computational method, gaze/blink technology has been implemented before, using various different methods, therefore we can confidently assume it's possible to solve computationally. Most of these methods require a webcam, which in order to work, require a computer for it to be connected to. Therefore, a computer must be used to gather the data recorded by the webcam.

The main aim of the software is to provide a way for people, who cannot speak and/or have a motor impairment, to interact with others via a computer. The computable part of the problem is how we use computer vision to detect the location of the user's pupil and translate that onto the screen to indicate where the cursor should be. As well as detecting when/where the person blinks on the screen in order to select. Therefore, the main problem is that we need to use computer vision contouring so that an open and closed eye can be distinguished, and the pupil of the user's eye can be recognized.

Thinking Abstractly

Abstraction is used in both of the main two problems. Both problems use a video input of user's face, where each frame of the video shows the users entire face however for gaze detection, we only need to focus on a specific region of interest, and that's the eye. Therefore, for this problem all features apart from the eye can be abstracted away as they aren't necessary. Another example of abstraction being used in gaze detection is the use of grey scale and blur, meaning color and noise within the frame is abstracted away in order to produce a clearer pupil contour which can be detected more easily.

Blink detection: abstraction is used when detecting facial landmarks for the eye length and width, using the dlib library. Unnecessary parts/features from the landmark predictor are abstracted away, as when as the colors and noise of the frames through the incorporation of greyscale, to enhance landmark predictors accuracy in detecting the corners and top/bottom of the eye.

Thinking procedurally

The gaze detection problem can be broken down into these main sub procedures:

1. Web cam records real time vid of user.
2. identify pupil.
3. Compares data to calibration
4. Using algorithm to estimate gaze location on screen

The gaze location will then be used to control the movement of cursor and the second problem involves detecting when the eye blinks to invoke selection. This problem can be broken down into:

1. Detecting facial landmarks for eye on live webcam recording
2. Using the eye aspect ratio to determine whether the eye is blinking

Thinking ahead

blink detection: the main input in order to determine the presence of a blink will be the length and width ratio of the eye in each frame, where the input will be a float data value that must be above a limit value in order for a blink to be detected. The duration of blink is also input as a float data value and depending whether the duration is <0.5, 0.5-1 or 2-3 seconds, will output whether user is not selecting/holding a key, clicking a key, or holding a key. The preconditions for these inputs are that the user's eye is within the screens frame, the webcam is at least 640x480 resolution and the user isn't moving too much and has a relatively straight face.

Gaze detection: the main input for the gaze detection will be the location of the pupil in each frame within the web cam screen. This input will be a real xy coordinate value and will be used to translate the position of the pupil relative to the pupil's position during the calibration to the position of the cursor on the screen. This input will only be valid assuming the user's eyes are in frame, the circle

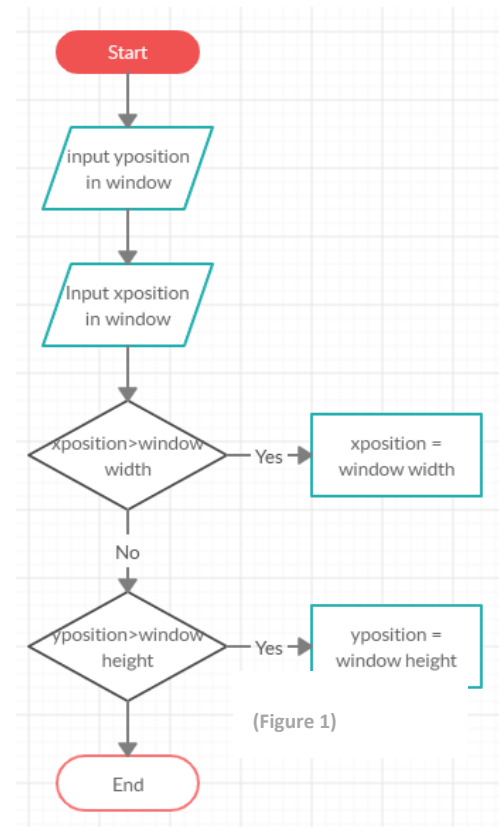
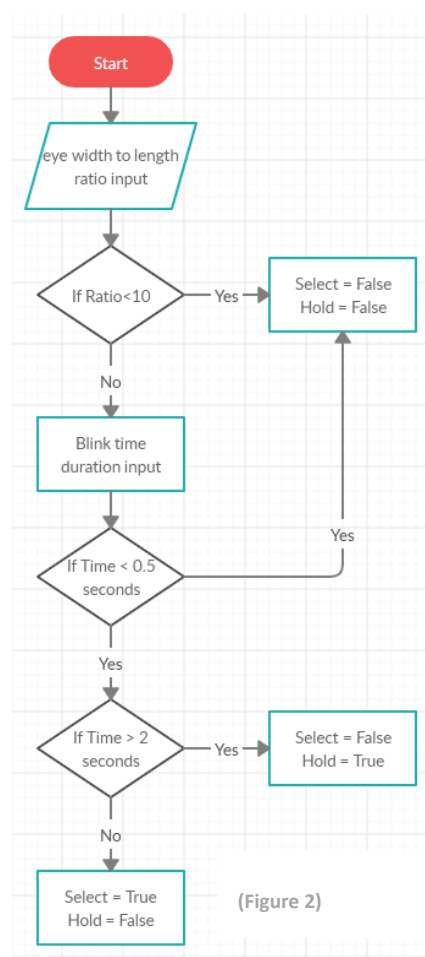
detection has worked correctly (contouring doesn't show any other areas of the face to be circles other than the pupils) and the users face is relatively straight).

Thinking logically:

Logical decisions associated with gaze detection:

Main logical decision involves determining whether the coordinate on the screen is out of range of the screen width/height (figure 1)

If a circle is not detected in frame, the algorithm will assume that the user is blinking and the duration will be calculated until the pupil is detected again – could potentially lead to some accuracy problems.



Logical decisions associated with blink detection:

Main logical decision involves determining the duration of the user's blink and distinguishing whether the user was trying to invoke select, hold, or was just blinking naturally (fig 2)

- For ratio < 10, 10 is just an example limit value, as when the eye is blinking, the eye width is very large in comparison to the length, so the ratio of eye length/eye width should produce a value larger than a limit value, which will be similar to the eye length.

Thinking concurrently:

The initial idea is that the program is to be executed in the following sequence for each frame:

1. Determine eye width/length using facial landmark predictors.
2. Determine blink using eye width to length ratio.
3. If no blink determined, find position of pupil using image contouring.
4. Using eye to screen scale factor, determine the coordinates of the user's gaze on the screen.
5. move the cursor to that position on screen.

Information in each step is required by the following steps, meaning at the moment, I am not aware of any processes that can run concurrently. In addition, the blink detection function must run before every other process that takes place, as the duration of the blink must be determined first and if a blink is detected it eliminates the need to run the gaze detection function removing the time needed to process the image and locate the pupil.

Stakeholder interview

Clients

I have chosen to interview 3 people as stakeholders for my product, the first being a carer (Jos Sullivan - 57), the second being an occupational therapist (Pru Sanchez - 60), and the third being a learning support teacher (Sue Kudlinski). All of these people have assisted those with a mental/physical impairment, therefore have experience in using assistive technology as well as witnessing the benefits and problems that come on account of their clients/students using them. Meaning I should receive a much broader range of feedback, as both stakeholders are very experienced and aware of the needs of their clients/students. As my assistive technology is designed to give those with disabilities more independence and a means to communicate their thoughts, products such as mine would be extremely beneficial to those in caring professions such as Jos and Pru as well as teaching professions like Sue. The reason being, it would be easier for them to understand the needs of clients/students who cannot talk, as well as maybe reducing the level of responsibility in looking after the person, on account of their new ability to communicate and interact with a computer.

Questions

Question 1: Do some of the people you work with/have worked with use an assistive technology to help them interact with other people and/or a computer?

Question 2: If so, what features makes this technology convenient for you/the user?

- Questions 1 and 2 will hopefully give me a more general idea of what sort of generic features – instructions, layout/aesthetic – I'll need to focus on in order to make the software as user friendly as possible.

Question 3: Do you believe eye gaze detection will make interacting with other people and/or a computer easier than using a standard keyboard and mouse, for people with certain mental/physical impairments?

Question 4: What features should be incorporated, to ensure you as well as the user can easily interact/understand the software?

Question 5: What should I avoid – what sort of features would make it difficult for you as well as the user to interact with/understand the software?

- Questions 3, 4 and 5 I've included so I can gather more insight into what specific features are required to make my eye gaze software easy to understand and use. As well as what features may make the software too complicated and inaccessible for both the user and the carer. I think it very important I receive the carers perspective on this, as many of the people who would need to use software like this to communicate don't have the motor ability to set it up and calibrate the software before it can be used. The carer of the user would have to do this; therefore, it is necessary that the software is accessible enough so that any carer can use it.

Question 6: should some features of the software be tailored to people how would just like an easier way to interact with a computer?

Interview 1: Jos (carer)

Question 1: Do some of the people you work with/have worked with use an assistive technology to help them interact with other people and/or a computer?

- *Most of my clients only use technology to assist in communicating with family, via skype or call, but none of my clients require assistive technology to help with mental or physical impairments – so no*

Question 2: If so, what features makes this technology convenient for you/the user?

- No answer as question one's answer was no

Question 3: Do you believe eye gaze detection will make interacting with other people and/or a computer easier than using a standard keyboard and mouse, for people with certain mental/physical impairments?

- *oh absolutely, as even just people getting older sometimes lose the ability to use their hands, so this is a good alternative for them.*

Question 4: What features should be incorporated, to ensure you as well as the user can easily interact/understand the software?

- *Well, it has to be very simple and easy to navigate, so **not too many windows, large buttons with clear labels** so people know what everything is. **Especially the calibration**, it has to quite easy to use and set up, maybe with instructions. There should maybe be a **help page** somewhere, so people who aren't familiar with it can learn how bits work.*
 - o Requirements:
 - Few windows** – to reduce clutter and make the application easier to navigate, max
 - Large buttons with bold labels** – so clearer for first time users of the software to understand what all the buttons do.
 - Step by step instructions with calibration** – to give directions for new users of software on how to calibrate correctly and quickly.
 - Info page** – an information bank containing instructions in using the software correctly - how to select and hold, how to position your head correctly and how to calibrate software. As well as the function of all buttons and displays, to ease confusion for first time users of the software.

Question 5: What should I avoid – what sort of features would make it difficult for you as well as the user to interact with/understand the software?

- *Too much information, like I understand clearly explaining what everything is very important, but maybe **avoid using too much technical language** and data that not everyone may understand. Just write about what the user needs to know, otherwise it gets a bit confusing. Also, I think it's important that the software is **relatively spread out**, as if you're following some instructions and you need to press a button, if everything is cluttered together, the button may be quite hard to locate and it all becomes rather stressful.*
 - Requirements:
 - Avoid overly technical language in info page** – as some users may not have the knowledge to understand the meaning of certain technical words
 - Avoid clutter** – make sure all buttons (other than the keys) are relatively spread out – 1cm apart on the screen perhaps – to avoid overwhelming the user and making it difficult to locate a button.

Question 6: should some features of the software be tailored to people how would just like an easier way to interact with a computer?

- *Definitely, also just because somebody can't communicate doesn't mean their brain isn't functioning properly so I think enabling both people who can't communicate or just elderly people who have a hard time **using a keyboard and mouse to do more than communicate is very important**. So maybe giving them access to things like emails and the internet is a good idea. Maybe you could **adapt it to MS paint** as drawing can be very therapeutic for some people.*
 - Requirements:
 - Ensure keyboard can be used to type on at least 2 other applications on the computer** – the ability to access and type on multiple computer applications without the assistance of another person, would give people with disabilities a greater sense of independence.

Interview 2: Pru (occupational therapist)

Question 1: Do some of the people you work with/have worked with use an assistive technology to help them interact with other people and/or a computer?

- Yes, I do know of someone.

Question 2: If so, what features makes this technology convenient for you/the user?

- *The technology isn't screen based, like yours, but it uses head pressure to help people in wheel chairs communicate. This could perhaps be adapted in way similar to your problem where each letter is represented as a certain number of presses. If anything, **the most useful feature of this technology is that it can be attached to the wheel chair** which the majority of people with tetraplegia need to use. But I understand that would be rather difficult to do for your project.*
 - Limitations:
 - Not accessible to many people with tetraplegia** – as most people with this condition require a wheel chair, a lot of the time commercial eye gaze devices can be attached to the user's wheel chair making their communications device more portable. As my product will be used on a normal desktop computer/laptop, it is less portable and is not designed to be attached to a wheel chair easily.

Question 3: Do you believe eye gaze detection will make interacting with other people and/or a computer easier than using a standard keyboard and mouse, for people with certain mental/physical impairments?

- *Well, the letters have got to be very, very big and also, I think it greatly depends on the level of disability. However, for those who are tetraplegic and can't speak, the **eye is probably the most available sensory organ** one can use to communication, so yes, it will be easier.*
 - o Requirements:
The eye must be used as a means of communication – Through controlling the mouse on a virtual keyboard, as it is the most available organ to those who cannot speak/move
Letters on virtual keyboard must be very large – virtual keyboard should take up at least 1/2 of main window, ensuring there is enough room in the keys to fit large (font size 20 – 24), bold letters. Meaning a user with 20/20 or corrected eyesight can read them easily from a distance of 1m, without having to move their head around to see more easily (may reduce accuracy of gaze location).

Question 4: What features should be incorporated, to ensure you as well as the user can easily interact/understand the software?

- *Well, the keys will definitely have to be larger than the normal keyboard and perhaps making the writing on the keys very big, so it's easy to see. **Avoid clutter** in the main window because that can be quite overwhelming when you first start using a software.*
 - o Requirements:
Avoid clutter – make sure all buttons (other than the keys) are relatively spread out – 1cm apart on the screen perhaps – to avoid overwhelming the user and making it difficult to locate a button.

Question 5: What should I avoid – what sort of features would make it difficult for you as well as the user to interact with/understand the software?

- No answer

Question 6: should some features of the software be tailored to people how would just like an easier way to interact with a computer?

- *I do think that, yes, it's important give people who cannot or struggle to use a keyboard the opportunity to use a computer for things that make life easier – such as **email, internet, shopping**.*
 - o Requirements:
Ensure keyboard can be used to type on at least 2 other applications on the computer –the ability to access and type on multiple computer applications without the assistance of another person, would give people with disabilities a greater sense of independence.

Interview 3: Sue (learning support teacher)

Question 1: Have you ever taught/worked with someone who uses an assistive technology to help them interact with other people and/or a computer?

- *I have worked with a pupil that tried assistive technology. He tried voice recognition software and an eye gaze software. These were used to help the pupil access the written word as he*

was unable to use his hands. Although he could write using his mouth this was quite slow and it was hoped the technology would eliminate the need for a human scribe and give the pupil more independence.

Question 2: Do you believe eye gaze detection will make interacting with other people and/or a computer easier than using a standard keyboard and mouse, for people with certain mental/physical impairments and why?

- *I have only had experience with one pupil. Unfortunately, he did not buy into the idea and consequently he did not invest the time to make the technology work for him due to the **slow and tedious calibration** required before using the software. It became quite tortuous and almost a battle of wills to get him to use it at all. However, I think there is great potential in this technology, particularly for people that have a physical impairment that makes conventional handwriting/typing difficult or impossible. Ultimately it has the potential to open up work place possibilities that would not normally be easily accessible to people with certain disabilities.*
 - o Requirements:
 - Step by step instructions with calibration** – to give directions for new users of software on how to calibrate correctly and quickly.
 - Quick calibration (less than a minute)** – to make the process less of a chore before using the software, to ensure it doesn't frustrate/bore users to the point of giving up on the software.

Question 3: What features should be incorporated, to ensure you as well as the user can easily interact/understand the software?

- *A good, accurate set up facility that is retained. The software needs to be easy to use and **consistently accurate**, after initial set up. **The speed of use was also an issue.** This may have been user reluctance. **Banks of words and common phrases** that are easy to access would be an advantage.*
 - o Limitations:
 - Lack of consistency due to head movement** – The gaze detection accuracy may vary depending on the movement of head, as if the head is moving too much the frame may be unclear, meaning a blink/pupil cannot be detected as easily.
 - o Requirements:
 - Minimize delay of cursor movement** – ensure webcam fps is high enough to minimize lag, so the cursor moves smoothly and in time to the movement of the user's pupil. Can be very frustrating for the user otherwise, as the cursor would be very difficult to control as it would be acting slower than the user.
 - Incorporate common phrase word bank** – would make it easier and faster for users to communicate a very predictable phrase, such as a greeting or agreement.

Question 4: What should I avoid – what sort of features would make it difficult for you as well as the user to interact with/understand the software?

- *For someone with a physical difficulty to plug in bits of extra hardware may not be easy.*
 - o Limitations:

Web cam needs to be plugged in – meaning a user who cannot move will require assistance from another person when plugging in the webcam as well as calibrating the software, hence the software does not provide total independence.

- Requirements:

Only hardware required is a webcam – to make the software as accessible as possible to those who have physical difficulties, by limiting the number of hardware plugins to just the web cam required to track the pupil movement of the user.

Question 5: Should some features of the software be tailored to people who would just like an easier way to interact with a computer?

- *The ideal would be for the software to be able to do all of the functions you mention.*

- Requirements:

Ensure keyboard can be used to type on at least 2 other applications on the computer –the ability to access and type on multiple computer applications without the assistance of another person, would give people with disabilities a greater sense of independence.

Question 6: additional comments?

- *I think the idea of eye gaze is great. In my experience the user has to be wholly behind the concept or they will not persevere with the set up. This then means they do not get proficient at using the software and they give up. (quite quickly). This will obviously depend on the individual and the support they get.*

Research

VR eye tracking



Figure 2

VR headsets use an internal screen placed close to eyes so the virtual 3D display gives the user a perception of depth. As the device is attached to the head, the user's line of sight within the VR display moves with their head. Meaning this technology follows the position of the user's eyes as they move with the head, as opposed to following the gaze of the user's pupil.

Features to consider for my solution

The VR head set is not in any way applicable to my solution as it requires expensive equipment and tracks the position and vergence of the eyes, rather than the position of the pupil. Also using a head set would be a very impractical solution to my problem as the demographic I'll be aiming my application at is people with impaired speech and/or movement. As the head set requires you to move your head in order to explore a virtual environment, the product wouldn't be accessible to those who cannot move their head/neck.

Tobii eye tracking glasses



Figure 1

Comprised of a processing unit, near infra-red lights and a high-resolution camera mounted onto a pair of glasses. The NIR lights shine into the eyes to form a pattern which is then captured by the cameras. The images of these patterns are processed where details within these processed images are used to run a complex 3D eye model algorithm on the processing unit, to determine the gaze of the user.

Features to consider for my solution

The general use of pupil detection shown in the Tobii eye tracker glasses is something I will apply to my solution; however, my solution won't be using the 3D eye model algorithm to estimate the gaze of the user. This is because that method, despite being one of the most accurate, requires a very high-resolution camera as well as multiple NIR lights and other hardware's. For my solution, I'd prefer to use a much less complex algorithm where the only hardware the user requires is somewhat adequate webcam. My product should also be screen based as opposed to head mounted, so a keyboard used for communication can be displayed on the monitor.

Gaze pointer – webcam-based eye tracking software

A software that uses a webcam to detect pupils and translate the pupil positions within a test area on the monitor (obtained through a quick 4-point calibration) to cursor movement within that area of the screen using a scale factor.

$$scale\ factor_x = \frac{width_{screen}}{width_{eye}}$$

$$scale\ factor_y = \frac{height_{screen}}{height_{eye}}$$

As the resolution of the web cam is lower than a normal eye tracker that uses NIR light to enhance the image and make it easier to detect the pupil, filtering is used to determine a clear contour of the pupil in a frame.

Features to consider for my solution

This software is similar to my solution as it is screen based and uses minimal hardware – only requires a webcam. Other features of this product I hope to incorporate into my own is a quick and easy calibration system as well as a relatively high accuracy in estimating the users point of gaze. The gaze pointer can be used for more general tasks that one would usually use a keyboard and mouse for. Therefore, as well as acting solely as a software that aids communication to those who can speak/move, I can now see it is important that the user can also access multiple parts of a computer (text, email, google, etc.) using gaze detection and a virtual keyboard.

Eye gaze edge, Tobii dynavox - Screen based eye tracking device



Figure 3

The user's eyes are tracked on the screen of a computer or tablet, rather than a virtual or surrounding environment, unlike eye tracking VR and glasses. An eye tracker is used, and is placed below the monitor screen, which illuminates the images captured with infrared light so pupil is clearer and easier to detect. an algorithm is used to estimate the location of gaze on the screen using the detected pupil and other details from the processed image. Calibration is usually required, unless algorithms such as the 3D eye model algorithm are used, where calibration can be used to improve accuracy but it is not required.

Features to consider for my solution

My solution will be a screen-based eye tracking software, so is similar to eye gaze edge and dynavox in that way, both devices use an expensive, sensitive infrared camera, whereas I'll be using a web cam in my solution. Also, both devices enable users to access other features of the computer using gaze detection as well as using it as a means of communication. Therefore, the eye gaze edge and dynavox are similar to my solution as their prime purpose is to act as an assistive technology and aid communication. However, I also believe I should give the users of my software the ability to use the gaze detection for other tasks on a computer (texting, email, etc.) rather than just communication, like with the dynavox and eye gaze edge. The eye gaze edge keyboard layout is another feature I would like to incorporate as the use of colored buttons makes them significantly easier to locate for the user, whereas the contrast of grey keys and white letter labels on the keyboard make the keys very easy to read and distinguish, even from a distance.

My solution

New proposed solution

The applications main display will be a virtual qwerty keyboard layout, as it is the most commonly used layout for computers as well as in commercial gaze detection keyboards meaning it would be easier for users to adapt to that layout. Plus, a text input box for the user to communicate through and coloured buttons that link to the info window, the settings window and the phrase bank window. The actual keyboard display will be relatively large ($>1/2$ window size) so the it's easier to estimate which key the user is looking at. To click a key, the user must blink for more that 0.5-2 seconds (this can be changed depending on the user's preferences) whereas to select/hold the user must blink for 2.5-4 seconds (can also be adjusted).

It's very important to incorporate an info window, opened using the info button, which opens a set of clear instructions explaining how to calibrate and use the application, as well as all the available features, as one of the main concerns of the stake holders was the confusion and frustration a technical software may cause without a clear, simply put set of instructions. So, having a clear set of instructions and an info bank is vital in making the software accessible for those using it for the first time.

The user should be able access the internet, email, files, MS paint or other features of a computer that could be operated using the gaze detection application - this will be explained on the instructions within the info button. This is an important feature in all of the stake holders' opinions when they proposed it was fair to give those using the software to access other areas of the computer using the virtual keyboard as an inability to communicate shouldn't mean a user is limited to only using the software for the purpose of communication.

The main keyboard should also include a common phrase bank the main user can access using a button labelled in bold 'PhraseB'. This feature, proposed by Sue Kudlinski, is necessary as one of the main reason's users don't commit to gaze detection software's is the length of time it takes to access the software/communicate using the software. Therefore, providing a common phrase bank could potentially speed up the time it takes for the user say something predictable, such as a greeting. The button should open a window with a list of at least 50 commonly used phrases, from opening the window it should take the user a minimum of 20 seconds to find and select a phrase.

Limitations of current solution

One of the main limitations to my application is accuracy due to the minimal hardware being used (webcam) meaning pupil detection may be less consisted than if we were to use an infrared light that is used in multiple commercial eye gaze technologies such as Tobii dynavox and Tobii eye tracking glasses. Another limitation to my solution it the inability to right/left click, however this could be solved if I find a way to detect winks for each individual eye as opposed to blink detection.

Another potential limitation to my solution is that the display may obscure the rest of the screen if the keyboard is being used when accessing other parts of the computer. Although, if tkinter allows me to make the keyboard display window transparent this may be less of an issue.

The need to plug in the webcam and set up the software initially using a keyboard and mouse is an additional limitation to my software. It may take away from the main user's sense of independence as those who cannot move their hands will need assistance from other people.

Calibration is a required procedure before the software can be used, which could be a potential limitation. As it may become a very tedious process for users of the software, as evidenced through

Sue Kudlinski's experience with similar application, where the student eventually gave up on the software, and sought out an easier alternative.

Another limitation is the lack of consistency in accuracy due to head movement, as a circle as well as the eye length and height may be very hard to determine in a blurred frame due to excessive head movement. In order to maximize accuracy as well as the consistency in accuracy, the user must keep their head still and in the same position, which will be made easier with large keys/buttons that are easy to read and focus on even from a distance (max 1m). Reducing the need for users to move around to see the screen more easily. The feature of large keys has been incorporated into multiple screen-based gaze detection devices such as eye gaze edge, as shown in figure 3.

Another potential limitation is that the software is that it may not be accessible to a lot of people with physical difficulty, as many people who have tetraplegia use wheel chairs, as mentioned by Pru Sanchez. As my software is to be used on a desktop/laptop, it isn't as portable as commercial devices such as VR headsets or eye tracking glasses, or designed to be attached to be attached to wheelchairs, such as Tobii dynavox tablet.

Another potential limitation is the blink duration timings, as the user may sometimes accidentally blink for too long and invoke hold when intending to select a button. This could be corrected by automatically setting cursor to select for buttons that are not designed to be held – keys, buttons – even when the blink duration is within the hold range (2.5-4 second).

Requirements

Hardware/software

Software

- **Python interpreter** – Should be installed to ensure the application, which will be written in python, can run.
- **PyAutoGUI** – GUI automation python module that will enable cursor control using gaze detection.
- **OpenCV-Python** – Computer vision python library that will be used to process and filter each frame of the webcam video input for the pupil/blink detection.
- **NumPy** – python library that is required for OpenCV-Python to run.
- **Windows, Android or Linux OS** – types of OS python can run on.

Hardware

- **Standard external I/O devices** – keyboard and mouse will be needed initially for setting up the application (opening the software and potentially the calibration process). A monitor is also used to show application display.
- **Webcam** – Used to input images (frames) of user, from the video captured by the webcam. Each frame is filtered and processed so the pupil as well as blinks can be detected. The resolution for the webcam should be at least 640x480 for the image filtering to work accurately.

Performance

Stakeholder requirements

- **Quick, easy calibration** – instructions included for each step of calibration to ensure anyone new to the application will easily understand how to set it up, so it can be used soon after opening, as well as work accurately. Especially important as a slow, complex calibration process before using the software may make it difficult and too tedious for user to commit to using the software. Calibration should take less than a minute and include 3 main steps – the calibration (user looks at 5 points on the screen), testing calibration (user uses gaze to point cursor at 5 points on the screen), and finally the option to recalibrate.
- **Incorporate common phrase bank** – would make it easier and faster for users to communicate a very predictable phrase, such as a greeting or agreement. Represented as a large button on the main window, labeled 'PhraseB'. Button should open a small window, to avoid clutter, containing a set of at least 50 common phrases the user may be trying to access. The phrases should be laid out in such a way that from opening the window, it should take the user less than 20 seconds to locate the desired phrase.
- **Minimal cursor movement delay** – important as large cursor delay would be very frustrating for the user, as the cursor would be very difficult to control as it would be moving much slower than the user. The maximum delay should be 0.5 seconds.
- **Info window** - an information bank containing instructions in using the software correctly - how to select and hold, how to position your head correctly and how to calibrate software. As well as the function of all buttons and displays, to ease confusion for first time users of the software. Can be accessed using a large button in the corner of the main virtual keyboard menu, labeled in bold 'Help'.
- **Ensure keyboard can be used to type on at least 2 other applications on the computer** – The ability to access and type on multiple computer applications without the assistance of another person, would give people with disabilities a greater sense of independence. Therefore, the user should be able to, for example, type emails or draw in MS paint using the virtual keyboard and gaze detection.

Research requirements

- **Web cam that records user and translates eye movements to cursor clicks/movements** – as it is the main function of gaze detection products, using eye gaze/blinks to control mouse/virtual keyboard, so user can communicate and navigate a computer.
- **Qwerty virtual keyboard with input box** – used to enable user to type for communication as well as other things such as writing an email or internet browsing. Input box is included in commercial products such as Tobii Dynavox and eye gaze edge as well as most screen-based gaze detection devices, as it provides a place on the screen where the user can express what they are trying to say. Must be relatively large – quarter the size of the main window - or have a scroll option to ensure the user isn't too limited in the amount they can type.
- **Option to change duration of blink for selection and click** – natural blink time may vary for different users so it's necessary that this setting can be changed to fit the user's needs. Duration of blink for selection can be varied between $0.5 < t < 2$ seconds as the average natural blink duration is 0.15 seconds, so 0.5 – 2 is a suitable range for a blink just slightly longer than a natural blink, whereas the hold selection can be varied between $2.5 < t < 4$ seconds.
- **Option to change virtual keyboard window transparency** – so user can change transparency to avoid obscuring the rest of the screen. The virtual keyboard to be used when accessing certain computer applications. Making the screen transparent is a way to compensate for the clutter and limited screen space within the monitor to display both the virtual keyboard and application it is being used for – emails, internet – as opposed to using an external device, such as the Tobii eye gaze glasses, that wouldn't obscure the screen in any way.

- **Settings window** – where user can adjust blink duration and keyboard transparency. Will be accessed by selecting a button labelled 'Options' in bold, on the main virtual keyboard menu. Settings page is incorporated into most screen-based eye gaze technologies to ensure the software is flexible, so is therefore accessible to all users.

Design

Stakeholder requirements

- **Avoid clutter** – make sure all buttons (other than the keys) are relatively spread out – 1cm apart on the screen perhaps – to avoid overwhelming the user and making it difficult to locate a button.
- **Letters on virtual keyboard keys must be very large** – virtual keyboard should take up at least 1/2 of main window, ensuring there is enough room in the keys to fit large (font size 20 – 24), bold letters. Meaning a user with 20/20 or corrected eyesight can read them easily from a distance of 1m, without having to move their head around to see more easily (may reduce accuracy of gaze location). Key color scheme will be white writing with dark gray background, which is the same color scheme to the eye gaze edge keyboard, as the contrast between the letter and background would make it easier for the user to interpret certain letters.
- **Large keys on virtual keyboard** – Virtual keyboard tends to take up the majority (half to two thirds) of the screen for most screen-based gaze detection applications, as provides greater area for user to gaze at for each key, making software more accurate in determining which key the user is looking at.

Research requirements

- **Large buttons with bold, relevant labels** – so clearer for first time users of the software to understand what all the buttons do. On the Tobii dynavox tablet, eye gaze edge and multiple other commercial screen-based gaze detection devices, the buttons surrounding the keyboard are colourful, making them easier to locate for the users.
- **Transparent virtual keyboard menu** – virtual keyboard window with adjustable transparency so the user can use the keyboard to type something on another application, without fully obscuring it.
- **Settings window button** - opened using button labelled 'Settings' in white on main keyboard menu. The button background will be dark, but coloured (green) so it is easy to locate for the user, but also provides a large contrast between the white label, so easy to read.
- **Info window button** - opened using button labelled 'Help' in white on main keyboard menu. The button background will be dark, but coloured (red) so it is easy to locate for the user, but also provides a large contrast between the white label, so easy to read.
- **Phrase bank button** - opened using button labelled 'PhraseB' in white on main keyboard menu. The button background will be dark, but coloured (blue) so it is easy to locate for the user, but also provides a large contrast between the white label, so easy to read.
- **Web cam feed of user** – placed in corner of keyboard window so user can see whether eye is on screen.

Success criteria

Criteria	How to show it's been successfully implemented
Webcam used to accurately control cursor movements	<ul style="list-style-type: none"> • Images of filtered images detecting the pupil/circles in a frame. • The code used to calculate the estimated point of gaze to translate it to mouse movements. • The testing to show whether it's working accurately (<0.5 second delay)
Webcam used to control clicks	<ul style="list-style-type: none"> • Images of the facial landmark predictor being used to output lines across the width and height of the user's eye. • Photo of computer set up, evidencing that a webcam is the only required piece of hardware. • the code used to invoke selection/hold.
User settings window	<ul style="list-style-type: none"> • Screen shot of user settings window. • Screen shot of keyboard window at 3 different transparencies. • The code enabling the user to adjust these settings. • test data proving it works.
User friendly info window	<ul style="list-style-type: none"> • Screen shot of info window, evidence that the window has explanations on how to select/hold, how to position your head correctly and how to calibrate software correctly. • Written evidence from stakeholders confirming that they can understand all language in the info window.
Common phrase word bank	<ul style="list-style-type: none"> • Screenshot of phrase bank window, evidencing it has at least 20 common phrases. • Test data confirming that from opening the phrase bank window, any desired phrase can be located and selected in under 20 seconds, using the gaze detection software.

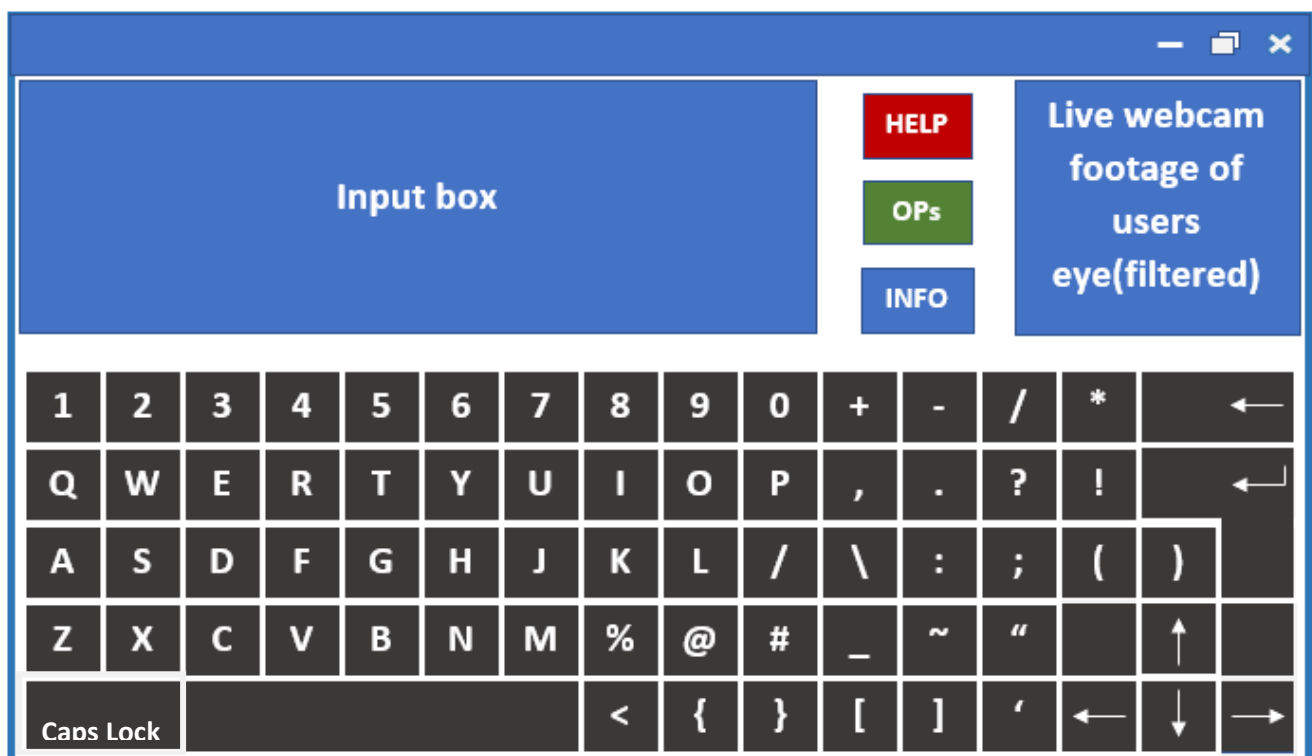
Easy to find buttons	<ul style="list-style-type: none"> • Screen shot of red 'Help' button with white label. • Screen shot of green 'Settings' button with white label. • Screen shot of blue 'PhraseB' button with white label.
Virtual keyboard – large, bold keys	<ul style="list-style-type: none"> • Screen short of virtual keyboard layout showing keys with a grey background and white writing, • Test data to show the target is large enough for each key to accurately estimate which key the user is looking at (correct key selected at least 15 times for 20 tests). • Screen shot of virtual keyboard evidencing that the main keyboard takes up at least ½ of the main window.
Uncluttered layout	<ul style="list-style-type: none"> • Screen shot of virtual keyboard windows evidencing that the buttons (settings, info and phrase), the keyboard, the input box and the web cam feed of the user are at least 1 cm apart from each other on the main window.
Large input box	<ul style="list-style-type: none"> • Screenshot of input box at top of main window (must be at least 1/5 the size of the main window – ensure a lot of space for the user to type). • Code that enables user to type in the input box.
Web cam feed displayed	<ul style="list-style-type: none"> • Screenshot of the camera display in the corner of the virtual key board window (must be at most 1/8 the size of the main window). • the code allowing it to be displayed on main window.
Quick calibration (max 3 step of calibration)	<ul style="list-style-type: none"> • Screen shots of each step of the calibration, the code for the calibration • test data showing difference in accuracy when the software is calibrated correctly and incorrectly (with the user moving around and not facing straight).

	<ul style="list-style-type: none"> test data evidencing that the calibration can be completed in less than a minute.
Instructions with each step of the calibration	<ul style="list-style-type: none"> Screen Shots of instructions at each three steps of the calibration.
Virtual keyboard/gaze detection can be used on 2 other applications	<ul style="list-style-type: none"> Screen shot of gaze detection keyboard being used to type on separate software (email, internet or notepad). Screen shot of gaze detection keyboard being used to draw on MS paint using the hold capacity. Code that enables the keyboard to be used on the separate applications.

Design

Screen designs and features

main user interface:



Features and usability (in relation to success criteria):

The main window design will be a 480X270 resolution screen with provides the user with a view of the webcam feed, a large input box for writing, a qwerty keyboard for typing and the option to view an info

screen, settings menu and a phrase bank. Each element of the window is well spread and uncluttered and the input box fills a large portion of the menu, ensuring the user is not limited in the amount they can type. The inclusion of in relation to the success criteria, this menu ensures:

- Uncluttered layout
- Large input box
- Web cam feed displayed
- Easy to find buttons

Input box:

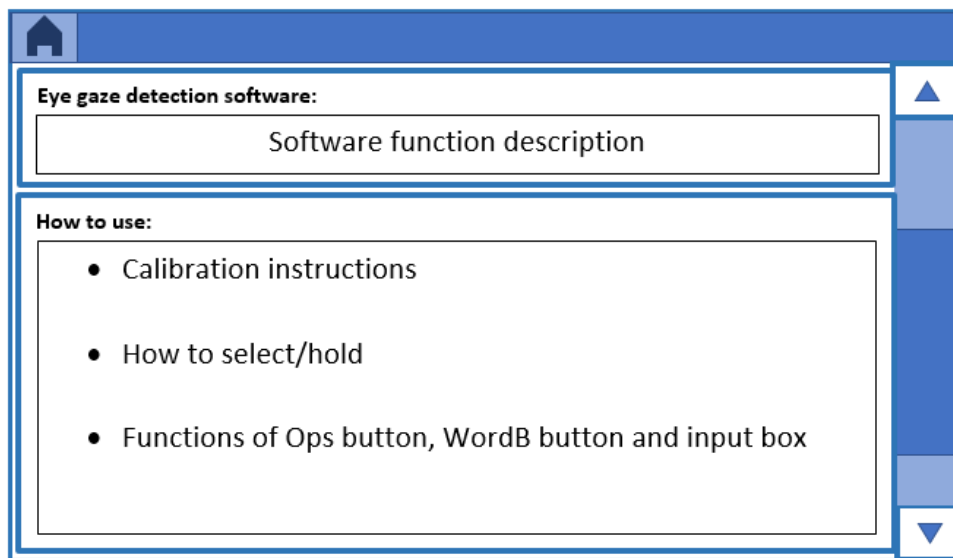
Contains a large area of empty space for the user to type on when clicked in order to communicate, may include a scrolling option along the side so the user can fill more than the space provided.

Relation to success criteria:

- Large input box

Help button:

When clicked will open an info window that contains 3 sections with a bold header, all text being written in a large font (14-18) to ensure all users can read the information:



1. Function of application:
 - Explains the main purpose of the – providing a means of communication for those who cannot speak/type using a keyboard through the use of gaze detection being used on a virtual keyboard.
2. Detailed description on how to calibrate/use application correctly:
 - Explains the calibration process and its purpose as well as how the user should position themselves in order to optimise the accuracy of the gaze detection (be seated comfortably from the point of calibration to limit head movement whilst using the software).
3. The functions of each element of the main menu:
 - Explain how the keyboard, input box, Ops and phraseB buttons and displays do/stand for, as well as the function of the webcam display (there to ensure the users eye is visible to the webcam).

Ops(options) button:

When clicked will open a settings window with three settings the user can alter:

OPTIONS:

SELECT LENGTH: ▲ ▼ Selecting (via blink) the arrows allows users to change the select blink length – in increments of 0.2

HOLD LENGTH: ▲ ▼ Selecting (via blink) the arrows allows users to change the hold blink length – in increments of 0.2

TRANSPARENCY: ▲ ▼ Selecting (via blink) the arrows allows users to change screen transparency – increments of 1

- Blink duration: automatically set to 0.5 seconds but can be altered by increasing the value in the blink detection settings box up to 1.9 seconds (as any longer is in the hold detection range). If value above 1.9 seconds and below 0.5 seconds input into settings box, error message displayed explain why this isn't enabled.
- Hold detection: automatically set to 2 seconds but can be increased to a reasonable limit of 8 seconds by increasing the value in the hold detection settings box. If value above 8 seconds and below 2 seconds input into settings box, error message displayed explain why this isn't enabled.
- Window transparency: has 5 different settings - 5 being fully transparent (invisible), and 1 being fully opaque – can be increased/decreased using arrows on either side of the transparency settings box.

Phrase bank(phraseB) button:

Word Bank:

Phrase 1	Phrase 2	Phrase 3	Phrase 4
Phrase 5			

When clicked will open phrase bank window containing at least 20 common phrases each grouped in to categories of greeting, questions, apologies, thanks to make the window more ordered and easier to navigate.

Webcam feed:

Will display what the webcam can see with the OpenCV filters applied for detecting the pupil, can be used by the user to ensure there is actually within frame of the webcam view as well as whether the pupil detection is working correctly.

Keyboard:

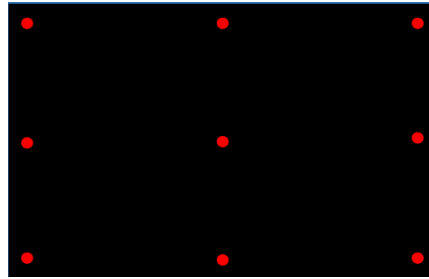
Large keyboard display ($> \frac{1}{2}$ size of the main menu) used by user to type what they would like to communicate using gaze detection. In order to press a key and hold a key the user would have to blink for certain amounts of time to ensure the code can detect the difference between a natural and intentional blink. The alphabet part of the keyboard is in a qwerty layout with the punctuation on the right-hand side and 0 to 9 at the top

Calibration section:**NEXT****process and usability (in relation to success criteria):**

The calibration window will take up the entire screen to ensure the eye gaze detection can be used on areas of the screen other than the application's main user interface. Using a haar cascade and OpenCV image filtering the user's eye corners and width is determined to create a rectangle of the image's main region of interest. The coordinates of the user's pupils within this ROI are recorded and used to predict the position of the user's gaze based on the pupil's position relative to the recorded coordinates for each point. The calibration process is broken into two sections: calibration and testing, ensuring the process is very fast and simple to understand

1. The calibration

- User is asked to look at a red dot whilst remaining still until the GO button changes to NEXT – this repeats 9 times, each dot appearing in different locations ensuring the algorithm has enough reference points.



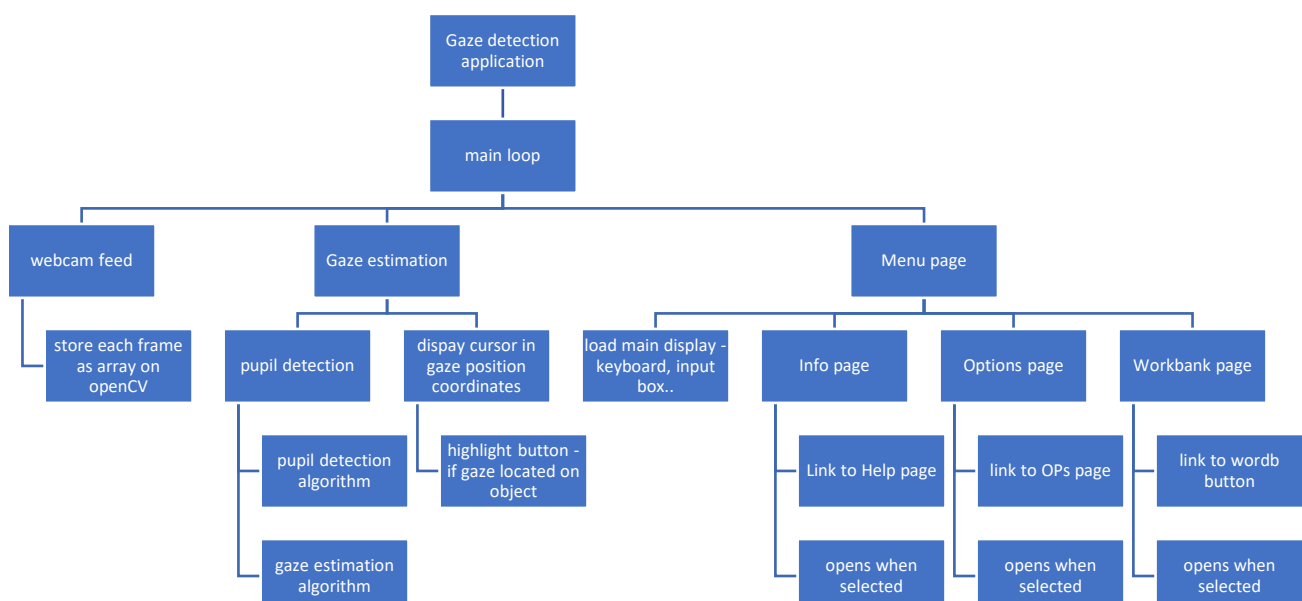
2. The testing

- A blank, black screen with two buttons labelled RECALIBRATE and CONTINUE will appear at the final stage of the calibration process. The users predicted gaze will be a white dot that the user can experiment with to determine whether the gaze detection actually works. If it doesn't, the user can click the recalibrate button and the calibration process starts again. Pressing the continue button results in the main menu window being opened and the calibration window shutting down.

Algorithms:

Main objective process:

1. Import webcam feed
2. Calibrate application – to enable the gaze detection
3. Calculate gaze estimation
4. Apply to application main menu



Program structure:

The software will be comprised of three main programs; calibration program (implemented using OpenCV and Tkinter), gaze estimation program (implemented using OpenCV) and the menu interface program (implemented using OpenCV).

Justification: writing the programs for the three main software components in separate files means the code will be less cluttered with each program involving only one problem, making the code much more readable than if we were to implement all the code in one file. The calibration code will be running first and hopefully only once (unless initial compulsory calibration is incorrect) as certain values need to be calculated at this stage which will be imported to the gaze estimation program from an external file. The calibration GUI code will be implemented in the calibration program to keep the calibration program separate to the main user interface program and the gaze estimation program, as they will be running at the same time, after the calibration is complete.

- **Calibration program Inputs:**
 - 'dlib_68_point_landmark_detector.dat' – imported from an external file.
 - Webcam_feed – stores image data input from webcam.
 - Re_eye_width – user directly inputs the width of their eye.
 - Dist_from_screen – user directly inputs the distance between their face and the camera, and their head must remain roughly in this position.
- **Gaze estimation program inputs:**
 - 'dlib_68_point_landmark_detector.dat' – imported from an external file.
 - Webcam_feed – stores image data input from webcam.
 - Re_eye_width – uploaded from an external file in the same directory as the three main programs and is constant for each user's eye.
 - Focal - uploaded from an external file in the same directory as the three main programs and is constant for each user's webcam.
 - hor_pupil_centre_x, hor_pupil_centre_y - uploaded from an external file in the same directory as the three main programs and is constant for each user.

Main loop:

- Required on both the calibration and gaze estimation programs as will enable all frames captured on the webcam to be processed through iterating through the array of image data using a while loop.

- Every function below, other than the calibration program will be calculated for each frame (item) in the `webcam_feed` array which will be iterated through using a main program while loop. Some variables will be stored outside the fame as they have either been loaded from external files so shouldn't be changed or will require data from multiple frames, including:
 - **Webcam_feed** – is required outside main loop as stores the list of items (video frames) being iterated through and processed.
 - **Re_eye_width** – uploaded from an external file in the same directory as the three main programs and is constant for each user's eye. Used in multiple functions so can be easily accessed throughout the code without risking the float value inside being changed.
 - **Focal** - uploaded from an external file in the same directory as the three main programs and is constant for each user's webcam.
 - **hor_pupil_centre_x, hor_pupil_centre_y** - uploaded from an external file in the same directory as the three main programs and is constant for each user.
 - **(only for calibration) dist_from_screen** – directly input by user.

Webcam feed:

- Inputting webcam feed:
 - **Justification** – necessary as will allow each frame of the video capture to be filtered and analysed, to detect the necessary features.
 - **Variables and structures:**
 - **Webcam_feed**
 - Data type/structure: 2D array storing float values:
 - Use: enables all frame of the video capture to be iterated through using main while loop and processed.
 - **Image**
 - Data type/structure: 1D array storing image data
 - Use: used to store the current frame in the `webcam_feed` array being analysed.
 - **Implementation/algorithm:**

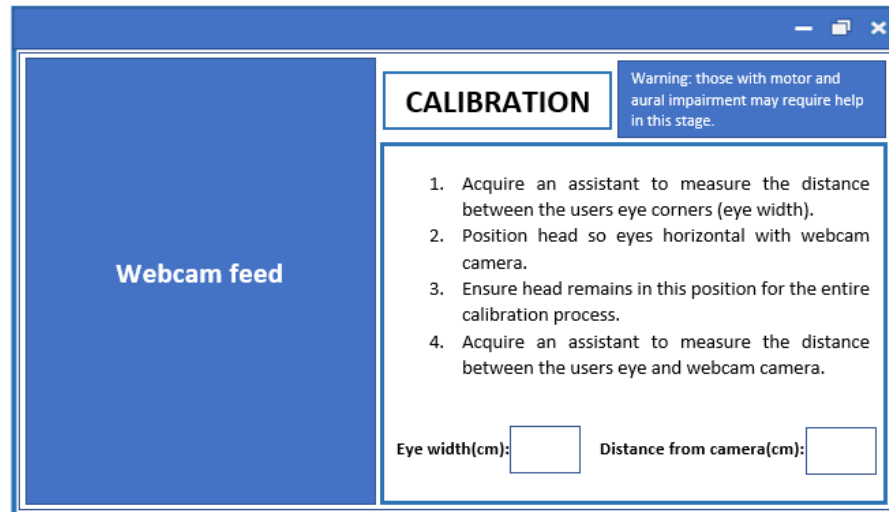
```
webcam_feed = videocapture(0)
while TRUE
    image = webcam_feed.read
```

Calibration program:

- New calibration:
 - **Justification:** originally the calibration would involve finding the scale factors for the horizontal and vertical pupil movement range and the x and y dimensions of the user's screen. This would be tedious for the user as it would mean having to calibrate the screen with every use of the software. As

the eye is spherical, calculating the gaze using a 2-dimensional method wouldn't account for this plane curvature, therefore I have decided to go with the method of abstracting the eyeball to be represented as a sphere. This would limit the number of compulsory calibrations to only one at the very beginning of the program, as well as being more accurate in estimating the gaze direction as the eye's curvature is used to determine it. Will be stored in a separate program, so doesn't need to be run with every use of the separate gaze estimation and menu interface programs.

○ **New screen:**



○ **Variables:**

- **Detector:**

- Data types/structures: 2D array storing float values
- Use: stores and array using Dlib frontal.face.detector() to detect a face within the camera frame and draw a rectangle around it – webcam frame is cropped to the dimensions of this rectangle.

- **Landmarks:**

- Data types/structures: 2D array variable storing float values
- Use: stores the positions of the landmarks (once converted to a NumPy array) - Dlibs 68-point facial landmark detector will be used instead of OpenCV's haar cascades as they are much more accurate and can be used to predict corner points of eyes – only corner coordinates of right eye corners stored.

- **Im_eye_width:**

- Data type/structure: single float value
- Use: a variable storing the width of the eye in the image (corner points of eyes coordinates determined using dib land-mark detection classifier and the distance between them is calculated using $\sqrt{(y1-y2)^2 - (x1-x2)^2}$).

- **Re_eye_width:**

- Data type: single float value

- Use: a variable storing the actual width of a person's right eye in cm – is initially input by user – is used in the calculation for the focal point of the webcam lens. will be stored permanently in a file in the same directory as the program.
- **Dist_from_screen:**
 - Data type: single float value
 - Use: distance from persons eye to camera in cm – is initially input by user - is used in the calculation for the focal point of the webcam lens.
- **Eye_rect:**
 - Data types/structures: 2D array storing coordinates for the bounding position of the right eye
 - Use: frame size can be set to this, meaning less unnecessary contours will be detected.
- **Thresh:**
 - Data types/structures: 2D array storing filtered image data as 1s and 0s.
 - Use: uses cv2.threshold to convert all pixels above a certain threshold to black and all below to white. Included as makes detecting image contours a lot easier.
- **Contours:**
 - Data types/structures: 2D array storing contour path coordinates.
 - Use: found using cv2.findcontours, contours detected can be iterated through and using image processing we can determine which is the pupil
- **Extent:**
 - Data types/structures: calculates and stores a float value that requires the bounding box area of a contour to be calculated, as well as just the contour area.
 - Use: finds the ratio of the contours bounding box to the area of the contours and can be used to discard detected contours that are too large compared to the contours bounding box.
- **Focal:**
 - Data type/structure: single float value – will be stored permanently in a file in the same directory as the program.
 - Use: This stored value can be used to calculate the distance from the eye pupil and the camera when the application is used.
- **hor_pupil_centre_x, hor_pupil_centre_y:**
 - data type/structure: two float values – will be stored permanently in a file in the same directory as the program and will store the x and y values of the pupil when at a horizontal position to the camera. Found using the function cv2.Findcircle or open CVs blob detection algorithm.
 - Use: used to approximate the horizontal distance between the screen and the user is they were to move their head when using the application once calibrated.

- **Subroutines:**

- **Eye_rect_finder function:**

- Function: eye_rect_finder function calculates returns the 2D array of rectangle coordinates around the eye and is set equal to eye_rect variable.
 - Parameters: landmarks between 36 to 42 are the landmark coordinates for the eyes, making them the only required parameters.
 - Implementation/algorithm:

```
#eye_rect_finder
function eye_rect_finder(array)

    top_of_eye = array[n]

    bottom_of_eye = array[m]

    inner_corner = array[u]

    outer_corner = array[v]
    #where n, m, u, v are values between 36 and 42 (eye landmark indexes)

    mp = (top_of_eye[0] + bottom_of_eye[0])/2, (top_of_eye[1] + bottom_of_eye[1])/2
    #finds middle of eye by finding midpoint between the
    #top and bottom of the eye landmarks

    diff = (mp[0] - bottom_of_eye[0]), (mp[1] - bottom_of_eye[1])
    #calculates difference in x and y from midpoint to bottom

    bottom_left = (outer_corner[0] + diff[0]), (outer_corner[1] + diff[0])

    bottom_right = (inner_corner[0] + diff[0]), (inner_corner[1] + diff[0])

    top_right = (inner_corner[0] - diff[0]), (inner_corner[1] - diff[0])

    top_left = (outer_corner[0] - diff[0]), (outer_corner[1] - diff[0])
    #finds rectangle corner coordinates around eye using difference
    #in x and y from midpoint to bottom

    rect = [bottom_left, bottom_right, top_right, top_left]

    eye_width = squareroot((inner_corner[0] + outercorner[0])^2 + (inner_corner[1] + outer_corner[1])^2)
    #calculates distance between eye corners

    return eye_width, rect

endfunction
```

- **Extent_finder function:**

- Function: extend finder function is what returns the calculated float value for extent which is set equal to the extent variable
 - Parameters: The current contour being processes is the only parameter required.

- Implementation/algorithm:

```
#extent_finder
function extent_finder(cont)

    rect = cv2.boundingRect(cont)

    area = cv2.contourArea(cont)

    extent = area/(rect[2] * rect[3])

    return extent

endfunction
```

- **Focal_finder:**

- Function: returns the calculated focal length which is stored in the focal variable which is then permanently stored in an external file.
- Parameters: focal length is calculated by: $((Im_eye_width * Dist_from_screen)/Re_eye_width)$ therefore, Im_eye_width, Dist_from_screen and Re_eye_width all must be parsed into function in order to calculate the focal length.
- Implementation/algorithm:

```
#focal_finder
function focal_finder(rew, dfs, iew)

    focal_length = (iew * dfs)/rew
    #calculates the focal length of the webcam

    return focal_length

endfunction
```

○ Implementation/algorithm:

```

import cv2
import tkinter as tk
webcam_feed = videocapture(0)

function eye_rect_finder(array)
endfunction

function focal_finder(rew, dfs, iew)
endfunction

function extent_finder(cont)
endfunction

function menu()
endfunction

root = tk.Tk() #creates calibration window using tkinter

im = tk.Label(root, image = ImageTK.PhotoImage(image)) #inserts webcam frame into calibraton window

eyewidth_input = tk.Entry(root)#creates input box for eyewidth

re_eye_width = eyewidth_input.get()

screendist_input = tk.Entry(root) #create input box for screen distance

dist_from_screen = screendist_input.get()

focal = []

im_eye_width = []

while re_eye_width <> '' and dist_from_screen <> ''

    image = webcam_feed.read

    detector = dlib.frontal_face_detector()

    landmarks = array(Dlib_68_point_facial_landmark_detector.dat)

    eye_width, eye_rect = eye_rect_finder(landmarks) #function calculates and returns eye width and surrounding rectangle

    im_eye_width.append(eye_width)

    foc = focal_finder(re_eye_width, dis_from_screen, im_eye_width) #function calculates and returns focal point

    focal.append(foc)

    ROI = image[eye_rect[y1]:eye_rect[y2], eye_rect[x1]:eye_rect[x2]] #cropping frame to region of interest

    ROI = cv2.erode(cv2.dilate(cv2.mediumBlur(ROI))) #image processing

```

```

thresh = cv2.threshold(ROI, 37, 255, cv2.Thresh_BINARY)

contours = cv2.findContours(thresh)

for cont in contours

    extent = extent_finder(cont)

    if extent > (unknown val) then

        continue

    if cv2.findcircle(cv2.boundingRect(cont)) is True then #if circle detected in contour area is pupil

        hor_pupil_centre_x, hor_pupil_centre_y = cv2.findcircle(cv2.boundingRect(cont))
        #horizontal pupil centre coordinates

    if len(focal) > (unknown val) then #breaks loop after certain number of iterations
        break
endwhile
vals_to_move = [[sum(focal)/len(focal)], [sum(im_eye_width)/len(im_eye_width)], [hor_pupil_centre_x, hor_pupil_centre_y], dist_from_screen]
#i realise i need to use the distance from screen in the pupil detection function in order to determine the z val of pupil
file = open('file.txt', 'w')
for i in vals_to_move:
    file.write(i + '\n') #writes focal, eyewidth and horizontal pupil x y coords
file.close()

menu() #closes calibration window, opens menu window

```

Gaze estimation program:

- New gaze estimation algorithm:
 - **Justification** – is required as is what enables the user to move the cursor with their eyes. Will be different to account for the easier and more accurate calibration, meaning scale factors won't be used and a 3d model of the eye will be worked with instead. This is beneficial as it means the estimation of gaze will be more accurate as the curvature of the eye is accounted for.
 - **Variables and structures:**
 - Pupil detection algorithm:
 - **Detector:**
 - Data types/structures: 2D array storing float values
 - Use: stores and array using Dlib frontal.face.detector() to detect a face within the camera frame and draw a rectangle around it – webcam frame is cropped to the dimensions of this rectangle.
 - **Landmarks:**
 - Data types/structures: 2D array variable storing float values
 - Use: stores the positions of the landmarks (once converted to a NumPy array) - Dlibs 68-point facial landmark detector will be used instead of OpenCV's haar cascades as they are much more accurate and can be used to predict corner points of eyes – only corner coordinates of right eye corners stored.

- **Eye_rect:**
 - Data types/structures: 2D array storing coordinates for the bounding position of the right eye
 - Use: frame size can be set to this, meaning less unnecessary contours will be detected.
- **Thresh:**
 - Data types/structures: 2D array storing filtered image data as 1s and 0s.
 - Use: uses cv2.threshold to convert all pixels above a certain threshold to black and all below to white. Included as makes detecting image contours a lot easier.
- **Contours:**
 - Data types/structures: 2D array storing contour path coordinates.
 - Use: found using cv2.findcontours, contours detected can be iterated through and using image processing we can determine which is the pupil.
- **Contour_area:**
 - Data types/structures: float value for the detected contour area found using cv2.contourarea.
 - Use: used to discard detected contours that are too small, e.g.: if the contour_area < 8 → discard
- **Extent:**
 - Data types/structures: calculates and stores a float value that requires the bounding box area of a contour to be calculated, as well as just the contour area.
 - Use: finds the ratio of the contours bounding box to the area of the contours and can be used to discard detected contours that are too large compared to the contours bounding box.
- **Pupil:**
 - Data types/structures: array storing float values
 - Use: detected using OpenCV greyscale filter and gaussian filter to reduce noise of each frame and ensure the contours are easier to determine. OpenCV findcontours() and then houghcircle() function is used to detect circles in the frame and the most pronounced circle detected will be the pupil. Stores the radius of circle as well as its x, y positions in a list which will be used to calculate a value for z.
- **Gaze estimation algorithm:**
- **Focal:**
 - Data types/structures: single float value
 - Use: will be loaded from file 'focal_value.txt' and will be used to calculate the distance from the eye of the user to the camera.
- **Re_eye_width:**
 - Data type/structures: single float value

- Use: will be loaded from file 'focal_value.txt' and will be used to calculate the distance from the eye of the user to the camera.
 - **Im_eye_width:**
 - Data types/structures: single float value
 - Use: a variable storing the width of the eye in the image (corner points of eyes coordinates determined using dib land-mark detection classifier and the distance between them is calculated using $\sqrt{(y1-y2)^2 - (x1-x2)^2}$)
 - **Screen_dist:**
 - Data types/structures: single float value
 - Use: stores the float value for the distance between the camera and the pupil of the eye. This will be used to calculate the horizontal distance between the users eye and screen.
 - **Screen_z:**
 - Data types/structures: single float value
 - Use: approximates the horizontal distance to screen – this is important as we can abstract the monitor to be a flat plane on the z axis where $z = \text{screen_z}$.
 - **Pupil_z:**
 - Data types/structures: single float value
 - Use: stores the z coordinates of the pupil, relative to the eye sphere using the detected x and y coordinates of the pupil in the $(x+a)^2 + (y+b)^2 + (z+c)^2 = 1/2\text{eye_width}$
 - **Centre_to_pupil:**
 - Data types/structures: 1D array storing line equations
 - Use: stores the equations $[[x + xt], [y + yt], [z + zt]]$ where the z equation is used to calculate a value of t which can be used to calculate the x and y coordinates on screen.
 - **T_var:**
 - Data types/structures: single float value
 - Use: contains value of t calculated by setting $z + zt = \text{screen_z}$ where $t = (\text{screen_z} - z)/z$. This value will be later used to calculate values of x and y on the screen plane.
 - **X_screen, y_screen:**
 - Data types/structures: two float values
 - Use: stores the x and y coordinates of the intersection of estimated gaze direction and screen plane, therefore cursor can be positioned here as it is the estimated gaze position.
- **Subroutines:**
 - **Eye_rect_finder function:**
 - Function: eye_rect_finder function calculates returns the 2D array of rectangle coordinates around the eye and is set equal to eye_rect variable.

- Parameters: landmarks between 36 to 42 are the landmark coordinates for the eyes, making them the only required parameters.
- Implementation/algorithm:

```
#eye_rect_finder
function eye_rect_finder(array)

    top_of_eye = array[n]

    bottom_of_eye = array[m]

    inner_corner = array[u]

    outer_corner = array[v]
    #where n, m, u, v are values between 36 and 42 (eye landmark indexes)

    mp = (top_of_eye[0] + bottom_of_eye[0])/2, (top_of_eye[1] + bottom_of_eye[1])/2
    #finds middle of eye by finding midpoint between the
    #top and bottom of the eye landmarks

    diff = (mp[0] - bottom_of_eye[0]), (mp[1] - bottom_of_eye[1])
    #calculates difference in x and y from midpoint to bottom

    bottom_left = (outer_corner[0] + diff[0]), (outer_corner[1] + diff[0])

    bottom_right = (inner_corner[0] + diff[0]), (inner_corner[1] + diff[0])

    top_right = (inner_corner[0] - diff[0]), (inner_corner[1] - diff[0])

    top_left = (outer_corner[0] - diff[0]), (outer_corner[1] - diff[0])
    #finds rectangle corner coordinates around eye using difference
    #in x and y from midpoint to bottom

    rect = [bottom_left, bottom_right, top_right, top_left]

    eye_width = squareroot((inner_corner[0] + outer_corner[0])^2 + (inner_corner[1] + outer_corner[1])^2)
    #calculates distance between eye corners

    return eye_width, rect

endfunction
```

- Extent_finder function:**

- Function: extent finder function is what returns the calculated float value for extent which is set equal to the extent variable.
- Parameters: The current contour being processed is the only parameter required.
- Implementation/algorithm:

```
#extent_finder
function extent_finder(cont)

    rect = cv2.boundingRect(cont)

    area = cv2.contourArea(cont)

    extent = area / (rect[2] * rect[3])

    return extent

endfunction
```

- Screen_dist_finder function:**

- Function: returns float value for the distance between the user's eye and the screen, which is set equal to the screen_dist variable.

- Parameters: calculated using $\text{Calc_screen_dist} = ((\text{Focal} * \text{re_eye_width}) / \text{Im_eye_width})$ therefore focal, re_eye_width and im_eye_width must be parsed.
- Implementation/algorithm:

```
#screen_dist_finder
function screen_dist_finder(rew, f, iew)

    screen_dist = (f * rew)/iew #finds distance from eye to camera

    return screen_dist
```

- **Screen_z_finder function:**

- Function: returns float value for the approximate horizontal distance to the screen, which is set equal to screen_z variable.
- Parameters: is calculated using $\text{screen_z} = \sqrt{(\text{screen_dist}^2 - X^2) - Y^2}$ therefore requires screen_distance value to be parsed. Additionally, hor_pupil_centre_x and hor_pupil_centre_y must be parsed along with the pupils current x, y coordinates. X and Y in the equation equal the distance between these two coordinates.
- Implementation/algorithm:

```
#screen_z_finder
function screen_z_finder(screendist, x, y, horx, hory)

    diffx, diffy = (horx - x), (hory - y)

    z = squareroot((screendist^2 - x^2) - y^2)

    return z

endfunction
```

- **Pupil_z function:**

- Function: returns float value for the z coordinate of the pupil on the eyeball sphere so it can be used to find line equation to represent gaze direction. Is set equal to variable Pupil_z.
- Parameters: is calculated using $z = \text{absolute}(\text{squareroot}((1/2\text{eye_width} - (x+a)^2 - (y+b)^2)) - c)$. Therefore, requires parameters eye_width, x, y, a, b, c.

- Implementation/algorithm:

```
#pupil_z
function pupil_z(eyewidth, x, y, horx, hory, ogz, z)

    a = horx - x

    b = hory - y

    c = ogz - z
    #above finds difference in x, y, and z to originonal horizontal coordinates determined in calibration..

    z = absolute(squareroot(0.5*eyewidth - (x + a)^2 - (y + b)^2)) - c
    #finds z coordinates of pupil on sphere

    return z

end function
```

- **Centre_to_pupil_func function:**

- Function: creates a set of 3 equations in the form $[[x + xt], [y + yt], [z + zt]]$ and stored in an array for each axis, which can be used to find the intersection for the screen. Will be set equal to variable centre_to_pupil.
- Parameters: values x, y and z will equal values stored in variables pupil[x], pupil[y] and pupil_z respectively. Therefore, these values will be parsed into the function.
- Implementation/algorithm:

```
#centre_to_pupil
function centre_to_pupil_func(x, y, pupil_z)

    x_par = x + xt

    y_par = y + yt

    z_par = z + zt
    #creates equation of line for each axis to form the 3D line from pupil to screen.

    return [x_par, y_par, z_par]

endfunction
```

- **Implementation/algorithm:**

- Pupil detection:

```
#pupil detection
import cv2

function eye_rect_finder(array)
endfunction

function focal_finder(rew, dfs, iew)
endfunction

function extent_finder(cont)
endfunction

function screen_dist_finder(rew, f, iew)
endfunction

function screen_z_finder(x, y, horx, hory)
endfunction

function pupil_z(eyewidth, x, y, horx, hory, ogz, z)
endfunction

function centre_to_pupil(x, y, z)
endfunction

webcam_feed = videocapture(0)
while TRUE

    image = webcam_feed.read

    detector = dlib.frontal_face_detector()

    landmarks = array(Dlib_68_point_facial_landmark_detector.dat)

    eye_width, eye_rect = eye_rect_finder(landmarks) #function calculates and returns eye width and surrounding rectangle

    ROI = image[eye_rect[y1]:eye_rect[y2], eye_rect[x1]:eye_rect[x2]] #cropping frame to region of interest

    ROI = cv2.erode(cv2.dilate(cv2.medianBlur(ROI))) #image processing

    thresh = cv2.threshold(ROI, 37, 255, cv2.Thresh_BINARY)

    contours = cv2.findContours(thresh)

    for cont in contours

        extent = extent_finder(cont)

        if extent > (unknown val) then

            continue

        if cv2.findcircle(cv2.boundingRect(cont)) is True then #if circle detected in contour area is pupil

            x, y = cv2.findcircle(cv2.boundingRect(cont))
```

- Gaze estimation:

```
#gaze detection
vals = [i.rstrip('\n') for i in open('file.txt', 'r')] #focal length and image eye width
#along with the horizontal x y coordinates and the original horizontal distance to screen

focal = vals[0]

re_eye_width = vals[1]

im_eye_width = eye_width

screen_dist = screen_dist_finder(re_eye_width, focal, im_eye_width) #find distance to screen using focal point
screen_z = screen_z_finder(x, y, vals[2]) #find horizontal distance to the screen

og_dist= val[3] #original distance from screen (in calibraton)

pupil_z = pupil_z(im_eye_width, x, y, vals[2], og_dist, screen_z) #funciton returns pupils z coordinate
centre_to_pupil = centre_to_pupil_func(x, y, pupil_z) #functions returns line equation from eye centre to pupil
T_var = (screen_z - pupil_z)/pupil_z #calculates t value for 3D parametric
x_screen, y_screen = (y + y*T_var), (x + x*T_var) #calculates screen intercept using t value
```

Menu display:

- User interface:
 - **Justification:** would provide a way for the user of the program to interact with and apply the gaze detection function through an interface containing buttons, and input boxes. The menus will be uncluttered and contain buttons which will be linked to separate windows (info/option/wordbank buttons):
 - **New button:** I have chosen to add a new 'Cal' for calibration button into the main menu that will enable the user to redo the calibration completed before using the program. This means if the calibration is correct the user won't need to use the button and the 3D eye model means calibration isn't necessary. However, if the initial calibration is incorrect, the user still has the option to redo it.
 - **Variables and structures:**
 - **Alph:**
 - Data types/structures: a 1D array storing these characters:
 - Letters: q w e r t y u i o p a s d f g h j k l z x c v b n m
 - Numbers: 1 2 3 4 5 6 7 8 9 0
 - Punctuation: + - * , . ? ! / \ ; : () % @ # _ ~ " ' < > & £ [] '
 - Commands: caps-lock, backspace, spacebar
 - Use: can be iterated through to instantiate buttons for every item in the list.
 - **Buttons:**
 - Data types/structures: a 1D array storing buttons for every value in Alph.
 - Use: stores and displays all the buttons on the keyboard whilst binding them all to the write function, which enables the user to type in the input box.

- **Input_box:**
 - Data types/structures: stores a text widget, created using Text()
 - Use: acts as the input box, dimensions parameters inputs will ensure it takes up a quarter of the main window. It will be attached to a scrollbar so more can be written in the given space.
- **scroll_bar:**
 - Data types/structures: stores a scroll bar widget, created using Scrollbar()
 - Use: enables user to view all text written into the input_box, even if the amount written is larger than the given space.
- **Image:**
 - Data types/structures: stores an array of image data from the gaze detection program.
 - Use: input into the corner of the main menu screen so the user can see if they're in frame. Also used to determine whether to enact selection and hold commands.
- **info:**
 - Data types/structures: stores a button object on menu window.
 - Use: Button command will be linked to the info window function that evokes the opening of the info window and the closing of the main menu window, to avoid clutter. 'INFO' will be displayed in bold white on the button which will have a blue background.
- **Settings:**
 - Data types/structures: stores a button object on menu window.
 - Use: Button command will be linked to the settings window function that evokes the opening of the settings window and the closing of the main menu window, to avoid clutter. 'OPS' will be displayed in bold white on the button which will have a green background.
 - Widgets in settings window:
 - **Transparency_input:** entry widget for the transparency arrow buttons to vary the transparency of all windows in the software.
 - **Transparency_buttons:** array of two button objects next to the transparency text widget that when clicked invoke the value in transparency_input to increase or decrease by 0.2 to vary the window transparency using root.attributes('-alpha',) function (can only vary between 0 and 1). Linked to transparency_write function.
 - **Hold_input:** entry widget for the hold length arrow buttons to vary the blink length to evoke the hold command.
 - **Hold_buttons:** array of two button objects next to the hold text widget that when clicked invoke the value in hold_input to increase or decrease by 0.2 to vary the float value stored in the hold_length variable. Linked to hold_write function.
 - **Click_input:** entry widget for the click length arrow buttons to vary the blink length to evoke the select command.
 - **Click_buttons:** array of two button objects next to the click text widget that when clicked invoke the value in click_input

to increase or decrease by 0.2 to vary the float value stored in the click_length variable. Linked to click_write function.

- **Click_length:**
 - Data types/structure: global float value
 - Use: changed in the settings_window and click_write function and is used to vary the blink time duration to allow a click/selection command.
- **hold_length:**
 - Data types/structure: global float value
 - Use: changed in the settings_window and hold_write function and is used to vary the blink time duration to allow a hold command.
- **transparency:**
 - Data types/structure: global float value
 - Use: changed in the settings_window and transparency_write function and is used to vary the transparency of the current window open.
- **Wordbank:**
 - Data types/structures: stores a button object on menu window.
 - Use: Button command will be linked to the Wordbank window function that evokes the opening of the Wordbank window and the closing of the main menu window, to avoid clutter. 'WORDB' will be displayed in bold white on the button which will have a red background.
 - Widgets in wordbank window:
 - **Phrase_buttons_names:** a 1D array of strings storing 50 commonly used phrases. Will be iterated through to instantiate all buttons to be stored in the window.
 - **Phrase_buttons:** a 1D array of button objects. Each button contains a grid location and is bound to a function that writes the common phrase stored on the button in the main menu input box, when a button is pressed. Also linked to main Write function.
- **Menu:**
 - Data types/structures: stores 3 identical button objects in array.
 - Use: each button stored in the top corner of the info, settings and wordbank window and is linked to the menu window function which evokes the closing of the current window and the opening of the menu window.
- **Calibration:**
 - Data types/structures: stored button object on menu window.
 - Use: Button command will be linked to the calibration window function that evokes the opening of the Wordbank window and the closing of the main menu window, to avoid clutter. 'WORDB' will be displayed in bold white on the button which will have a red background.
 - Widgets in calibration window:

- **eyewidth_input:** entry widget for the user to type the width of the user's eye (using a keyboard and mouse).
- **screendist_input:** entry widget for the user to type the distance between the webcam and the users face (using a keyboard and mouse).

○ Subroutines:

• Write:

- Procedure: bound to the buttons stored in phrase_buttons and buttons arrays – enable a particular phrase, action or character to be input into the input_box text widget on the main menu when a button is clicked.
- Parameters: the button_name (a value stored in the alph array: button_name = alph[i]) must be input to the write function so it can evoke the appropriate action for the button being parsed.
- Implementation/algorithm:

```
#write function
function write(out)

    if out = 'caps-lock' then

        caps = not caps #changes letter case

    endif

    if out = 'backspace' then

        input_box.delete('end-2c') #removes current and last element stored in input_box widget

    endif

    else:
        if caps == TRUE then

            out.capitalize()

        endif

        input_box.insert(out) #inserts out into input_box

    endif
endfunction
```

• Info_window:

- Procedure: bound to info button which when clicked evokes the closing of the menu window and the opening of the info window.
- Parameters: only has the info button using this function so requires no parameters.
- Implementation/algorithm:

```
#info_window
function info_window(arg)

    quit(root) #quits menu window

    info = tk.Tk() #opens info window

    info.attributes('-alpha', transparency)

    #info text imported to window

    endif

endfunction
```

• Settings_window:

- Procedure: bound to settings button which when clicked evokes the closing of the menu window and the opening of the settings window.
- Parameters: only has the settings button using this function so requires no parameters.
- Implementation/algorithm:

```
#settings_window
function Settings_window()

    quit(root) #quits menu window

    settings = tk.Tk() #opens new settings window

    settings.attributes('-alpha', transparency)

    transparency_input = entry(settings, width, height)

    hold_input = entry(settings, width, height)

    click_input = entry(settings, width, height)
    #creating input boxes for all variable values in the settings menu

    transparency_button_inc = button(settings, command = transparency_write(0))
    transparency_button_dec = button(settings, command = transparency_write(1))
    hold_button_inc = button(settings, command = hold_write(0))
    hold_button_dec = button(settings, command = hold_write(1))
    click_button_inc = button(settings, command = click_write(0))
    click_button_dec = button(settings, command = click_write(1))
    #button objects created to enable the user to increase and decrease values in setting

    menu = button(settings, command = menu(settings) #button to close settings window and open menu

    endif
endfunction
```

- **Transparency_write, hold_write, click_write:**
 - Procedure: each bound respectively to transparency_buttons, holdlength_buttons, clicklength_buttons. Function evokes the increase or decrease of the value stored in transparency_input, hold_input, and click_input by 0.2.
 - Parameters: need one parameter (1 or 0) to distinguish as to whether the button is increasing or decreasing the value.

- Implementation (transparency_write):

```
#transparency_write
function transparency_write(i, win)

    if i == 0 then # checks if increasing or decreasing transparency

        if transparency_input + 0.2 > 1 then

            transparency_input = transparency_input

        endif

        else then

            transparency_input = transparency_input + 0.2 # increment transparency by 0.2 between 0 and 1

        endif

    endif

    if i == 1 then

        if transparency_input - 0.2 < 0 then

            transparency_input = transparency_input

        endif

        else then

            transparency_input = transparency_input - 0.2

        endif

    endif

    transparency = transparency_input #defines current window transparency

endfunction
```

- Implementation(hold_write/click_write)

```
#hold function - works exactly same way for click_write function
#difference being hold_input would be click_input, hold_length would be click_length
#and the range for blink is 0.5 to 1.9 seconds, compared to holds 2 to 8 seconds
function hold_write(i)
    if i == 0 then

        if hold_input + 0.2 > 8 then

            hold_input = hold_input

        endif

        else then

            hold_input = hold_input + 0.2 # increment hold length by 0.2 between 2 and 8

        endif

    endif

    if i == 1 then

        if hold_input - 0.2 < 2 then

            hold_input = hold_input

        endif

    endif
```

```

        else then
            hold_input = hold_input - 0.2
        endif
    hold_length = hold_input #defines current blink hold length
end
endif
endfunction

```

- **WordBank_window:**

- Procedure: bound to wordbank button which when clicked evokes the closing of the menu window and the opening of the wordbank window.
- Parameters: only has the wordbank button using this function so requires no parameters.
- Implementation/algorithm:

```

#wordbank
function wordbank()
    quit(root) #closes menu window

    wordbank = tk.Tk() #opens new wordbank window
    wordbank.attributes('-alpha', transparency)

    phrase_buttons_names = [' hello there! ', ' how are you ', ' whats your name? ' .... ' thank you '] #array of common phrases
    phrase_buttons = []

    for i in phrase_buttons_names
        phrase_buttons.append(button(wordbank, text = i, command = write(i)) #makes array of buttons for phrases
    end

endfunction

```

- **Menu_window:**

- Procedure: bound to menu buttons which when clicked evokes the closing of the current window and the opening of the main menu window.
- Parameters: requires the index of the button in the menu array to be parsed in order to determine which window needs to be closed before opening the menu window as well as the root variable.
- Implementation/algorithm:

```

#menu_window
function menu_window(i)
    quit(i) #closes old window

    open(window) #open root menu window
endfunction

```

○ Implementation:

```

function wordBank_window()
endfunction

function menu()
endfunction

function calibration_window()
endfunction

global click_length = 0.5

global hold_length = 2
|
global transparency = 1

root = tk.Tk() # creates main menu window

root.attributes('-alpha', transparency)

input_box = Text(root, height, width) # creates text widget for key buttons and phrases

scroll_bar = Scrollbar(input_box) # attaches scroll bar to input box

alph = ["q", "w", "e", "r", "t", "y", ..., "n", "m", "l", "2", "3", ...,
        "g", "0", "+", "-", "=", "", ..., "f", "[", "]", "'", "caps-lock",
        "backspace", "spacebar"]

buttons = []

cap = FALSE

for i in alph:
    if i == 'spacebar' then
        inp = ' '
    endif
    else then
        inp = i
    endif
    buttons.append(button(root, text = i,
                        command = write(inp))) # creates list of keyboard buttons

image = tk.Label(root, image = ImageTK.PhotoImage(im))

info = button(root, text = 'INFO', command = info_window())

settings = button(root, text = 'OPs', command = settings_window())

wordbank = button(root, text = 'WORDB', command = wordbank_window())

calibration = button(root, text = 'CAL', command = execute(calibration_program))
#defining all window buttons

```

Testing method:

The program will be tested as it is being developed to ensure every part of the program works as we build on the program. at least 3 cycles of development will be completed each building on the previous prototype and each developing different functions within the program.

Iterative testing approach:

As stated above, the program will be broken down into functions that will be iterated through and developed and tested separately. This means at the evaluation/final testing stage, many aspects of the code have already been debugged, tested and analysed separately, so final test should therefore run a lot smoother with fewer errors.

Final testing:

This will be carried out post development, to determine whether the program meets the requirements of the success criteria. Various part of the program such as the accuracy of the pupil/gaze detection and calibration or the functionality of the input box and buttons, will all be tested separately using valid, borderline and invalid data as inputs, to determine the robustness of the program as a whole.

Test checklist:

Each of these parts of my program will be tested individually to ensure every part of the success criteria is met and conclude the development stage:

To test:	Works:
Cycle 1: Pupil detection:	
Pupil detection program can detect pupils	
Pupil detection program can detect clicks and hold commands from blinks	
Cycle 2: Gaze estimation:	
Gaze estimation program enables the cursor to be controlled by pupil movements	
Gaze estimation program accesses calibration data from external file and uses them correctly	

Cycle 3: windows/layout:	
Correct formatting for each window	
Calibration window opens when program opened	
Webcam feed and instructions present in calibration window	
Input box can be written into on calibration window and inputs are stored in external file	
Menu screen opened when both input boxes in calibration window are filled	
Blinking for set click time enables a button to be clicked	
Blinking for set hold time enables a button to be held down and released	
'HELP' button opens info window	
'OPs' button opens settings window	
Arrow buttons on settings window alter value in text box	
Error message displayed in front of settings window when values in text box go beyond or below boundary values	
'WORDB' button opens word bank window	
Hold function can be used on vertical scroll bar to move down a window	
When phrase button pressed in word bank window, menu window opened and phrase selected is displayed in input box	
'CAL' button opens calibration window	
Home button opens main menu	
Character buttons can be used to input text to input box	

Webcam feed displayed on menu screen	
'Cap lock' button capitalizes all text being entered into input box	

If I succeed in meeting every condition on this list, all criteria on the success criteria will be met and that will conclude the development. The program will then be passed onto the final testing phase and one of the stake holders where we will be taking in their feedback and evaluate the success of the program

Development/testing cycles:

On the list above, the testing has been divided into 3 different cycles involving 3 different aspects of the program: the pupil detection, gaze detection and windows/layout. I have divided the development cycle up into these three iterations as they each provide a different feature and element of functionality to the program so should therefore be tested and tried out separately rather than all at once, to see if each feature works as the program is developed. They will also be developed and tested in the order of pupil detection, then gaze detection and then window/interface as gaze detection required the pupil to be detected and the menu interface cannot be accurately designed fully tested if the gaze estimation isn't working (because if we were using a keyboard and mouse to test the design and functionality of the windows, it wouldn't be true to the accuracy or inaccuracy of the gaze detection cursor movements).

Pupil detection accuracy:

No conditions on the success criteria are specifically related to pupil detection however there are criteria related to gaze estimation functionality and accuracy, which requires the pupil detection algorithm to work in at least one environment (lighting, head position/distance from camera).

Once the pupil detection code has been written out, the main thing to test is whether different factors could influence the accuracy of the pupil detection potentially including:

- Room lighting
- Head position relative to camera
- Eye colour
- Pupil position (would be investigated when the optimal head position and room lighting is determined to see how pupil position at that position and lighting effect the accuracy)

We will be determining 'accuracy' of the pupil detection algorithm at different lightings, head positions/distances, eye colours and pupil positions in order to get an idea of the best environment and position the user should be in to enhance the programs accuracy. As well as giving us an idea of how well the program adapts to different settings and whether it will provide the expected output for erroneous and borderline data.

- An example of an erroneous input for pupil detection might include an image where no face or eye is detected, or an image where more than a limit number of circles is detected.

In order to show the result of the testing under different conditions, I could use my webcam to take screenshots of my face in the different settings with a green circle used to represent the detected circles/pupil.

This is an effective way of determining the accuracy of the code as the quality of the webcam is average so will act as a better indicator of the actual accuracy of the program than if I was using a higher quality image. I will also have the freedom to move around and adjust the screenshots lighting using image editing.

Gaze detection accuracy:

The main condition to be met from the success criteria involving the gaze detection aspect of the program is:

- Webcam used to accurately control cursor movement

The method of testing this features functionality and accuracy will be similar to the pupil detection testing as through using screenshots of a screen the feature is being used on. One way we could test the accuracy of the gaze estimation is using it on separate window containing multiple dots for me, the user, to look at along with a dot of a different colour representing the actual predicted gaze location. The average distance between the dots the user is supposed to be looking at and the predicted gaze location for each dot will be used to determine the accuracy in the predicted gaze location. The delay will also be tested through measuring the difference in time between the user adverting their gaze between two dots and the actual time it takes for the cursor to travel between the two locations in order for the success criteria to be met, the delay must be less than 0.5 seconds. Erroneous data may involve multiple eyes controlling the cursor at once, as the software is supposed to be designed for one user, therefore multiple controllers should be designed to throw an error.

I believe this method of testing the gaze detection will be effective as it both enables us to test both whether the gaze detection even works as well as its accuracy (delay time).

Interface testing:

The main criteria to test in terms of the window/menu layout is the functionality of:

- The buttons:
 - The buttons must be tested to ensure they perform the function they are supposed to and can work without crashing. During development the buttons will be tested as they are coded on Tkinter to check they do what they are supposed to do, as well as in the final testing when the development is complete.
- Scroll bars:

- Vertical scroll bars may be implemented to windows holding a lot of text and content. Testing will involve both seeing as to whether the scroll works and how accurate it is depending on how quickly the user moves their eye up and down.

Development

Stage 1: pupil detection:

For this algorithm to work, we need to install a facial landmark detector, which can be used to locate faces within an image and determine eye position.

```
#'shape_predictor_68_face_landmarks.dat'
detect = dlib.get_frontal_face_detector() #dlib landmark face detector imported
predictor = dlib.shape_predictor('shape_predictor_68_face_landmarks.dat') #facial feature detector implemented
```

After implementing the webcams Video Capture function, which enables the face/landmark detector to use the webcams input, the main loop is run, which will iterate through and output every frame captured by the webcam.

```
webfeed = cv2.VideoCapture(0) #creates an array of image data for every frame captured by the webcam
while True:
    ret, im = webfeed.read() #im = current frame processed
```

Next, we need to convert the image to grayscale to make processing and contour detection easier, and then the frontal face detector is used to detect all faces within the frame.

```
gray_im = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY) #converts current frame being processed to grayscale
face = detect(gray_im, 0) #detects all faces within the frame using detector
```

As there could potentially be multiple faces detected, initially I am just going to have the program iterate through the detected faces and then later have the gaze detection only work for the largest face detected.

```
for face in face: #iterates through detected faces
    landmarks = predictor(gray_im, face) #predicts landmarks on face |
```

However, if this ends up significantly slowing the system down, I could find a way to store the array of detected faces outside the loop iterating through the faces, and just use the pupil detection algorithm on the largest detected face.

To test if the landmarks were being successfully detected, I had the code iterate through the landmark coordinates and output a circle in its position:

```
for x,y in landmarks: #iterates through landmark coordinates
    cv2.circle(im, (x,y), 1, (255, 0, 0), -1) # outputs circle in coordinate position
```

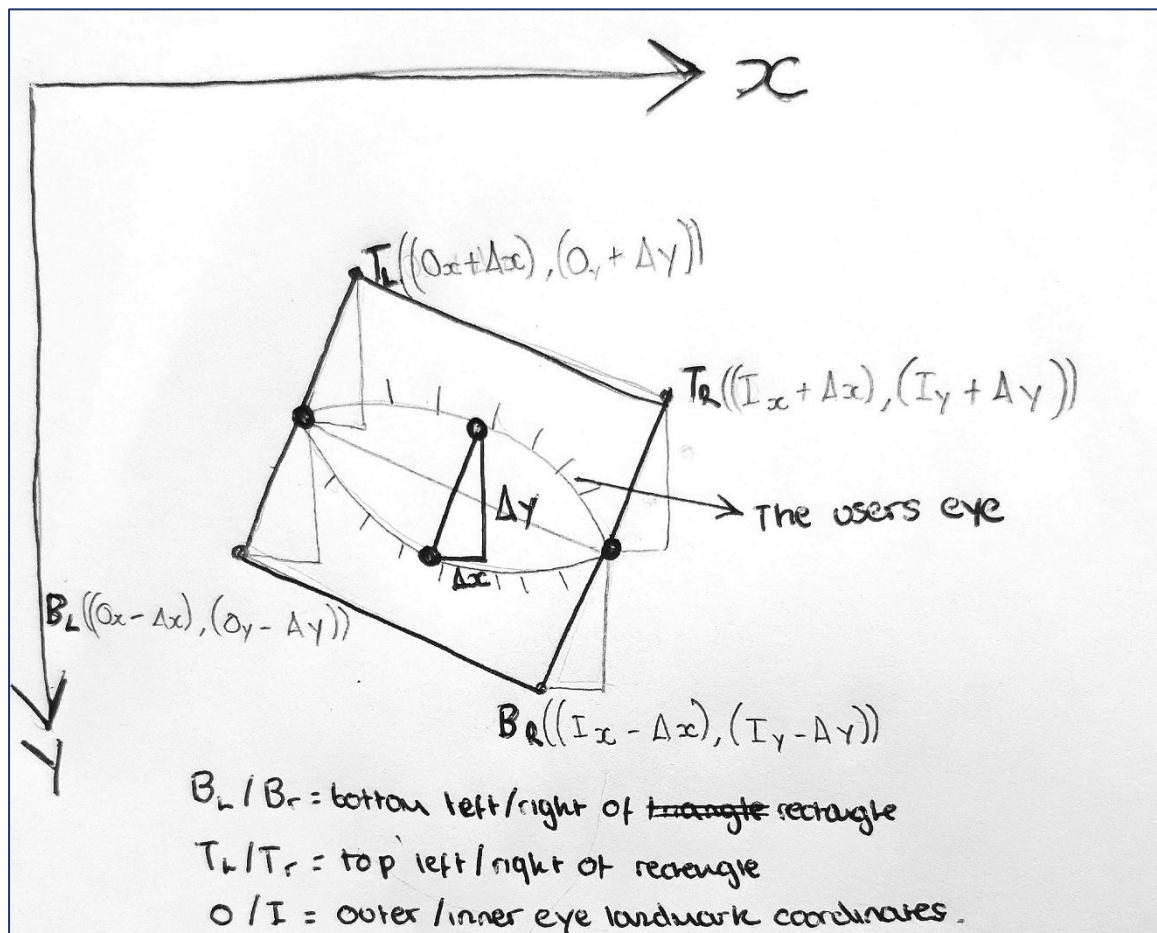
This however through an error as the landmarks detected was not iterable due to it not being an array, but a shape. To solve this, I imported the imutils module so I could convert the shape to a NumPy array using the shape_to_np() function.

Now we have the landmark coordinates we can detect the pupil. Initially I thought we could use the entire image as the region of interest when finding the contours, however early on in design I realised there would be too many contours that could be the pupil within the entire image so it is important to narrow the ROI (region of interest) down to the area around the eyes. This could be done by creating a function that creates a rectangle around the eye, using its landmark coordinates, and then using this rectangle as the ROI.

The landmarks array is a 2d array of xy coordinates and this will be input into the eye rectangle function as the parameter 'array'. Initially I thought there was a landmark coordinate for the top and the bottom of the eye, however there are actually two at the top and the bottom of the eye. Therefore, I found the coordinate of the midpoint between these two points so I can produce a rectangular area around the eye as the new ROI.

```
def eye_rect_finder(array):
    top = int((array[37][0]+array[38][0])/2), int((array[37][1]+array[38][1])/2) #top of eye
    bot = int((array[40][0]+array[41][0])/2), int((array[40][1]+array[41][1])/2) #bottom of eye
    out = array[36][0], array[36][1] #outer corner of eye
    ins = array[39][0], array[39][1] #inner corner of eye
```

Then to produce a rectangle around the eye, I can find the difference in x and y coordinate values between the top and bottom landmarks of the eye. Then accordingly add/subtract these values from the inner and outer eye corner coordinates to create a rectangle that can rotate with the eye:

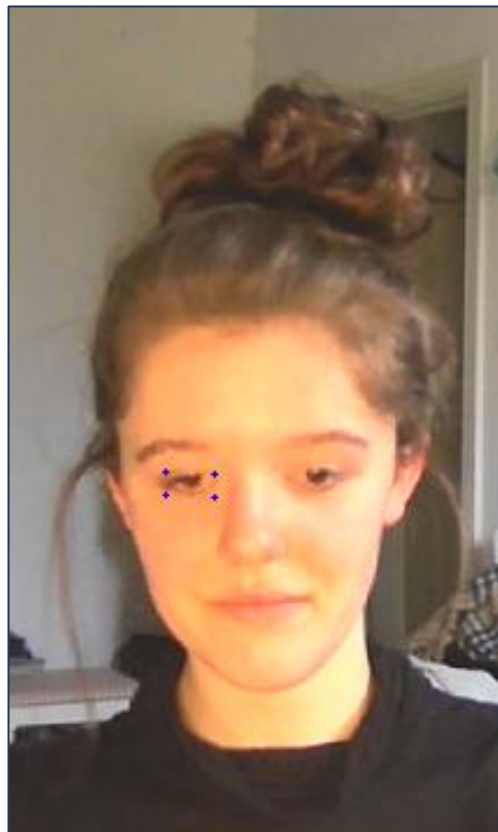


```
diff_x, diff_y = (top[0] - bot[0]), (top[1] - bot[1]) #difference in x,y values of top and bottom of eye landmarks
#coordinate system of frame is such that the origin is in the top left corner
bot_left = [(out[0] - diff_x), (out[1] - diff_y)]#bottom left corner of rectangle coordinate
bot_right = [(ins[0] - diff_x), (ins[1] - diff_y)]#bottom right corner of rectangle coordinate
top_left = [(out[0] + diff_x), (out[1] + diff_y)]#top left corner of rectangle coordinate
top_right = [(ins[0] + diff_x), (ins[1] + diff_y)]#top right corner of rectangle coordinate
```

As well as returning the rectangle coordinates from the function so we can use them for the ROI, I will also return the 4 landmark coordinates of the eye. As the corners can be later used to find the width of the eye which will be needed for the gaze detection algorithm and the top and bottom landmark coordinates will be used to find the height of the eye could be used for the blink detection algorithm.

```
return [bot_left, bot_right, top_right, top_left], top, bot, out, ins
```

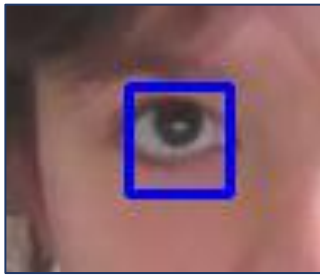
The function is then called inside the face loop where the landmarks array is parsed as the parameter. To test the rectangle coordinate positions, I implemented a temporary loop that iterates through the returned eye_rect array and outputs a circle in the coordinate positions:



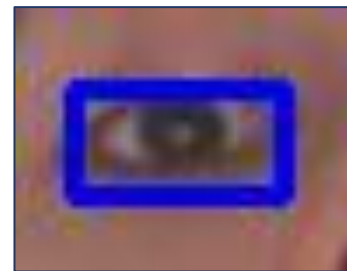
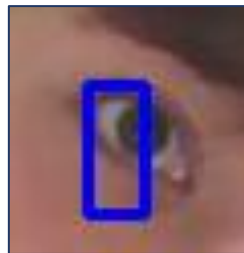
```
for x,y in eye_rect: #eye_rect coordinate position test
    cv2.circle(im, (x, y), 1, (255, 0, 0), -1)#outputs circle in verticie location
```

Whilst trying to implement an algorithm that reduced the frames size to the rectangle produced, I discovered that it would be incredibly difficult to do so as the rectangle isn't parallel to the x and y coordinates of the original frame. Therefore, I tried a different method of producing a rectangle

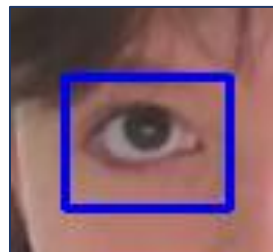
around the user's eye that ensures the rectangle is always at the same orientation (parallel to the frames axis) so reducing the frame size to this new rectangle should be easy:



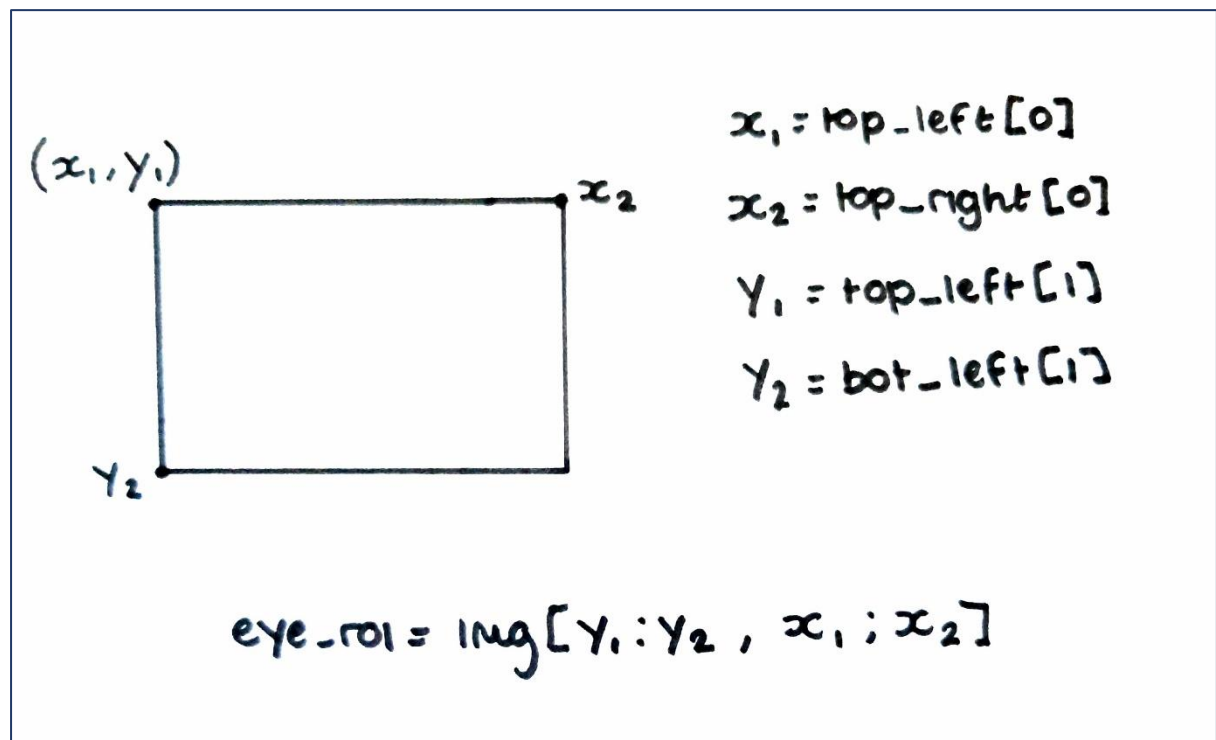
The outer lines of a rectangle were too small so I extended the length of the rectangle by 10 pixels on either side to the whole eye is within the frame. This method worked very well; the only issue occurs when the user tilts their head at an extreme angle. However, the product is designed for people with limited motor abilities and it will be advised in the calibration that the user must keep their head straight and still, so this shouldn't be too much of a problem:



This is the final rectangle where the width has been extended, along with the height by 5 pixels – I realised slightly increasing the height prevents the eye from being cut off when the user tilts their head. Although this means a slightly larger area for the ROI, it still a significant improvement from the original frame size and will still improve the algorithms accuracy in detecting pupils.

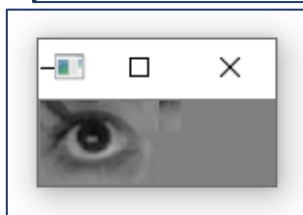


The rectangle has been output around the eye using `cv2.rectangle()`, this was only done to test whether this new method worked and how the image would be cropped. To reduce the ROI, we need to crop the grayscale image of the original frame (`gray_im`):



In code:

```
eye_roi = gray_img[eye_rect[3][1]:eye_rect[0][1], eye_rect[3][0]:eye_rect[2][0]]
#line above crops the original frame to the rectangle around the eye
```

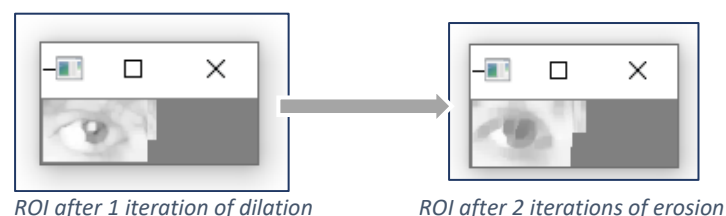


The original grayscale frame has been cropped to this size so this is the new region of interest that will be processed and used to detect the pupil location.

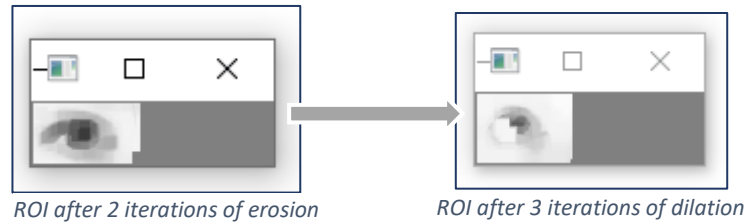
Image processing:

To make detecting the pupil a lot easier we will be processing the image to abstract every frame to its main components through the use of image erosion, dilation and contours. The ROI will be dilated initially to fill gaps, enlarge distinguished features and remove noise from the image.

Then this image will be eroded shrinking connected sets of 1s in the binary image, meaning the pupil should be the most distinguishable feature remain in the processed image:



The pupil was not clear enough in the processed image produced so I swapped the functions around and eroded the image first to test if it would make the pupil clearer:



Through swapping the functions around and changing the interactions, a much clearer pupil contour was produced in the final processed image. I have implemented the image dilation and erosion using `cv2.erode()` and `cv2.dilate()` functions:

```
eye_im = cv2.erode(eye_roi, None, iterations=2) #erodes image
eye_im = cv2.dilate(eye_im, None, iterations=3) #dilates image
```

In order to locate contours, we must convert the image to a binary image, I implemented this using `cv2s cv2.threshold()` function which converts all bits to 0s (white) if they exceed a particular threshold value, and the rest to 1s (black):

Threshold value: 50	Threshold value: 100	Threshold value: 150	Threshold value: 200

The black dot in the 100 and 150 threshold images is the pupil. The 50 threshold was far too low as all pixels were converted to white and the 200 threshold was too high as all pixels were converted to black. Therefore, an optimal value should be found somewhere between 100 and 150:



A threshold value of 135 gave the clearest area of the pupil and will be what we use to find the contours in this section:

Code for creating binary image:

```
_, threshold = cv2.threshold(eye_im, 135, 255, cv2.THRESH_BINARY)
#line above converts all pixels above 135 to white(255) and all values below to black(0)
```

Now we have found the optimal threshold values for detecting the pupil we can find the contours (detect objects) within the frame. This can be done using `cv2.findContours()` and will return the detected contours within the binary image:


```
contours, hierarchy = cv2.findContours(threshold, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
#line above detects objects/shapes within a binary image
```

As multiple contours may be detected, we will use an embedded for loop to iterate through all detected contours. Then for each contour we must find the extent, which is the ratio of the area of the detected contour to the area of the bounding rectangle of that contour. As we are finding a circular contour (the pupil) the extent should be roughly 0.8 as the ratio of a circle to its bounding square is $\pi/4 = 0.79$. Therefore, any detected contour with a ration larger than this should be discarded.

```
for cont in contours: #iterates through detected contours
    bound_box = cv2.boundingRect(cont) #finds the bounding rectangle of the detected contour
    extent = cv2.contourArea(cont)/(bound_box[2] * bound_box[3])
    #finds ratio of contour area (cv2.contourArea()) to the bounding rectangle of a contour (finds extent)
    if extent > 0.8:
        continue #discards all contours with an extent greater than pi/4 = 0.8
```

Within the image, assuming the prior image processing has worked correctly, the pupil contour will most likely have the largest and most dense cluster of pixels within the image, therefore blob detection is the method I will be using for obtaining a coordinate for the centre of the pupil.

A blob is a cluster of pixels with the same grayscale value detected in an image, and the centroid of a blob is the average pixel intensity within that blob and it can be used to find the centre of a blob within an image as well as other properties, such as the radius and area of a blob. Image moments is OpenCV's function for finding blobs and a blobs centre of mass (centroid), where the centroids X and Y coordinates is $\frac{m10}{m00}$ and $\frac{m01}{m00}$ respectively.

When implementing this I realised as the m00 is the sum of blob intensity and the image has been converted to a binary image where the pupil blob is black, a black pixel's intensity is 0 meaning m00 is equal to 0, so a zero error would occur. Therefore, in the line of code where I convert the grayscale image to a binary image, I also invert the image so the white area in the image that the findContour function detects is the pupil and not the area surrounding the pupil:

```
invert = cv2.bitwise_not(threshold) #inverts pixel intensity of image
```

Now we can implement the blob detection into the code, and to ensure no zero errors occur I will set a condition that only the m00's != 0 can be used to find its centre otherwise a zero error would be thrown and the program would stop running:

```
Mom = cv2.moments(cont) #finds the average intensity (moments) of all detected contours
if Mom['m00'] != 0: #discards blobs with average intensity of 0
    cent = [int(Mom['m10'] / Mom['m00']), int(Mom['m01'] / Mom['m00'])] #algorithm for finding centroid value of blob
```

As more than one blob could be detected, other than the pupil, I will create an array to store all detected blob centroids called cont_cont, so when testing the accuracy of the algorithm we can see the effect lighting, head position and movement have on the number of falsely detected pupils.

Now to test whether this algorithm has worked correctly I will use the cv2.circle function to output a circle in the location of the centroid:

```
cv2.circle(im, ((cent[0]), (cent[1])), 1, (0, 255, 0), 5) #outputs circle in blob location
```



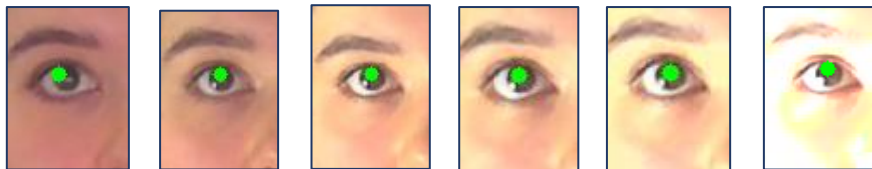
I realised as I had cropped the image initially to reduce the ROI to a rectangle around the eye, the output coordinates were in the position they would be in within the ROI but instead I wanted to output them onto the much larger, original image (im). Therefore, I have to add the centroid x, y coordinates onto the distance between the ROI x and y axis and the original image (im) axis:

```
cv2.circle(im, ((eye_rect[3][0] + cent[0]), (eye_rect[3][1] + cent[1])), 1, (0, 255, 0), 5)
```

Iteration 1 review:

Implemented:

The pupil detection algorithm was successfully implemented and now we have an accurate means of detecting the location of the user's pupil we can move onto developing the gaze detection algorithm. I have however decided not to implement the blink detection algorithm in this stage of development, as I said I would in the design section. This is because the second iteration will be dealing with the functionality of the mouse through the implementation of gaze estimation and blink detection, therefore I believe those two algorithms should be developed together so we can assess the functionality of the mouse as a whole. The program so far filters and thresholds the image to produce a binary image in which contours can be detected. The program uses the extent algorithm to discard contours that are unlikely to be the pupil and obtains an accurate location for the pupil's centroid through the implementation of image moments:



As the image uses a threshold value to convert all grayscale values above a particular intensity to white pixels, lighting intensity and position affects the accuracy of the pupil detection. Having a light to right of the user improves the algorithms accuracy as it reduces glint and increases the pixel intensity of the skin when the grayscale is applied. This is important because when the threshold function is applied, the skin and surrounding features are more likely to be converted to white, so fewer contours (other than the pupil) will be detected. Testing showed that improved accuracy came with brighter light from the right as the skin and glint were not being recognised as pupils. However, the program started to lose accuracy when the light intensity became too great as features became less distinguishable and the facial landmark detector frequently failed at locating the eyes and even the face.

These optimal environmental settings will be advised within the calibration section of the program to ensure the user's experience of using the software is the most accurate it can possibly be. This will now lead onto gaze estimation (mouse function).

Changes to design:

- Instead of using eye_rect array coordinates to crop the ROI of the eye, I changed the algorithm so the axis of the new ROI was parallel to the original frame's axis. I changed to this method because it was far easier to implement and compute as well as being equally accurate.
- Not implementing extent algorithm as a function as it was only utilised once in the program so for the time being it seemed unnecessary.

With the exception of these two changes all other functions and features implemented is accurate to the pupil detections original design.

Success criteria:

No objectives on the success criteria have been fully met so far as no functions being developed in this stage of development will be directly used or seen by the user. This iteration is what will enable the gaze estimation and blink/hold detection algorithms to function, which is what will enable the user to control the mouse. Meaning the following criteria's have been partially met:

- Webcam used to accurately control cursor movements
- Webcam used to control clicks

Stage 2: gaze detection:

In this iteration of development, I will be dealing with 3 different parts of the program:

- Part of the calibration: this section will only involve storing and handling the input data required for initial calibration so the gaze estimation can work correctly. It won't include the calibrations user interface as that will be completed in the third iteration.
- Blink detection
- Gaze estimation: will use the calibration data and the pupil detection algorithm to control cursor movements around the screen – no interface to test on so will implement a testing environment (a window containing multiple dots for the user to try directing the cursor to - explained in the design section).

Calibration:

Using the pupil detection algorithm code and the initial design code, created a new calibration window containing the gaze detection algorithm along with the necessary variables required for the calibration to work:

- **Im_eye_width:**
 - Data type/structure: single float value
 - Use: a variable storing the width of the eye in the image (corner points of eyes coordinates determined using dib land-mark detection classifier and the distance between them is calculated using $\sqrt{(y1-y2)^2 - (x1-x2)^2}$)
- **Re_eye_width:**
 - Data type: single float value

- Use: a variable storing the actual width of a person's right eye in cm – is initially input by user – is used in the calculation for the focal point of the webcam lens. will be stored permanently in a file in the same directory as the program.
- **Dist_from_screen:**
 - Data type: single float value
 - Use: distance from persons eye to camera in cm – is initially input by user - is used in the calculation for the focal point of the webcam lens.

```
re_eye_width = float(input('distance from inner to outer corner of eye(cm): '))
dist_from_screen = float(input('approximate distance from users eye to the webcam(cm): '))
#lines above are a temporary way of inputting user data (real eye width and distance from screen)
```

```
im_eye_width = math.sqrt((out[0] - ins[0])**2 + (out[1] - ins[1])**2) #image eye width calculated for focal length calculation
```

Obviously, the final product, the `re_eye_width` will be entered via an input box on the calibration menu, but for now I'll be entering them directly onto the shell – 2.7 cm for the eye width and 40cm for the distance used.

Secondly, we must implement the `focal_finder` function in order to calculate the cameras focal point, which will be used to calculate the distance between the user and the camera for implementing the gaze estimation algorithm.

```
def focal_finder(rew, dfs, iew):
    focal_length = (iew * dfs)/rew #calculate the focal point of the camera lens at each frame
    return focal_length
```

This function is called for every frame, therefore the focal lengths calculated must be stored in an array called `foc` – through doing this once a certain number of focal points have been stored in the array `foc`, the while loop will terminate and the average focal length can be calculated and stored in a text file called 'calibrationinput.txt' along with the pupil position and distance from the camera during calibration.

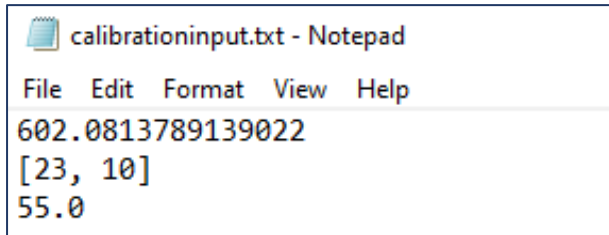
```
vals_to_move = [(sum(focal)/len(focal)), cent, dist_from_screen] #stores average focal length along with
#the central position and distance to the camera of the pupil

file = open('calibrationinput.txt', 'w')

for i in vals_to_move: #writes the contents of the array into the file 'calibrationinput.txt'
    file.write(str(i) + '\n')

file.close()
```

We can now import these values to the gaze estimation program.



Blink detection:

This section won't take long as we have already calculated the height and width of the eye. all that needs to be imported is the time module so that we can find the times where the ratio of eye height to eye width is close to 0. If the time where this ratio is true is < 2 seconds and greater than 0.5 seconds a click is recorded. If the ratio is held for a time larger than 2 seconds a hold is recorded. As there are no buttons to test the clicks and holds on, I will evoke the click and hold in the third iteration when the interface is being developed. For now, I will print 'click' and 'hold' when the program detects that either one is occurring:

```
im_eye_width = math.sqrt((ins[0]-out[0])**2 + (ins[1]-out[1])**2) #finds the eye width of the eye detected in frame
im_eye_height = math.sqrt((top[0]-bot[0])**2 + (top[1]-bot[1])**2) #finds the eye width of the eye detected in frame
if im_eye_height/im_eye_width < 0.1 and closed = False: #boolean variable is initially set as false outside of main loop
    timel = time.time() #sets current time
    closed = True #changes closed to true so current time isnt overwritten
elif im_eye_height/im_eye_width >= 0.1:
    diff_time = time.time() - timel #finds difference between the current and initail time the eye was closed (timel)
    closed = False #closed set back to false
    if diff_time < 2 and >= 0.5:
        print('click') #would evoke the mouse to click if the blink hold is < 2 and > or equal to 0.5
    elif diff_time >= 2:
        print('hold') #would evoke the mouse to hold as long as the blink hold is > or equal to 2
```

Gaze estimation:

On a separate python file to the calibration, I create a copy of the pupil detection code. Then I need to import the data stored in the calibration input text file so it can be used in the gaze estimation algorithm. I then realised I also required both average inner and outer coordinate of the eyes to be sent to the calibration input file so they can be used to find an approximate coordinate for the centre of the eye, along with the real eye width, so it can be used to calculate the distance between the screen and the user:

```
average_in = [sum(x)/len(x) for x in zip(*in_coords)] #finds average coordinates for inner corner of eye
average_out = [sum(x)/len(x) for x in zip(*out_coords)] #finds average coordinates for outer corner of eye
vals_to_move = [(sum(focal)/len(focal)), cent, dist_from_screen, re_eye_width, (average_in, average_out)] #stores average focal length along with
#the central position of the pupil, the distance to the camera from the pupil, the real width of the eye and the average inner and outer coords
```

The data was then read from the text file on the gaze estimation window and the values in it were assigned to their appropriate variables:

```
vals = [line.rstrip('\n') for line in open('calibrationinput.txt','r')] #retrieves calibration data from file 'calibrationinput.txt'
focal = vals[0]
pupil_position = vals[1]
```

```
dfs = vals[2]
re_eye_width = vals[3]
in_out_coords = vals[4]
#assigns all calibration data to their appropriate variable names
```

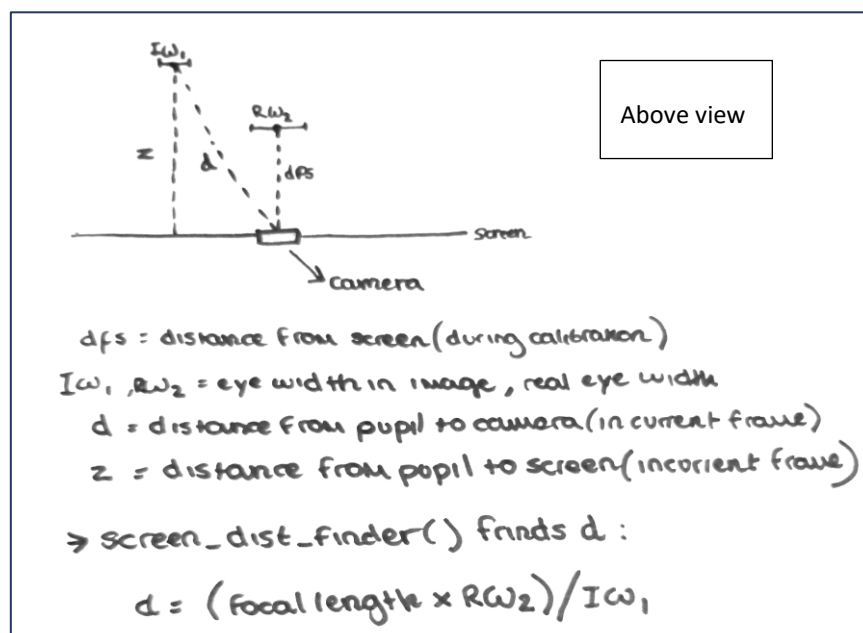
The approximate pupil centre during calibration is then calculated using the in_out_coords as this will be required when finding the distance between the user's eye and the screen (z coordinate of screen):

```
cal_pupil_centre = (in_out_coords[0][0]+in_out_coords[1][0])/2, (in_out_coords[0][1]+in_out_coords[1][1])/2)
#line above calculates an approximation for the centre of the eye by finding the mid point of the inner and outer corner coordinates
```

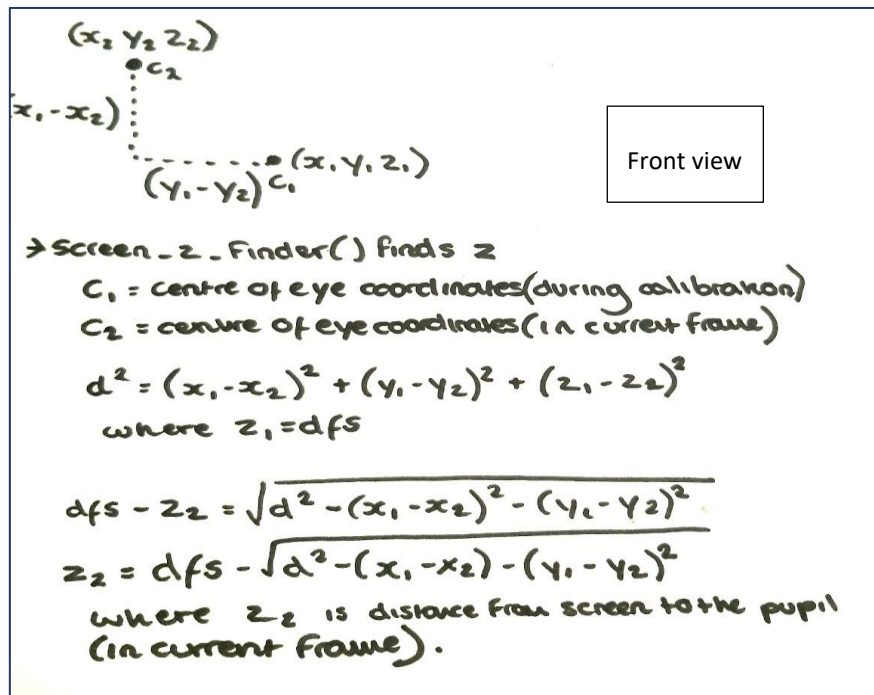
Then within the main loop of the pupil detection algorithm, the image eye width needs to be calculated as it is a parameter for the screen_dist_finder() function, this is done using Pythagoras on the inner and outer coordinates of the eye within the current frame:

```
im_eye_width = math.sqrt((ins[0]-out[0])**2 + (ins[1]-out[1])**2) #finds the eye width of the eye detected in frame
```

Whilst coding the screen_dist_finder() function, I realised this was all we might need to find the z coordinate of the screen, as originally screen_dist_find() was there to find the exact distance between the camera and the users pupil:



Whereas screen_z_finder() was required to calculate difference in the z coordinates of the user's eye and the z coordinates of the screen:



However, as the `screen_dist_finder()` function uses the user's eye width in the image, which is assumed to be flat, to find the exact distance from the user's eye to the camera. It means that effectively the distance shouldn't be different from the z value calculated in the `screen_z_finder()` function, unless the user's head is tilted at a very extreme angle or is at the boundary of the webcam frame – which the user will be instructed not to do in the info page.

This means for the time being I'm just going to be using the `screen_dist_finder` calculation as the z distance between the user's eye and the screen:

```
def screen_dist_finder(rew, f, iew)

    screen_dist = (f * rew)/iew #finds the distance from the users eye to the computer screen

    return screen_dist
```

Now the pupil z function can be implemented by approximating the eye to be a sphere with radius equal to half the image eye width, which is used to find the z value of the point of the eyeball that the pupil is located. Before the function can be called the eyeball centre must be calculated (`im_eye_centre`). For x and y , this is done by finding the midpoint of the inner and outer coordinates of the eye, and for the z coordinate, it will be set to the screen distance + half the eye width as that is the radius of the eyeball:

```
im_eye_centre = ((ins[0]+out[0])/2, (ins[1]+out[1])/2, 0.5*im_eye_width+screen_dist)
#line above calculates coordinates for the centre of the eyeball
```

Instead of parsing the coordinates as separate variables, they were parsed as arrays with each coordinate x , y , z being referenced by index value 0, 1, 2 as it's easier to follow the code and know which coordinate is for the pupil and which is for the eyeball centre:

```
def pupil_z(eyewidth, screen_dist, pupil, eyeball):
    # $(0.5 \times \text{eye\_width})^2 = (\text{pupil}[0] - \text{eyeball}[0])^2 + (\text{pupil}[1] - \text{eyeball}[1])^2 + (z - \text{eyeball}[2])^2$ 

    z = math.sqrt((0.5*eye_width)**2 - (pupil[0] - eyeball[0])**2 - (pupil[1]- eyeball[1])**2) #finds z coord of pupil
    return z
```

The value returned for z is then appended to the pupil coordinates (cent).

Now for the `centre_to_pupil()` function, used to create a function of a line that passes through the centre of the eye and through the pupil. As the z distance to the screen is known to be 0 at the screen, we can set this equal to the functions parametric equation of z to find the value of t, which can be used to estimate where the x and y coordinates may be located on the screen. This will be done quite differently that how it was designed as in the original design the difference in coordinates isn't used as the direction vector, and the coordinates have been parsed as variables. They will also be parsed as arrays in this function. In addition to this, in the original design, the parametric functions were returned, with the value for t, x and y being found after the function was called. I decided to have the function calculate the value of t, x, and y so it was actually being used in the code rather as in the design, it didn't really serve a purpose:

```
def centre_to_pupil(pupil, eyeball, screen_dist)
    # $z(t) = \text{eyeball}[2] + (\text{pupil}[2] - \text{eyeball}[2]) * t$ 

    t = (screen_dist - Eyeball[2]) / (pupil[2] - eyeball[2])

    x = eyeball[0] + (pupil[0] - eyeball[0]) * t
    y = eyeball[1] + (pupil[1] - eyeball[1]) * t
    #finds x and y value for t when z is equal to the distance from the screen

    return x, y
```

Before using these coordinates to control the mouse, initially the measurements given by the user in the input stage was in cm so we need to convert these input values to pixel units by multiplying them by 37.795:

```
import pyautogui

re_eye_width = float(input('distance from inner to outer corner of eye(cm): ')) * 37.795
dist_from_screen = float(input('approximate distance from users eye to the webcam(cm): ')) * 37.795
#lines above are a temporary way of inputing user data (real eye width and distance from screen)
```

Now we must import pyautogui in order to control the mouse using these coordinates:

```
pyautogui.moveTo(x, y) #sets cursor to gaze estimation coordinates found from centre_to_pupil()
```

However, if the coordinates given are > screen resolution or < 0 the loop should continue. Therefore, conditions must be set, and system metrics must be imported from win32api to get the screen resolution:

```
from win32api import GetSystemMetrics
```

```
if x > GetSystemMetrics(0) or y > GetSystemMetrics(1) or x < 0 or y < 0: #ensures the coordinates are not out of range of screen so
    #no error will be thrown
    continue
```

when running the program, I realised I had been using the image coordinates to find the position of the screen, meaning the cursor was only moving a tiny amount. Therefore, I will change the real coordinates of the calibration pupil location to be relative to the screen (where the origin is in the top left corner, the webcam is in the middle of the top of the computer and the user is measuring parallel to the x axis when calibrating). This means we need to create a calculate magnification function so we can convert image measurements to real life measurements. I will calculate this in addition to the focal length during calibration and it will also be stored in the calibration input file.

```
magnification = iew/rew #calculates magnification of image |
```

```
vals_to_move = [(sum(focal)/len(focal)), cent[0], cent[1], dist_from_screen, re_eye_width, (sum(magnification)/len(magnification))]
```

This is then read from 'calibrationinput.txt' in gaze estimation program and stored in variable mag. The real-life calibration pupil coordinates are set so they can be used later to find the real-life coordinates of the pupil detected in frame:

```
cal_re_coords = (0.5*GetSystemMetrics(0), 0, dfs) #find the real life coordinates of the pupil during calibration (relative to screen)|
```

Once we've found the screen distance from the new pupil using the screen distance function, I am going to call a new function called find_real_xydiff() which finds the difference in the x and y image coordinates of the calibration pupil, the centre of the eye detected in the frame (midpoint of inner and outer pupil) and the centre of the pupil detected in the frame (cent). It then divides the difference by the magnification to find the real-life difference in x and y coordinates and returns these values:

```
centre_diff, pupil_diff = find_real_xydiff(im_eye_centre, cent, pupil_position, magnification)
```

```
def find_real_xydiff(new_centre, new_pup, cal_pup, mag):
    pup_diff = (cal_pup[0]-new_pup[0])*mag, (cal_pup[1]-new_pup[1])*mag
    centre_diff = (cal_pup[0]-new_centre[0])*mag, (cal_pup[1]-new_centre[1])*mag
    return centre_diff, pup_diff #returns the real life difference in x and y coords of calibration pupil, the new pupil detected in current frame,
    #and the centre of the eye detected in current frame
```

After doing this I calculated the new coordinates of the eyeball centre and pupil location by adding the difference onto the real calibration coordinates for the pupil:

```
new_cent = [(cal_re_coords[0] + pupil_diff[0]), (cal_re_coords[1] + pupil_diff[1])]

new_eye_centre = [(cal_re_coords[0] + centre_diff[0]), (cal_re_coords[1] + centre_diff[1]), (0.5*re_eye_width + screen_dist)]

z = pupil_z(re_eye_width, new_cent, new_eye_centre)
```

I have also changed the parameters for the pupil_z() and centre_to_pupil() functions to the real-life coordinates so the z value calculated is also calculated. The z value is also now appended to new_cent, rather than cent.

Now we have found real life coordinates of the detected pupil we can calculate a function for the user's gaze by parsing the `new_cent` and `new_eye_centre` arrays by reference into the `centre_to_pupil()` function. This should return the approximate real-world coordinates for where the user is looking on the screen.

Section 4: changes to gaze estimation algorithm:

Unfortunately, due to the inaccuracy of the method and the unsophisticated equipment being used to carry out the gaze estimation, I have had to change the algorithm so it detects whether the user is looking up, down, left or right and uses this information to move through the keyboard and buttons within the virtual keyboard.

There are two main reasons the initial algorithm was unfeasible:

- **Lack of equipment required for accurate pupil detection:** given the pupil detection only required a low-resolution webcam and a well-lit room, the algorithm was quite accurate. However, the accuracy was not a level where real-world systems could be modelled well as many calculations relied too heavily on approximations, such as the users' eye's location relative to the screen, and the approximate centre of the eye. Approximations had to be made as certain equipment often used by gaze estimation software was not available. Two examples of these are infrared lasers and a head rest. Infrared lasers are often used to remove glint from the camera making the pupil much clearer and easier to detect, whereas for our algorithm, glint as well as a poor camera quality meant the pupil detected was often a little bit off centre due to the program occasionally processing the glint as the pupil coordinates rather than the actual pupil centre. Whereas a head rest is often utilised to ensure the users head remains in the same spot, so when calibrating the distance from the screen inputs is more accurate and will remain the same throughout the whole calibration process.
- **Positional approximations:** approximations had to be made throughout to save the user from having to complete an extensive calibration every time the program is run. Meaning positions, such as the users eye position during calibration relative to the screen, and even the screen distance during calibration (due to no head rest, meaning the users head would most likely vary from this position during calibration), were all estimations.

The combined uncertainties of all of the calculations done throughout the code meant that the final predicted coordinates were very far from where the user is intending to look:

In addition to this the extensive number of calculations being run in this code meant that the delay between the user moving their eye and the mouse moving was too significant for the program to be considered user friendly or even usable.

This means the first criteria cannot be achieved:

Webcam used to accurately control cursor movements	<ul style="list-style-type: none"> • Images of filtered images detecting the pupil/circles in a frame. • The code used to calculate the estimated point of gaze to translate it to mouse movements. • The testing to show whether it's working accurately (<0.5 second delay)
--	--

Therefore, I have consulted one of my stakeholders (Jos Sullivan) on the issue with the statement:

'Hi, Jos

Unfortunately, due to lack of professional equipment, the gaze detection algorithm cannot be implemented as initially proposed. Meaning the user cannot directly use their pupil to move the mouse around the screen. As a result of this, the criteria for using the webcam to accurately control cursor movements cannot be effectively met. However, an alternative method is using the users gaze direction to determine whether the cursor should move right, left, up or down throughout the virtual keyboard and program windows. Do you think this is an acceptable alternative?'

Response:

'Yep, that sounds fine. Seems simple and user friendly, and maybe more accurate. It may be a lot slower to use though which could be frustrating for some people but I suppose it's better than using a software that doesn't work.

Well for requirements, you could maybe make sure there's a range. So, the cursor should only move if the user is looking up or down or in a direction a certain amount, so the mouse can actually stay in the same place rather than moving around all the time.'

Requirements:

- The algorithm must detect whether the user is looking up, down, left or right:
 - This can be done by finding the difference in x and y coordinates between the pupil and landmarks for the inner and outer corner and top and bottom of eye coordinates. Whichever coordinate has the smallest difference (aka which ever part of the eye is closest to) gives a clear indicator as to where the user is looking.
- The algorithm must only move the cursor when the user's pupil is outside a certain range:
 - Meaning there should be a max value for the difference in coordinates meaning anything larger would cause the cursor to remain stationary and not move around.
- The cursor can only move up again once the user's pupil has returned back to the centre (or the stationary range):
 - This is to ensure the mouse doesn't go flying around the keyboard when the user is looking outside the stationary range. This means it'll be a lot easier for users to control movement of cursor and ensure it stops at particular keys.

Design: variables:

- eye_coords:
 - o Stores 2d array of float values which are the coordinates of the inner and outer corner and the top and bottom of the eye. the difference in the x and y values of the coordinates stored in this array and the pupil detected will be calculated.
- diff_x, diff_y:
 - o stores a 1d array of floats where the diff_x array stores the difference in x value of the coordinates stored in eye_coords and the detected pupil coordinates, whereas diff_y stores the difference in y value. The minimum difference will then need to be found by comparing the minimum values stored in these two arrays.
- stat:
 - o This variable will store a Boolean value, so will act as a flag to determine whether the cursor has returned to the stationary region. Initially set to true, when the pupil is outside the stationary region it is set to False, and when it returns to stationary region stat is set to True.

Development:

so, I will begin by putting the eye coordinates (top, bot, ins, out) into the array eye_coords so diff_y and diff_x can be easily calculated using a for loop:

```
eye_coords = [top, bot, ins, out]
diff_x = [(cent[0] - x[0]) for x in eye_coords]
diff_y = [(cent[1] - x[1]) for x in eye_coords]
#finds difference between x and y coords of pupil and inside, outside, top and bottom of eye
```

Then as some coordinates are negative and some a positive and we need to compare how small they are by how close to zero the difference is, I'll find the minimum values stored in diff_x and diff_y using the min() inbuilt python function and compare the absolutes of these values using the abs() inbuilt python function, so they are being compared irrespective of sign:

```
if abs(min(diff_x, key=abs)) < abs(min(diff_y, key=abs)) and abs(min(diff_x, key = abs)) < 2 and stat = True:
#compares the minimum differences in each array, irrespective of sign
#ensures the minimum diff is outside the stationary region
#ensures the users pupil was stationary before this frame

    if min(diff_x, key=abs) < 0:
        print('left')
    else:
        print('right')
    stat = False #ensures the users cannot move the cursor again until their pupil returns to stationary pos
```

In addition to comparing the absolute minimum differences, whether the difference is outside the stationary region and whether the user was in the stationary region in the previous frame are also checked in the if statement.

If all of these conditions are true the user's pupil must be outside the stationary region. The user's pupil also must have been in the stationary region in the previous frame, meaning the user can move the cursor left or right in the virtual keyboard, where if the difference is minus the cursor will be moved left and if the difference is positive the cursor is moved right.

This code is then repeated for when the absolute minimum of `diff_y` is less than `diff_x`, where a positive difference means the cursor should be moved up and a negative difference means it should be moved down.

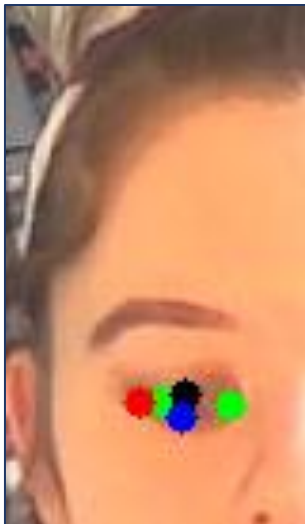
If none of the conditions are true, the user's pupil must be in the stationary region so `stat` is set to `True`:

```
else:
    stat = True #if no conditions true the stationary flag is set to true
               #as user must be in stationary region
```

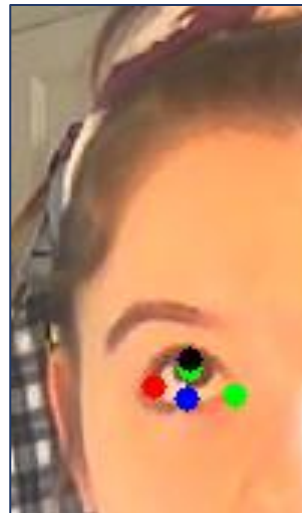
Upon testing I realised there was a few issues. When the `diff_x`, `diff_y` arrays were produced, if the pupil was to move very far to the left or right, the pupil would never quite reach the corner of the eye. In addition to that, as the array was storing the difference in `x` and `y` for all vertices, the minimum `diff_y` value would be between the pupil and the in and out coordinates as the pupil was always roughly level with the eye corner coords. Therefore, I had to change the `eye_coords` array into `eye_coords_x` (storing ins and out coords) and `eye_coords_y` (storing top and bot coords):

```
eye_coords_x = [ins[0], out[0]]
eye_coords_y = [top[1], bot[1]]
diff_x = [(cent[0] - x) for x in eye_coords_x]
diff_y = [(cent[1] - x) for x in eye_coords_y]
```

Therefore, we had `diff_x` iterate through `eye_coords_x` and `diff_y` iterate through `eye_coords_y`. the method still wasn't producing the desired result, often just continuously printing up and down. Therefore, we needed a better way to control the calling of up and down so it doesn't call as frequently, and is a better way for the program to recognise when the user is looking left and right.



Looking down



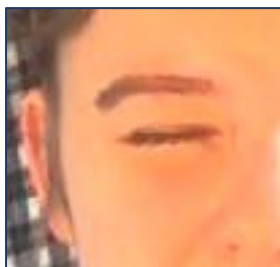
Looking up

To stop the program from just randomly printing up and down, I thought of a better way to determine whether the user is looking up and down – how open or closed the eye is, or in other words, how large or small the ratio of eye width to eye height is. Therefore, I implemented a code that, after checking whether the user is looking left or right, calculates whether the eye height to

width ratio is smaller or larger than a particular value. From the use of trial and error I found for looking up, the average ratio is 0.6 so I set the condition as if the ratio is greater than 0.5 the user is looking up, to account for people with smaller/less round eyes. For looking down, the average ratio was found to be around 0.3 therefore I set the condition as if the ratio is less than 0.3 and greater than 0.29 the user is looking down. The reason the ratio must be greater than 0.29 is so looking down doesn't get confused with a blink. In addition to the above changes, under each condition I made the program pause for 1 second using the `time.sleep()` function so the program wouldn't continuously print directions like it was before with up and down:

```
elif eye_ratio < 0.3 and eye_ratio > 0.29 and stat == True:
    print('down')
    stat = False
    time.sleep(1)
elif eye_ratio > 0.5 and stat == True:
    print('up')
    stat = False
    time.sleep(1)
else:
    stat = True
```

This stopped the program from continuously printing up and down, now only printing up when the user looks up and down when the user looks down. However, the program stopped recognising when the user is blinking, I had to change my original method of recognising blinks:



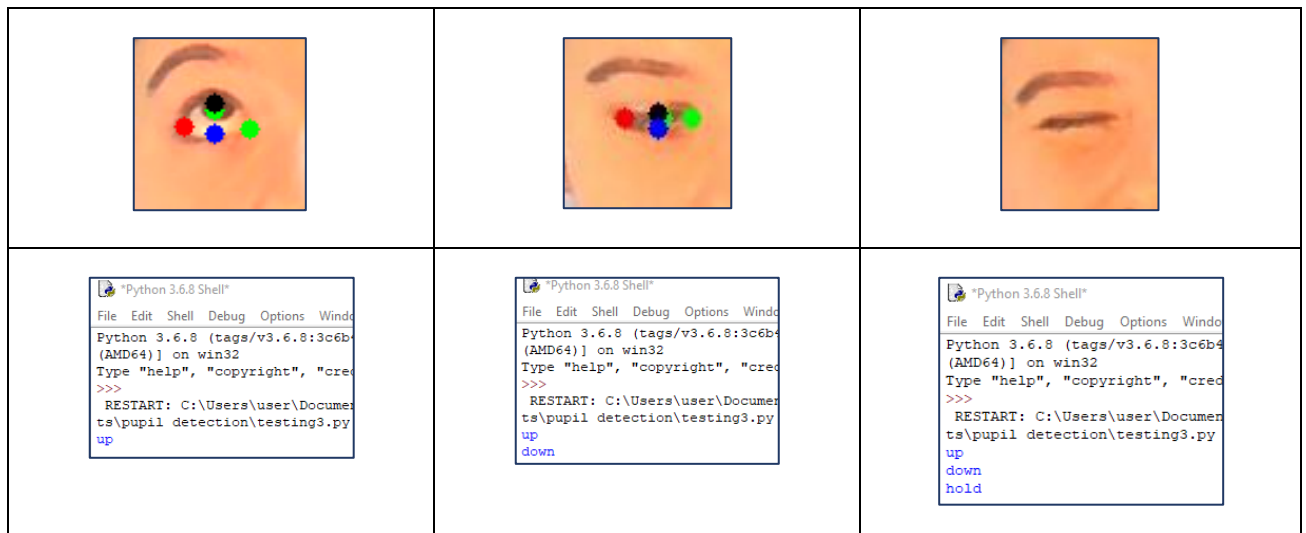
I can now see why my original blink detection algorithm was only partially accurate. When the user closes their eyes, more often than not, the dlib landmarks cannot be detected so a pupil cannot be detected. Therefore, we can use the absence of the user's pupil to determine whether they are blinking. If a contour in the image has an average intensity not equal to 0, it means a pupil has been detected, as an elif statement to that condition I check if the closed variables Boolean value is false, if so, closed is set to

true and time2 is set to `time.time()`. Then the next time a pupil is detected (if `Mom['m00'] != 0`) closed is checked to be true. If it is true, closed is set to false and the original blinking algorithm is used (if the difference in time is < 2 and ≥ 0.5 print click, if the difference in time is ≥ 2 print hold) – whilst checking for each of these conditions the program pauses for 2 seconds. This allows the user to open their eyes, meaning whilst they are opening their eyes and the eye height to width ratio is low, the program cannot output down:

```
elif closed == False:
    time2 = time.time()
    closed = True
```

```
if closed == True:
    diff_time = time.time() - time2 #finds d
    closed = False #closed set back to false
    if diff_time < 2 and diff_time >= 0.5:
        print('click') #would evoke the mous
        time.sleep(2)
    elif diff_time >= 2:
        print('hold') #would evoke the mouse
        time.sleep(2)
```

Now when the code is run:



Now for the left and right detection. To simplify the code so there's fewer conditions to be met in order to produce an output of left and right, only two conditions are checked:

- **The pupils x coordinates relative to the x coordinate of the top coordinate** – as if the x coordinate of the pupil is greater than the x coordinate of the top coordinate, there is a chance the pupil is looking to the left. Same with the right if the pupil x coordinate is less than the top x coordinate.
- **If the above condition is met, we can then check how far the pupil is from the opposite corner of the eye** – so example, if the pupil is potentially looking left on the screen, if the distance from the pupils x coordinate and the outer corners x coordinate (as the screen is flipped) is greater than the image eye width * k, where k is a value less than 1 but greater than 0, the user must be looking to the left. The same applies for right, but the inner corners x coordinate is used instead. This prevents left or right being output if the user is just slightly looking to the left or right. The optimal value of k was found, from trial and error, to be 0.75 for left and 0.77 for right:

```
if cent[0] > top[0] and (cent[0]-out[0]) >= 0.75*im_eye_width and stat == True:
    print('left')
    stat = False
    time.sleep(1)
elif cent[0] < top[0] and (ins[0]-cent[0]) >= 0.77*im_eye_width and stat == True:
    print('right')
    stat = False
    time.sleep(1)
```

Changes to calibration:

As a result of these changes the calibration process and window are no longer needed in the program as we do not need to be finding the distance between the user and the screen at any point in the algorithm. This means the keyboard may potentially be less cluttered due to the lack of a calibration button, potentially making the virtual keyboard easier to navigate. It also means user will not have to do an initial calibration or continually recalibrate the program if it ever stops working, so the time spent actually using the program is optimised therefore the user won't grow tired of having to frequently recalibrate the software. Removing the calibration also gives the user more

independence as the initial proposed calibration required the assistance of someone to measure the users eye width as well as their distance from the screen.

Iteration 2 review:

Implemented:

In this implementation I have managed to get the blink detection algorithm working correctly meaning when it comes to applying the algorithm to the keyboard interface the user will be able to effectively interact with the virtual keyboard keys in order to communicate. In addition to the blink detection, we failed to execute the initial gaze detection, however we managed to program an alternative method that will still allow the user to control the selection and movement of an entity/selector throughout the virtual keyboard. Although this method is slightly slower and less advanced, it is significantly more accurate given the equipment being used for the program. With the initial gaze estimation algorithm, the user would have to keep their head unrealistically still and it required a fairly extensive and complicated calibration process to work correctly. In addition to this, many users would not have the patience and time for this sort of tedious requirements, meaning, although this new method of communication may be slower to execute as the user has to traverse the keyboard key by key to get to a desired location, at least a user can get started using it as soon as possible. I consulted with one of my stake holders that I have altered the program such that the initial criteria of it has been slightly altered, as I am no longer controlling the cursor with my gaze, but rather a selector, throughout the virtual keyboard. She agreed that the program still met the basic foundations of the initially proposed program, therefore this will be the gaze detection method I will be applying to the communications interface.

Changes to design:

- Changing the gaze estimation algorithm – initial plan was having the user control the movement of the cursor freely using their pupil, however due to the lack of professional equipment and the frequent use of approximations within the program, it turned out to be very inaccurate and non-functioning. Therefore, I changed the algorithm so it detects whether the user is looking left, right, up and down which will determine the direction the user will move throughout the keyboard.
- Removing the calibration – as the new gaze estimation algorithm no longer requires the distance from the screen, calibration is no longer needed, meaning the program was discarded and the calibration window button will no longer be present on the virtual keyboard menu.

To test:	
Cycle 1: Pupil detection:	
Pupil detection program can detect pupils	✓

Pupil detection program can detect clicks and hold commands from blinks	✓
Cycle 2: Gaze estimation:	
Gaze estimation program enables the cursor to be controlled by pupil movements	✓ (partially)
Gaze estimation program accesses calibration data from external file and uses them correctly - (no longer needed)	-

Success criteria:

- Webcam used to accurately control cursor movements – partially, as although the user is technically controlling the movement of an element (selector) throughout the screen, the user is not controlling cursor movements unfortunately.
- Webcam used to control clicks.

Stage 3: Interface development:

This part of development will be divided into 2 parts: the menu window, then the info, settings and word bank window being done together. Due to the recent changes to the gaze detection algorithm, I have had to change the menu design as originally, it was heavily utilising tkinter elements such as buttons for the help windows and text widget for the input box. These elements were however impossible to be rendered alongside the gaze detection while loop without implementing threading, which I was not prepared to use. Therefore, I decided I should use a canvas to design my window instead, creating buttons and input boxes from rectangles rather than tkinter widgets. This is actually a very suitable change as not only does it allow the window to be rendered intime with the pupil position being calculated, as the user is no longer clicking buttons, but rather traversing a keyboard grid with a selector, buttons do not need to actually be used. In addition, when we call the help windows via their buttons on the menu window, the contents of their windows will overwrite the original canvas, meaning multiple windows will not be created, meaning less clutter ensuring the state of the software remains organised for the user. In addition to this I will be implementing the interface within the same python file as the gaze estimation as it should be easier to navigate the program when all of the code is contained in one place.

Keyboard/menu window:

To begin I import tkinter and PIL, which is needed to changes the cv2 webcam feed to a format that can be drawn onto the canvas. Then I created the main window using tkinter.Tk() and set its width and height to 1200 and 750 respectively. The I set created the canvas for the window; were the canvas width and height is set equal to the window width and height. The canvas is then packed onto the root window:

```
#make window
root = tkinter.Tk() #sets tkinter window
win_width = 1200 #sets width and height of canvas
win_height = 750
home = tkinter.Canvas(root,width=win_width, height=win_height)#everything is drawn onto canvas
home.pack() #puts canvas in tkinter window
```

Next, I defined all the keys on the keys present on the keyboard in a 2d array, with each row storing the characters in one row of the keyboard. The array variable was initially 'alph' in the design however I decided to change it to 'keys' as its easier to understand considering the keyboard stores more than the alphabet.

```
keys = [['1','2','3','4','5','6','7','8','9','0','+','-','=', '*', 'BACK'],
        ['q','w','e','r','t','y','u','i','o','p','[',']','?', '!', 'ENTER'],
        ['a','s','d','f','g','h','j','k','l','/','\',';', ':', '(', ')'],
        ['z','x','c','v','b','n','m','&','@','#','_','~','"','<','>'],
        ['CAPS', 'SPACE', '{', '}', '[', ']', '\\', '']]
```

Next, we must define the positions of the keys as we are no longer using tkinter buttons. First, I created an empty array called key_positions. The program uses a nested for loop to iterate through each row of the keys array and creates a set of coordinates for each key on the keyboard:

```
key_positions = [] # stores x and y positions of all the keys on the keybaord
for i in keys:#the '1' key is [0,0]
    for j in i:#this loop creates these coordinates
        key_positions.append([keys.index(i),i.index(j)])#if key exist in keys array, create coordinates for it
```

As well as the keyboard positions, we must also set the info, settings and word bank button positions – which will be 9 buttons along and as the coordinates system for the key positions are using the top left button as the origin, the y coordinate of the buttons must be negative as they are above the keyboard. We can then add these button positions to the key positions array, as well as set the colours and text of the info, settings and word bank buttons:

```
button_positions = [[-1,9],[-2,9],[-3,9]] #the coordinates of the help buttons relative to [0,0]
key_positions += button_positions #adds button coordinates to other ones
|
button_colours = ['red','green','blue'] #colours for help buttons
button_texts = ['HELP','OPs','INFO'] #text for help buttons
```

Before creating the menu window function, we must set all the necessary variables, including:

- Rows
- Button_width/height

- x/y_offset – this is the distance from the keyboard to the top and left of the screen which is required to ensure the keyboard is rendered at the bottom as opposed to the top of the canvas.
- Margin – stores the number of pixels between each key
- Button_fonts – set to arial 20 bold
- Key_selection - is the set of coordinates that the selector will appear on.
- Output_string – the initially empty string that is displayed on the input rectangle
- CAPSLOCK – initially set to False
- Cam_width/height

Current_win is also a variable defined that will be used throughout the program to determine which window needs to be rendered in a frame. It is initially set equal to 'home':

```
rows = 15 #rows in keyboard
y_offset = 345 #distance of keyboard from top of screen
x_offset = 7.5 #distance of keyboard from left of screen
button_width = 72 #width of keys
button_height = 72 #height of keys
margin = 7.5 #pixels between keys/buttons
button_fonts = 'Ariel 20 bold' #font and size of the text
key_selection = [2,8] #keyboard coordinates of the currently selected key (key with red border)

output_string = '' #string displayed in blue text on
CAPSLOCK = False #toggles if letters uppercase or not

cam_width, cam_height = 270, 270 #width and height of webcam displayed on canvas

current_win = 'home'
```

Before creating our home function, we need to implement the move and write functions. The move function takes in the direction determined by the pupil position and

Next, we can start creating home_window() function – was changed from menu_window to home_window from design as it was too easy to confuse with the settings window. Initially we update and the home canvas within the function to enable the canvas to update with each frame. Then we tag each element we draw onto the canvas with destroy so they are deleted before the next frame – to avoid each layer of the canvas being drawn over and stored in memory. I then used a rectangle to set the entire window background to a solid grey:

```
def home_window():
    home.update() #update the canvas every frame
    home.delete('destroy') #delete everything on the canvas with the tag 'destroy'
    ##Draw Background
    home.create_rectangle(0,0,win_width,win_height,fill='#7f7f7f',tags='destroy') #set whole background to solid gray colour
```

Next, I drew the keyboard keys onto the canvas by iterating through each row in the keys array and then through every key in each of these rows. We also set counters, equal to x and y, indicating which row and element within that row the loop is at. Then for each key we calculate the top left x and y coordinate of the button by multiplying its counter by the button width/height (+ the margin + the x/y offset). Then we can find the bottom right coordinates of the button just by adding the button width/height. The program then creates a black rectangle on the canvas using these coordinates calculated for each key in the keyboard:

```

x = 0
for row in keys:
    y = 0
    for key in row:
        x1 = (y*(button_width+margin)) + x_offset
        y1 = (x*(button_height+margin))+ y_offset
        x2 = x1 + button_width
        y2 = y1 + button_height
        home.create_rectangle(x1,y1,x2,y2,fill='#000000',tags='destroy') #create all black keyboard keys

```

Then we can right the appropriate text onto the button – after checking the number of characters stored on the key – if greater than 1, will have a slightly smaller font size than a key storing one character. Once the text has been added to the button the x and y counters are incremented and the loop continues to the next key or row:

```

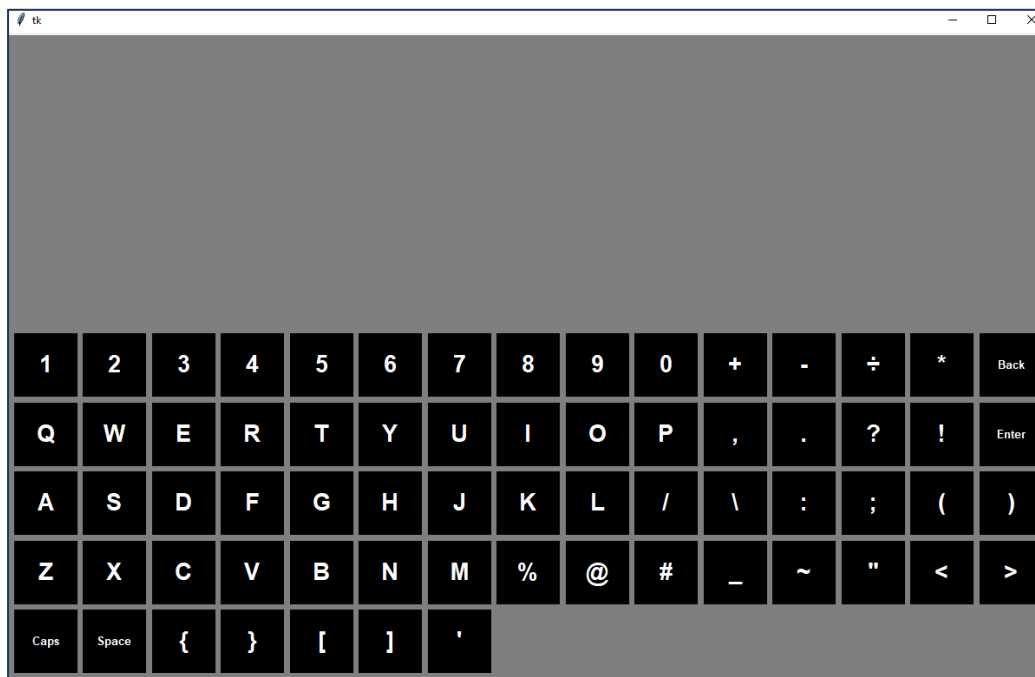
if len(key) > 1: #if key has more than a single character
    button_font = 'Ariel 10 bold'
else:
    button_font = 'Ariel 20 bold' #change font size if single letter on key

home.create_text(x1+button_width/2,y1+button_height/2, #draw text centered on each key
                fill='white',text='{}'.format(key.capitalize()),
                font=button_font,tags='destroy')

y += 1
x += 1

```

This is how the menu appears so far:

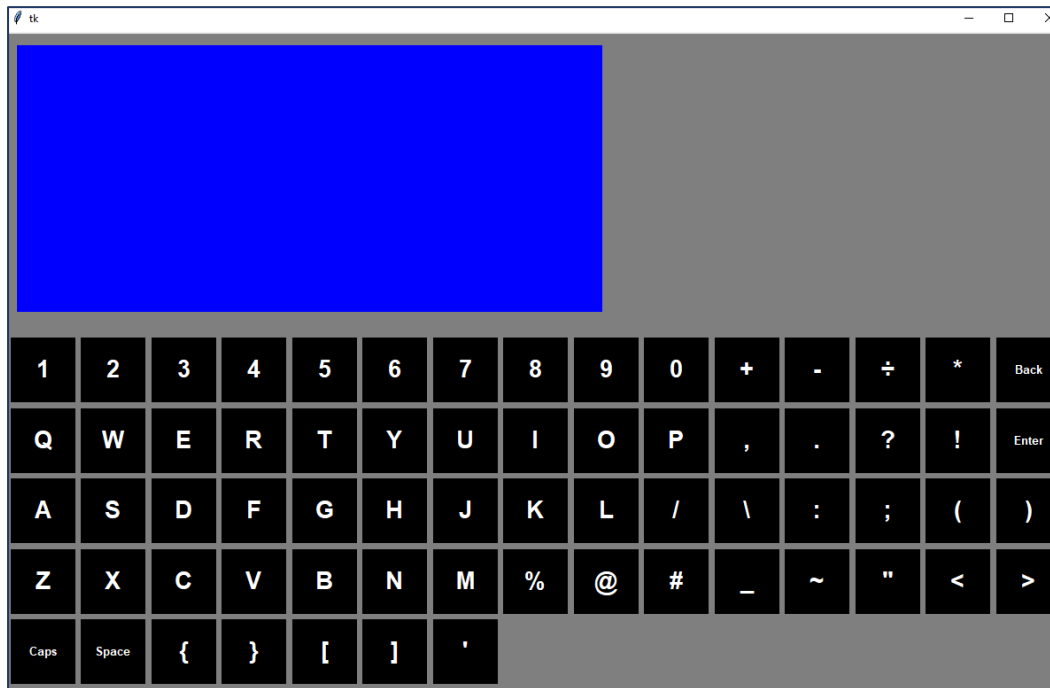


The text input box is created using another rectangle being drawn over the canvas in blue:

```

home.create_rectangle(15,15,675,315, fill='blue',outline='blue',tags='destroy') #create blue input box

```

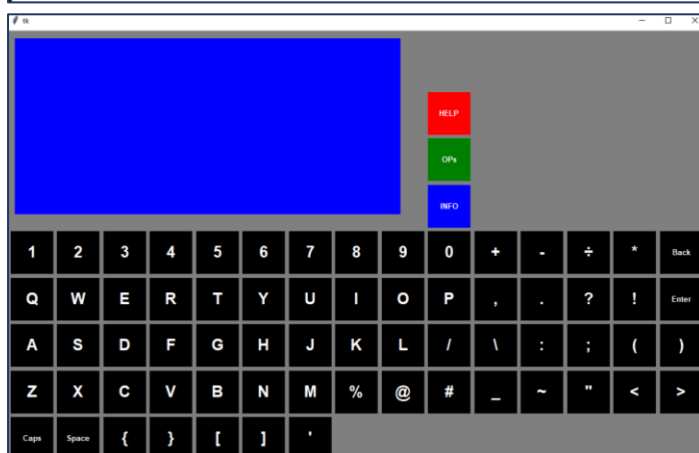


Next, I need to draw the other 3 buttons onto the canvas using rectangles – info, settings, word bank buttons – as the origin of the keyboard positions is at the top left of the key board, the help buttons are 1, 2 and 3 positions above that. Meaning -1, -2, -3 can be iterated through as the y coordinates of the buttons relative to the origin. The above method for finding the rectangle coordinates of the keyboard keys is then applied for the buttons, where x is replaced with 9 as help the buttons will be at the 9th value along each row in the keyboard. The button texts and colours can then as well be written onto the rectangle at each iteration:

```
for y in [-1,-2,-3]: #negative keyboard coordinate so the buttons show above the keyboard
    x1 = (9*(button_width+margin)) + x_offset
    y1 = (y*(button_height+margin)) + y_offset
    x2 = x1 + button_width
    y2 = y1 + button_height

    col = button_colours[y]
    home.create_rectangle(x1,y1,x2,y2,fill=col,outline=col,tags='destroy')#draw button with colour from button_colours list

    b_text = button_texts[y]
    home.create_text(x1+button_width/2,y1+button_height/2, #draw text on button from button_texts list
        fill='white',text=b_text,
        font='Ariel 10 bold',tags='destroy') #font size smaller to fit text on button
```

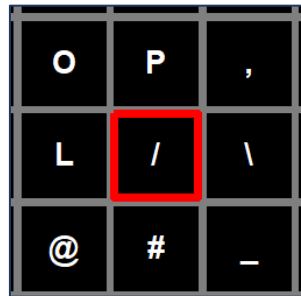


The key selector will be red border to each key that moves depending on which direction the user is looking. Key selection is a variable storing the coordinates of a key currently being selected, meaning the coordinates for the top left of the selector border will be the key selection variable coordinates times by the button width/height + the margin + the x/y offset. Whereas the bottom right should equal the top left coordinates + the button height/width:

```
x1 = (key_selection[1]*(button_width+margin)) + x_offset
y1 = (key_selection[0]*(button_height+margin)) + y_offset
x2 = x1 + button_width
y2 = y1 + button_height

home.create_line(x1,y1,x1,y2,x2,y2,x2,y1,x1,y1,fill='red',width=7)#create line with list of 5 coordinates making a square
#width = 7 so line has thickness around key
```

Create_line() is used to create a line surrounding the button using these 4 coordinates where the thickness is set to 7 to ensure the selector heavily borders the button so obvious to user which button they are selecting.



Next, we must draw the text being input by user onto the canvas using create_text () which stores the string output of the buttons onto the output box:

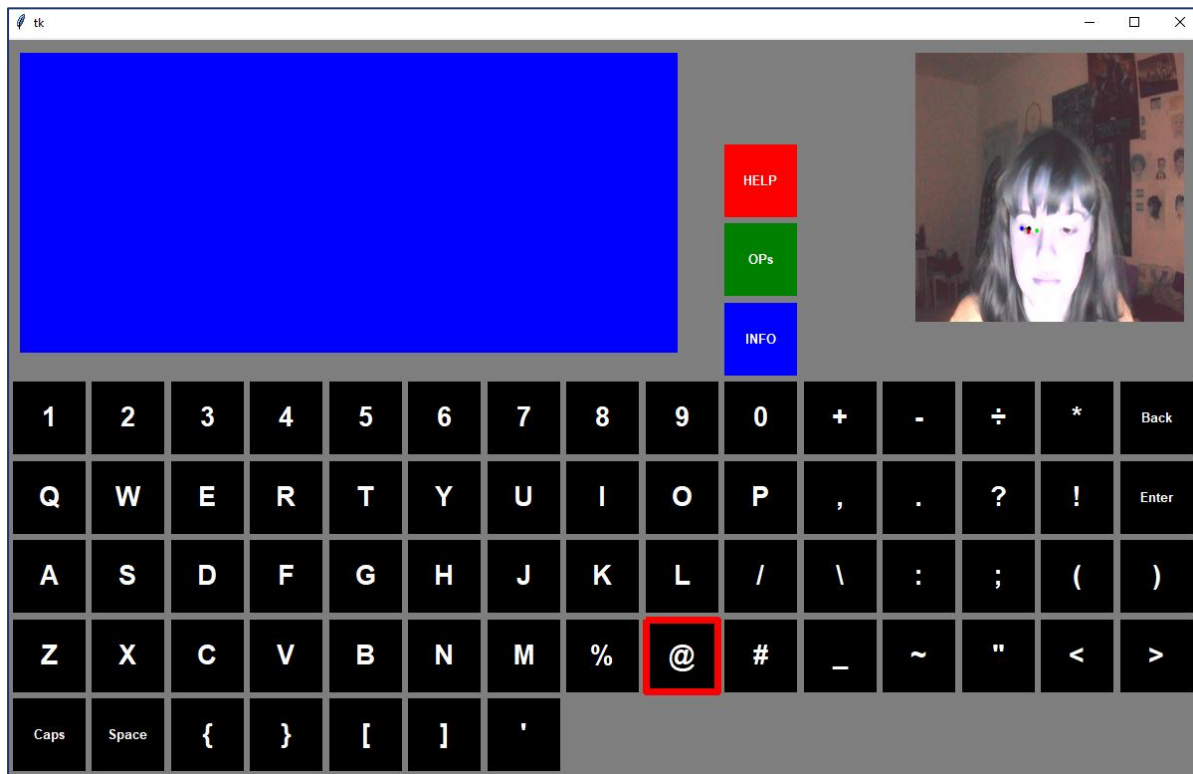
```
##Text Output
home.create_text(20,20,fill='white',text=output_string,font='Ariel 10 normal',#draw the string output_string onto the blue background
width=420,anchor='nw',tags='destroy')#starting at (20,20) and anchor north-west so text goes to the right
```

Finally, as we cannot convert a cv2 image to a photo image, in order to display the webcam feed onto the menu we need to convert the cv image array 'im' into a PIL image using image.fromarray(). I then need to resize the image using the variables defines outside the function. Then I can change the PIL image to a photo image which is the image format that can be displayed by tkinter. The webcam feed can now be output onto the menu using create_image() where it is anchored to the top right corner:

```
imagePIL=Image.fromarray(im) #converts opencv image to PIL image
imagePIL = imagePIL.resize((cam_width,cam_height)) #resize image
imagetk=ImageTk.PhotoImage(imagePIL) # Tkinter PhotoImage

home.create_image(1185,15,image=imagetk,anchor='ne',tags='destroy') #anchor image to top right corner
```

Here is the final menu display:



Now we have the menu function working we need to implement the move and write functions() the move functions is what will enable the user to move the selector left, right, up and down throughout the keyboard, wordbank and options window. As the info window only contains 1 button (home button) this window will not use the move function.

The write function will need to be changed from the initial design write function as we are no longer using tkinter element in the program. In addition to this, the write function is being applied to all three windows which will have different button grid, whereas in the design write, it was only being applied to the keyboard window. Therefore in the write function we need to parse the output string and the current window so the function can determine which button grid needs to be checked. For all the buttons on the keyboard window, there are two possible outcomes: the output string has a character added to it or the current window is changed. Therefore we return the output string and the new current window from write function. Inside the write function, first we must check which window is currently being rendered so we can determine which set of buttons are being used – as the grid of buttons for each window will be different. The keyboard window contains two sets of buttons, the keys – adds text and changes the format of text being added to the output box, and the window buttons – changes the current window being rendered. Therefore after determining whether the current window is the keyboard window, we must check whether the button being selected is in button_positions (which is the window holding the window button positions). If it isn't in button_positions the user is selecting a key on the keyboard, so we can either add the value stored in keys with the key_selection index to the output string or alter the output string using CAPSLOCK, ENTER, SPACE and BACK commands:

```

if curr_win == 'home': #checks whether current
    if key_selection not in button_positions:
        key_pressed = keys[key_selection[0]][key_selection[1]] #get letter in keys array of keyboard coordinates
        if key_pressed == 'CAPS': #if key is CAPS key, toggle CAPSLOCK variable
            CAPSLOCK = not CAPSLOCK
        elif key_pressed == 'ENTER': #write new line for ENTER
            out_string = out_string + '\n'
        elif key_pressed == 'SPACE': #write a space if SPACE
            out_string = out_string + ' '
        elif key_pressed == 'BACK': #sets string equal to whole string minus last letter
            out_string = out_string[0:-1]
        else:
            out_string = out_string + keys[key_selection[0]][key_selection[1]] #for any normal key, just add the character to the output_string

```

If else and the button being pressed is in button_positions, then depending on which element in button_positions is being pressed the current window is changed:

```

else:
    #if the key_selection is one of the buttons to another window, change the current_window
    if key_selection == button_positions[1]: #1
        curr_win = 'settings'
    if key_selection == button_positions[2]: #2
        curr_win = 'info'
    if key_selection == button_positions[0]:
        curr_win = 'wordbank'
return out_string, curr_win

```

Now we have complete the write function for the keyboard window (this function will be added to when we come to make the wordbank window), we can move on to the move function.

To determine the position of the next button the user is meant to move to, based off their gaze direction, the program should require the users gaze direction, the type of button position grid and the position of the key the selector is currently on. For the keyboard window, the button position grid is the key_positions array, whereas for another window, as the buttons will be different to the buttons on the keyboard window, the button position grid parsed to the function will also be different. If the key the user is trying to look at as a position stored in the button position grid then the selected key is changed to that value:

```

def move(di, key_sel, pos):
    if di == 'left':
        if [key_sel[0],key_sel[1]-1] in pos: #check if the key to move to is an actual position in position grid
            key_sel[1] -= 1 #moving left so x position changed
    elif di == 'right':
        if [key_sel[0],key_sel[1]+1] in pos:
            key_sel[1] += 1

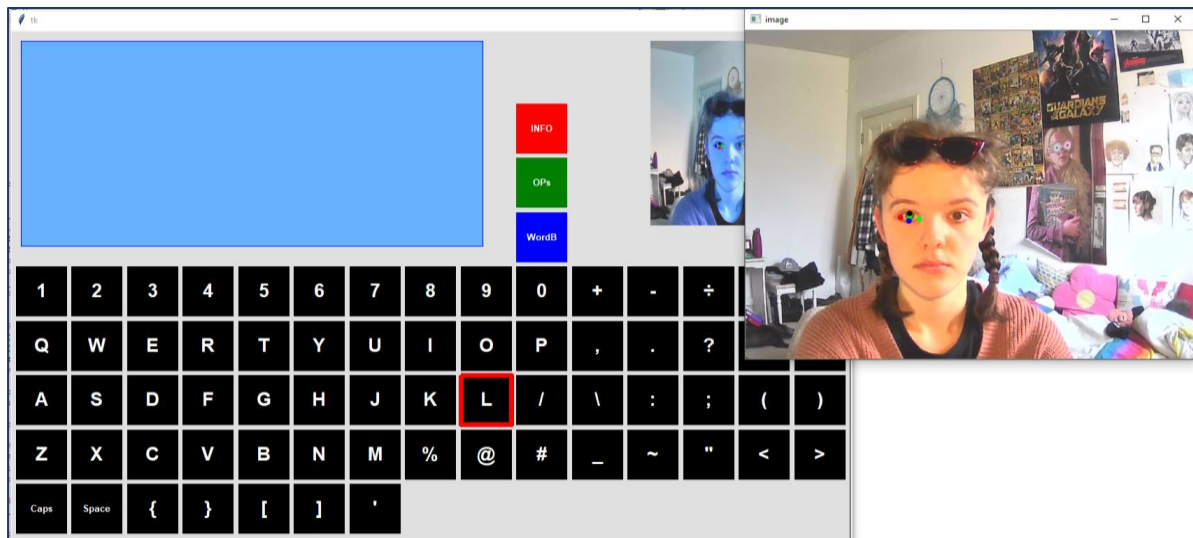
```

```

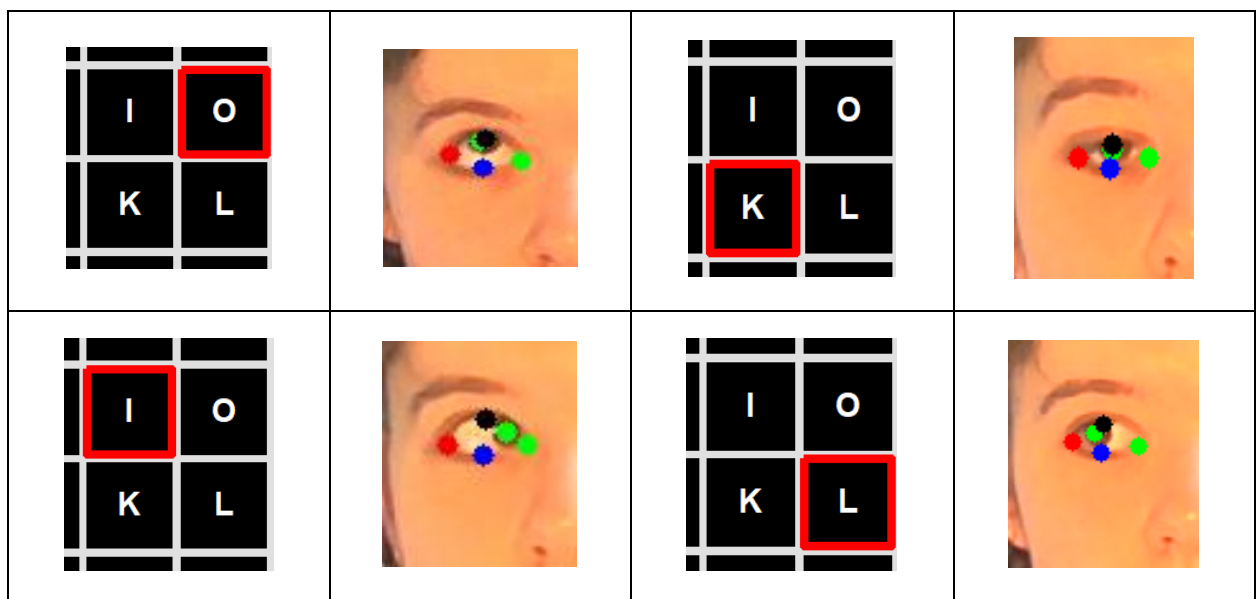
elif di == 'down':
    if [key_sel[0]+1,key_sel[1]] in pos:
        key_sel[0] += 1 #moving down so y position changed
elif di == 'up':
    if [key_sel[0]-1,key_sel[1]] in pos:
        key_sel[0] -= 1
return key_sel

```

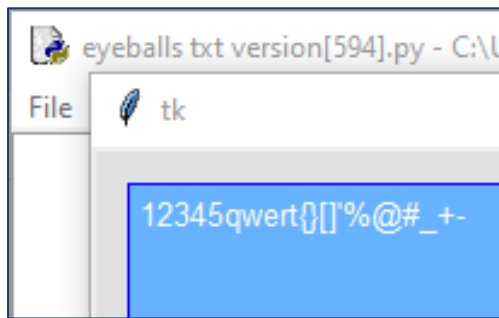
Now we have the move and write functions fully implemented, we can quickly test their functionality:



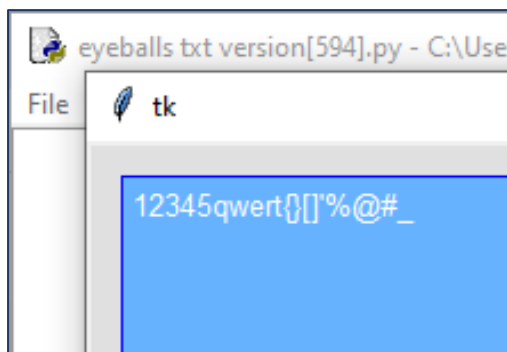
The key selectors automatic position is 2, 8 which is 'L', when I look up, then left then down, then right, the selector ends up back on the initial position meaning it works correctly in all directions:



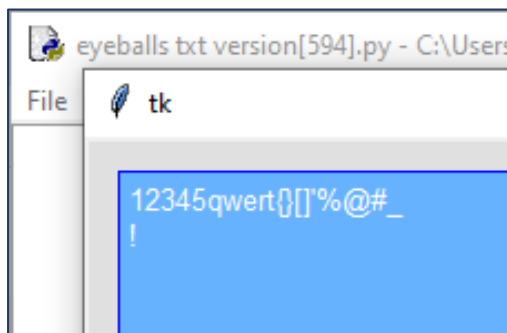
For the write function the user can type letters into the input box as well as capitalize, change line and remove text:



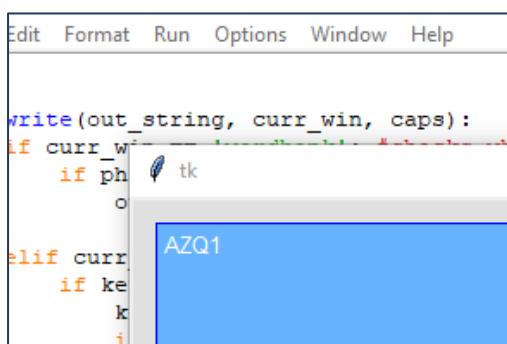
From testing multiple keys from various different locations on the keyboard, the write function is working very well at appending characters to the output string.



BACK is also working effectively as it was used to remove the + and – element from the output string.



ENTER is also working correctly as it was used to add a new line to the output which an exclamation mark was written onto.



CAPS can also capitalise the text being input to the output box - It also correctly reverts the output back to lowercase when pressed a second time.

Therefore, all buttons on the keyboard seem to be working correctly meaning so now we can move onto designing the three other windows.

Lastly, I am going to add another button that clears the output string for the user, so they don't have to continuously blink on the back button to remove text. First, I need to add the key 'CLEAR' to the keys array and then inside the write function, if the current window is home and the selected key is CLEAR, the output string is an empty string.

Info, options, word bank window:

Info window

For the info window function, as there is only one button, which will be the home button in the top left corner of the window, we do not need to define any variables outside the function. This also means the write function will not need to use the write function, meaning that if the user blinks, it can only mean one thing – the user has selected the home button, so therefore if a blink is detected and the current window is the info window, the current window is changed to the home window:

```
if diff_time < 4 and diff_time >= 1:
    print('click') #would evoke the mouse to click if the blink hold is < 2 and > or equal t

if current_win == 'info':
    current_win = 'home'
```

The move function will not be used on the info as the user shouldn't have to move a selector between any buttons as there is only one. Now we can create the info window function:

Initial implementation of the info window function is exactly the same as the home window function: update, delete window with each frame, and set background to solid grey, however I changed the background colour of both windows to a lighter grey so it appeared more like the initial design mock-ups of the windows. On the info, options and word bank windows, they each have a banner containing the home button and the name of the window at the top, therefore using `create_rectangle()` I added the banner along the top of the window. The home button dimensions are the `button_width` x `button_height`, therefore the blue rectangles height was set to `button_height + 14`. This is because this banner is present on all windows except home, and both the word bank and options window will contain the red button selector, seen in the home window with a width of 7, meaning the banner should account for this additional 14 pixels so nothing is out of frame. The home button is also set 7 pixels away from the x axis for the exact same reason.

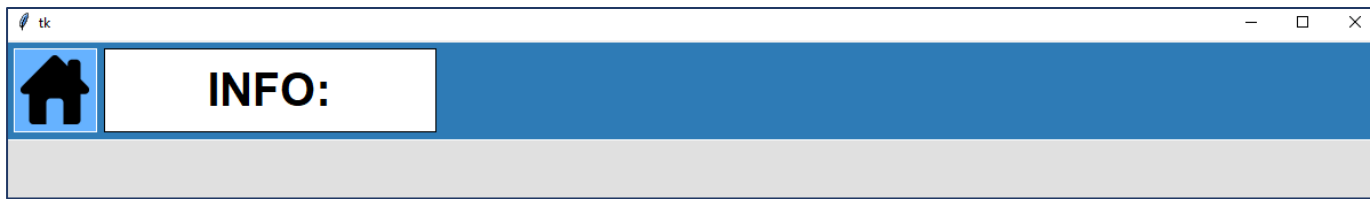
Outside of the info function, the home button image is opened, resized and converted to a photo Image format that tkinter can display:

```
home_image_button = Image.open('home_image_button_large.png') # opens png
home_image_button.mode = 'RGBA' # maintain transparency of image (A is alpha channel)
home_image_button = home_image_button.resize((60,60)) # resize it to desired size here
home_image_button = ImageTk.PhotoImage(home_image_button) # convert to tkinter image
```

The `home_image_button` is then displayed in the info window within the home button and the 'INFO: 'header is also displayed within the bank through the implementation of an additional `create_rectangle()` and `create_text()` element:

```
#Blue Bar and Home Button
home.create_rectangle(0,0,win_width,button_height+14, fill= '#2E7BB6', outline = 'white', tags='destroy')
home.create_rectangle(7,7,button_width+7,button_height+7,fill='#66B2FF',outline = 'white', tags='destroy')
home.create_image(13,13,image=home_image_button,anchor='nw')
#Window Heading 'info'
home.create_rectangle(button_width+14,7,button_width+304,button_height+7,fill='white',tags='destroy')
home.create_text(button_width+159,(button_height+14)/2,fill='black',font='Ariel 30 bold',
                 text='INFO:',tags='destroy')
```

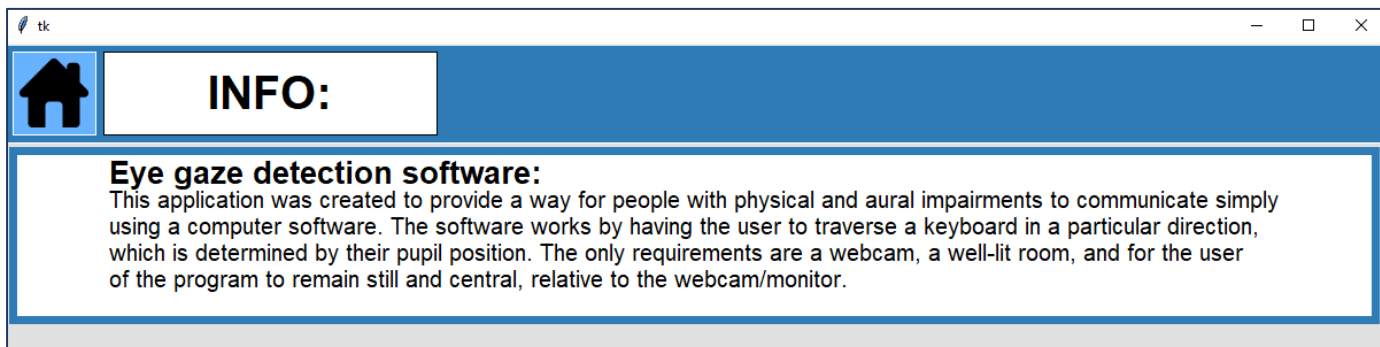
Info window banner:



Now I need to implement the top text box in the input table, which will contain a message explaining the purpose of the software as well as (very briefly) how it works.

The text will read as: 'This software was created to provide a way for people with physical and aural disabilities to communicate simply using a computer software. The software works by having the user to traverse a keyboard in a particular direction, which is determined by their pupil position. The only requirements are a webcam, a well-lit room, and for the user of the program to remain still and central, relative to the webcam/screen.'

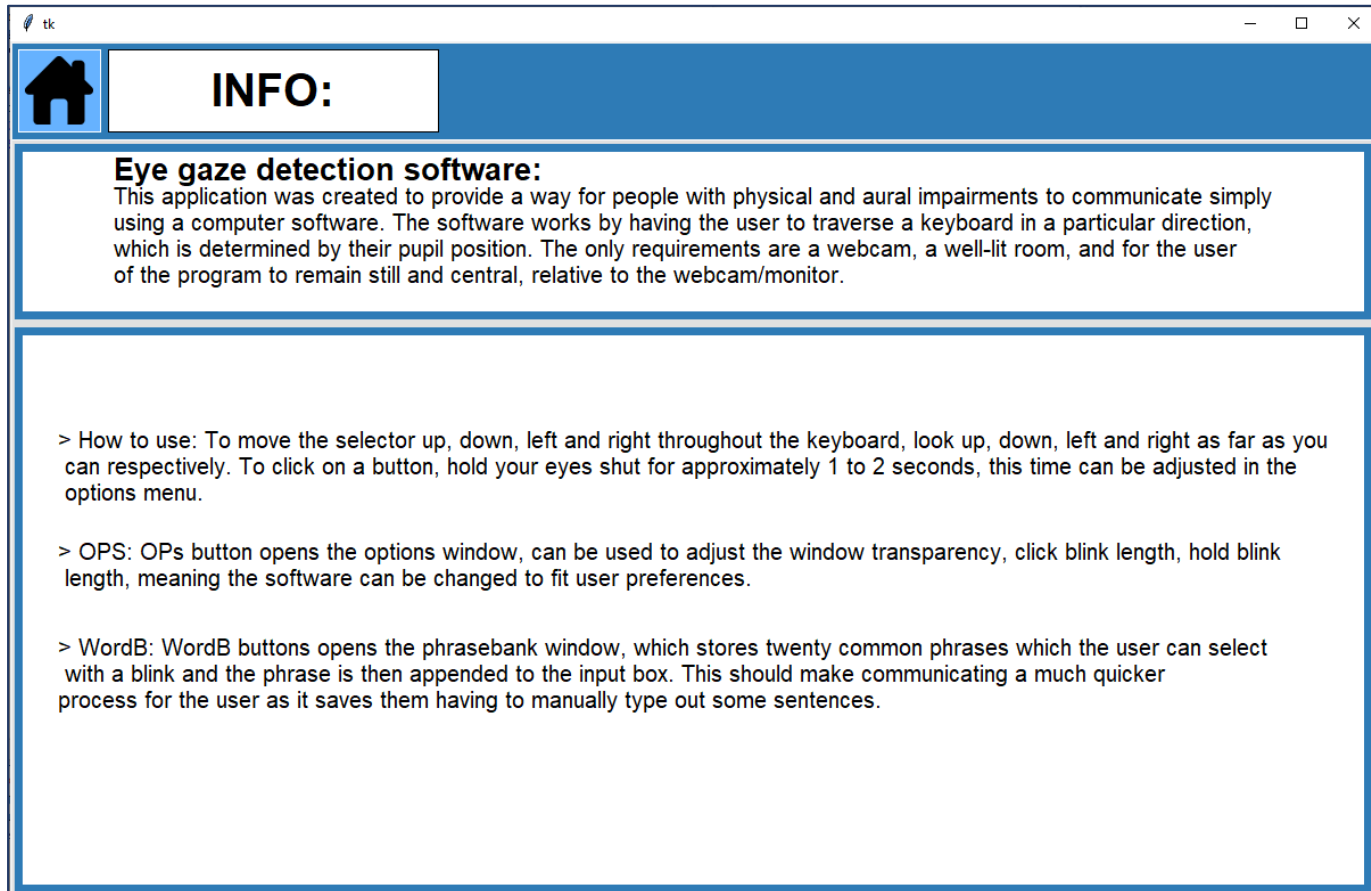
```
home.create_rectangle(7, button_height+21, win_width-7, 240, fill = 'white', outline = '#2E7BB6', width = 7, tags = 'destroy')
home.create_text(5*7, button_height+3.5*7, #draw text centered on each key
                fill='black',text='Eye gaze detection software:',
                font='Ariel 20 bold', anchor = 'nw', tags='destroy')
home.create_text(win_width/2, button_height+14*7,
                fill='black', text='This software was created to provide a way for people with physical and aural impairments
                font='Ariel 15', tags = 'destroy')
```



Finally, we need to add the lower text box which should contain three points. Initially, one of those points was on how to calibrate the software. However, now that the software no longer has a calibration window, the three points will be on how to use the software, the options window, and the word bank window:

- How to use: 'To move the selector up, down, left and right throughout the keyboard, look up, down, left and right as far as you can respectively. To click on a button, hold your eyes shut for approximately 1 to 2 seconds, this time can be adjusted in the options menu.'
- OPS: 'OPs button opens the options window, which can be used to adjust the window transparency, blink length, and hold length, meaning the software can be changed to fit user preferences.'
- WordB: 'WordB buttons opens the phrase bank window, which stores twenty common phrases which the user can select with a blink and the phrase is then appended to the input box. This should make communicating a much quicker process for the user as it saves them having to manually type out some sentences.'

```
#lower text box on info window
home.create_rectangle(7, 240 + 14, win_width - 7, win_height - 7, fill = 'white', outline = '#2E7BB6', width = 7, tags = 'destroy')
home.create_text(6*7, 240+14*7, fill = 'black', text = '> How to use: To move the selector up, down, left and right throughout the k
font = 'Ariel 15', anchor = 'nw', tags = 'destroy') #adds text on how to use software
home.create_text(6*7, 240+28*7, fill = 'black', text = '> OPS: OPs button opens the options window, can be used to adjust the window
font = 'Ariel 15', anchor = 'nw', tags = 'destroy') #adds text on options window
home.create_text(6*7, 240+40*7, fill = 'black', text = '> WordB: WordB buttons opens the phrasebank window, which stores twenty commu
font = 'Ariel 15', anchor = 'nw', tags = 'destroy') #adds text on word bank window
```



Finally, for the info window to run, we must set a condition at the end of the program that checks whether the current window is 'info'. If it is, the info window function is called – the same condition is implemented for all windows in the program:

```
#determines which window is to be rendered
if current_win == 'info':
    info_window()
```

Options

the options window has its own set of buttons – initially there was 6 but now there is 4, as the hold feature is being removed from the code, as its only purpose was so the user could control the scroll bars. However so far, we haven't needed to add any scroll bars and the based off window dimension, for the word bank window, fitting 20 short phrases into the space given should work.

The 4 buttons include the select length increase button, the select length decrease button, the window transparency increase button and the window transparency decrease button. The options button positions and names will need to be stored in two separate arrays. The options buttons

positions array will contain a fifth position for the home button [-1, 0], meaning its position is above the origin of the positions array, which would be the select length increase button. In addition to the name and grid positions of the initial select length and window transparency must be set:

```
#options variables
SELECT_LENGTH = 0.5 #sets initial select length to 0.5

TRANSPARENCY = 1 #sets initial window transparency to 1 - opaque

options_buttons = [['sel_len_up','sel_len_down'], ['trans_up','trans_down']] #set the options button names
options_button_positions = [[0,0],[0,1],[1,0],[1,1],[-1,0]] #sets option buttons + home button positions
```

Then we need to set the initial button selection position, which will be at position (1, 0):

```
options_button_selection = [0,1] #sets initial position of option buttons selector
```

As the options window doesn't write anything to the output box, it will not utilise the write function. When a click is evoked and the current window is the options window, in order to check through the options_buttons array we must find the name of the current button being pressed in the options_buttons array, which can be done by using the coordinates of the options_button_selector as index values:

```
elif current_win == 'settings':
    #finds the button being pressed in options_buttons array by using the options selector coordinates
    button_clicked = options_buttons[options_button_selection[0]][options_button_selection[1]]

    if options_button_selection == options_button_positions[-1]:
        current_win = 'home' #renders home window if button selector coordinates = home button coords

    elif button_clicked == 'sel_len_up':
        if SELECT_LENGTH + 0.2 <= 5:
            SELECT_LENGTH += 0.2 #adds 0.2 to the select length if the total select length <= 5 sec

    elif button_clicked == 'sel_len_down':
        if SELECT_LENGTH - 0.2 >= 0.4:
            SELECT_LENGTH -= 0.2 #minuses 0.2 from select length if total select length >= 0.4 sec

    elif button_clicked == 'trans_up':
        if TRANSPARENCY + 0.1 <= 1:
            TRANSPARENCY += 0.1 #adds 0.1 to transparency if the total transparency <= 1

    elif button_clicked == 'trans_down':
        if TRANSPARENCY - 0.1 >= 0.3:
            TRANSPARENCY -= 0.1 #minuses 0.1 from transparency if total transparency >= 0.3

elif current_win == 'home':
    output_string, current_win, CAPSLOCK = write(output_string, current_win, CAPSLOCK)
    time.sleep(1)
```

However, as the settings window does contain grid of buttons, therefore if the current window equals the settings window, then the move function can be applied to the options_button_positions array, enabling the user to move between the 5 buttons on the window – like in the home window:

```
elif current_win == 'settings' and stat == False:
    options_button_selection = move(direction, options_button_selection, options_button_positions)
```

Now we have sorted out the movement and button controls within the options window we can start implementing the options window function.

```
def options_window():
    home.update() #update the canvas every frame
    home.delete('destroy') #delete everything on the canvas with the tag 'destroy'
    #Draw Background
    home.create_rectangle(0,0,win_width,win_height,fill='#E0E0E0',tags='destroy') #set whole background to solid gray colour
    #Blue Bar and Home Button
    home.create_rectangle(0,0,win_width,button_height+14, fill= '#2E7BB6', outline = 'white', tags='destroy')
    home.create_rectangle(7,7,button_width+7,button_height+7,fill='#66B2FF',outline = 'white', tags='destroy')
    home.create_image(13,13,image=home_image_button,anchor='nw')
    #Window Heading 'Options'
    home.create_rectangle(button_width+14,7,button_width+304,button_height+7,fill='white',tags='destroy')
    home.create_text(button_width+159,(button_height+14)/2,fill='black',font='Ariel 30 bold',
                    text='OPTIONS:',tags='destroy')
```

The code above is copied from the info window as it is the section of code used update/delete the window every frame, as well as render the window header, which is present in the three extra windows – the text present in the header is changed to OPTIONS:



Then we need to create the white box containing the options buttons and explanation text, which can be done using another create_rectangle() function:



The rectangle uses the same colours and thickness (7) as the text boxes shown in the info window – this ensures consistency across all windows. Next, we can add the text to the box, with 2 text elements naming the aspect of the software that the buttons will affect (transparency and select length), with the other two explaining the buttons incrementation so the user knows how much each blink affects the transparency/select length:

```
#Box with options in it
home.create_rectangle(margin,button_height+21,win_width - margin,win_height - margin,fill='white',outline='#2E7BB6',width=7,tags='destroy')
#Options texts
home.create_text(150,350,anchor='w',fill='black',font='Ariel 20 bold',
                text='SELECT LENGTH:',tags='destroy')
home.create_text(150,500,anchor='w',fill='black',font='Ariel 20 bold',
                text='TRANSPARENCY:',tags='destroy')
home.create_text(800,320,anchor='nw',fill='black',font='Ariel 13 bold',
                text='increases/decreases duration of blink selection by +/- 0.2 seconds',
                width=300,tags='destroy')
home.create_text(800,470,anchor='nw',fill='black',font='Ariel 13 bold',
                text='increases/decreases transparency of window by +/- 0.1',
                width=300,tags='destroy')
```

Next, we need to create the arrows buttons – as there are only 4 buttons in the grid, we can just draw the 4 triangles (representing up and down arrows) using the create_polygon() function:

```

x1,y1 = 500,380
home.create_polygon(x1,y1,x1+100,y1,x1+50,y1-86,fill='#66B2FF',outline='#2E7BB6',tags='destroy')
x1,y1 = 500,530
home.create_polygon(x1,y1,x1+100,y1,x1+50,y1-86,fill='#66B2FF',outline='#2E7BB6',tags='destroy')
x1,y1 = 650,300
home.create_polygon(x1,y1,x1+100,y1,x1+50,y1+86,fill='#66B2FF',outline='#2E7BB6',tags='destroy')
x1,y1 = 650,450
home.create_polygon(x1,y1,x1+100,y1,x1+50,y1+86,fill='#66B2FF',outline='#2E7BB6',tags='destroy')

```

The text positions and x1, y1 positions were all found through trial and error to determine which values best fitted the window.

Next, we need to draw on the button selector – first we need to find the top left coordinate of buttons bounding square which can be found easily as the arrow triangles are equilateral. As the coordinates of x1 and y1 are the left vertex of the triangle, the x coordinate of top left corner of the bounding square for the bottom two buttons is the same as x1 and y1. Whereas for the top two buttons, the top left corner coordinate is x1-100 and y1 – 100 as 100 is the height of the triangle. To make it easier, I will create an array outside the options window function storing the screen positions of the buttons:

```
options_buttons_screen_locations = [[500,290],[650,290],[500,440],[650,440],[7,7]]
```

Now we can find the top left coordinates of the selected buttons bounding square. Then we must check which button is being selected as the options buttons are larger than the home button, meaning the selector size will be different. Therefore, we can check if the options_button_selector equals the options_buttons[-1] which is the home position. If it is equal to home position x2 and y2 just equal x1 + button_width and y1 + button_height. If the position isn't equal to home position, then x1, y2 equal x1+100, y1+100. We can then draw the selector with these coordinates:

```

#Draw button selector
x1,y1 = options_buttons_screen_locations[options_button_positions.index(options_button_selection)]
if options_button_selection == options_button_positions[-1]:
    x2,y2 = x1+button_height,y1+button_height
else:
    x2,y2 = x1+100,y1+100
home.create_line(x1,y1,x1,y2,x2,y2,x2,y1,x1,y1,fill='red',width=7)

```

Finally, we must set the condition at the end of the while loop that if the current window is equal to settings, the options window function is run:

```

elif current_win == 'settings':
    options_window()

```

Word bank

like the options window, the word bank window also has its own set of buttons, each of which contain a phrase so when selected, that phrase is output to the output box in the home screen. This means we need to create two arrays storing the phrases and the phrase positions. The 20 different phrases I have decided to include in the word bank window are: 'Hello', 'How are you?', 'I am good', 'My name is', 'Thank you', 'Yes please', 'No thank you', 'What is your name?', 'Can I have', 'Explain', 'I like it', 'I don't like it', 'Stop', 'I am feeling', 'I agree', 'I disagree', 'What', 'Why', 'When', 'Where'. These are very short, but helpful phrases for the user to have quick access to as they aren't limited in

meaning and are very common in normal speech. I will write these phrases into a 2d array called phrases. I will use the same method of using a nested for loop to iterate through the keys name array to find the key positions used in the home window, but it will be applied to the phrases array to create a phrase positions array:

```
#wordb variables:
phrases = [['Hello', 'How are you?', 'I am good', 'My name is'],
            ['Thank you', 'Yes please', 'No thank you', 'What is your name?'],
            ['Can I have', 'Explain', 'I like it', "I don't like it"],
            ['Stop', 'I am feeling', 'I agree', 'I disagree'],
            ['What', 'Why', 'When', 'Where']]
phrase_positions = [] #stores x and y positions of all the phrases in the wordbank
for i in phrases:#the 'Hello' phrase is [0,0]
    for j in i:#this loop creates these coordinates
        phrase_positions.append([phrases.index(i),i.index(j)])
home_position = [-1, 0]
phrase_positions += home_position
```

In addition to the phrase buttons, the home button position can be appended to the phrase positions array. The phrase bank window will contain a header meaning the window space available will have a width of the windows width – 2*x offset which will equal 1185, and a height of the windows height – (button_height + 14) – 3*x off set which will equal 657. As there should be an offset from the edge of the window the x offset will equal 7 and will be act as the margin in the word bank window. The y offset just equals the banner height - (button height + 14) – plus the x offset:

```
rows = 5
y_offset_wb = button_height + 21
x_offset_wb = 7
phrase_height = 131.4
phrase_width = 296.25
phrase_selection = [-1, 0]
```

The phrase height is found by dividing the available window height, below the header, by 5:

$$- \quad 657 / 5 = 131.4$$

The phrase width is found by dividing the available window width by 4:

$$- \quad 1185 / 4 = 296.25$$

As the user is both using the word bank window to write into the text box and using a selector to move around buttons within the window, both the write() and move() functions can be applied to the word bank window.

When a blink is detected, if the current window is equal to the word bank window, then the write function can be called:

```
elif current_win == 'wordbank':
    output_string, current_win, CAPSLOCK = write(output_string, current_win, CAPSLOCK)
    current_win = 'home'
```

Inside the write function, after determining whether the current window is word bank or not, we need to implement another if statement to test whether the button being selected is the home button by checking if the button being selected is the home button. If it isn't the phrase in the button is added to the output string and is returned from the function. The current window is then changed to home after the function call so the user can see the phrase displayed in the output box:

```
def write(out_string, curr_win, caps):
    if curr_win == 'wordbank': #checks whether current window is wordbank
        if phrase_selection != options_button_positions[-1]: #if selector phrase not home button
            out_string = out_string + phrases[phrase_selection[0]][phrase_selection[1]] #add phrase to output string
```

Now we have sorted output the text to the menu, we can apply the write function to the word bank window. This will be done exactly the same way as in the home and setting window, but instead parsing the phrase_positions array and the phrase selection coordinates:

```
elif current_win == 'wordbank' and stat == False:
    phrase_selection = move(direction, phrase_selection, phrase_positions)
```

Now we have sorted the movement of the selector in the word bank window, as well as the writing of the phrases to the output box in the menu window, we can move onto creating the word bank window function.

```
def wordb_window(phrases):
    home.update() #update the canvas every frame
    home.delete('destroy') #delete everything on the canvas with the tag 'destroy'
    #Draw Background
    home.create_rectangle(0,0,win_width,win_height,fill='#E0E0E0',tags='destroy') #set whole background to solid gray colour
    #Blue Bar and Home Button
    home.create_rectangle(0,0,win_width,button_height+14, fill= '#2E7BB6', outline = 'white', tags='destroy')
    home.create_rectangle(7,7,button_width+7,button_height+7,fill='#66B2FF',outline = 'white', tags='destroy')
    home.create_image(13,13,image=home_image_button,anchor='nw')
    #Window Heading 'Options'
    home.create_rectangle(button_width+14,7,button_width+304,button_height+7,fill='white',tags='destroy')
    home.create_text(button_width+159,(button_height+14)/2,fill='black',font='Ariel 30 bold',
                    text='WORDBANK:',tags='destroy')
```

Firstly, the header code, present in all of the extra windows (info, options and wordbank) is integrated into the word bank window, and the text is changed to 'WORDBANK'.

Next, we need to draw the grid of buttons containing all of the phrases, which is done using exactly the same method as in the home window – using an embedded for loop iterating through every element in each row of the phrases array to find the coordinates of the top left corner of the phrase button by multiplying the x/y counters by the button height/width and then adding the y/x offset to this value. The bottom right coordinates should therefore just each the top left x, y coordinates plus the phrase width and height:


```

x = 0
for row in phrases:
    y = 0
    for phrases in row:
        x1 = (y*(phrase_width)) + x_offset_wb
        y1 = (x*(phrase_height)) + y_offset_wb
        x2 = x1 + phrase_width
        y2 = y1 + phrase_height

```

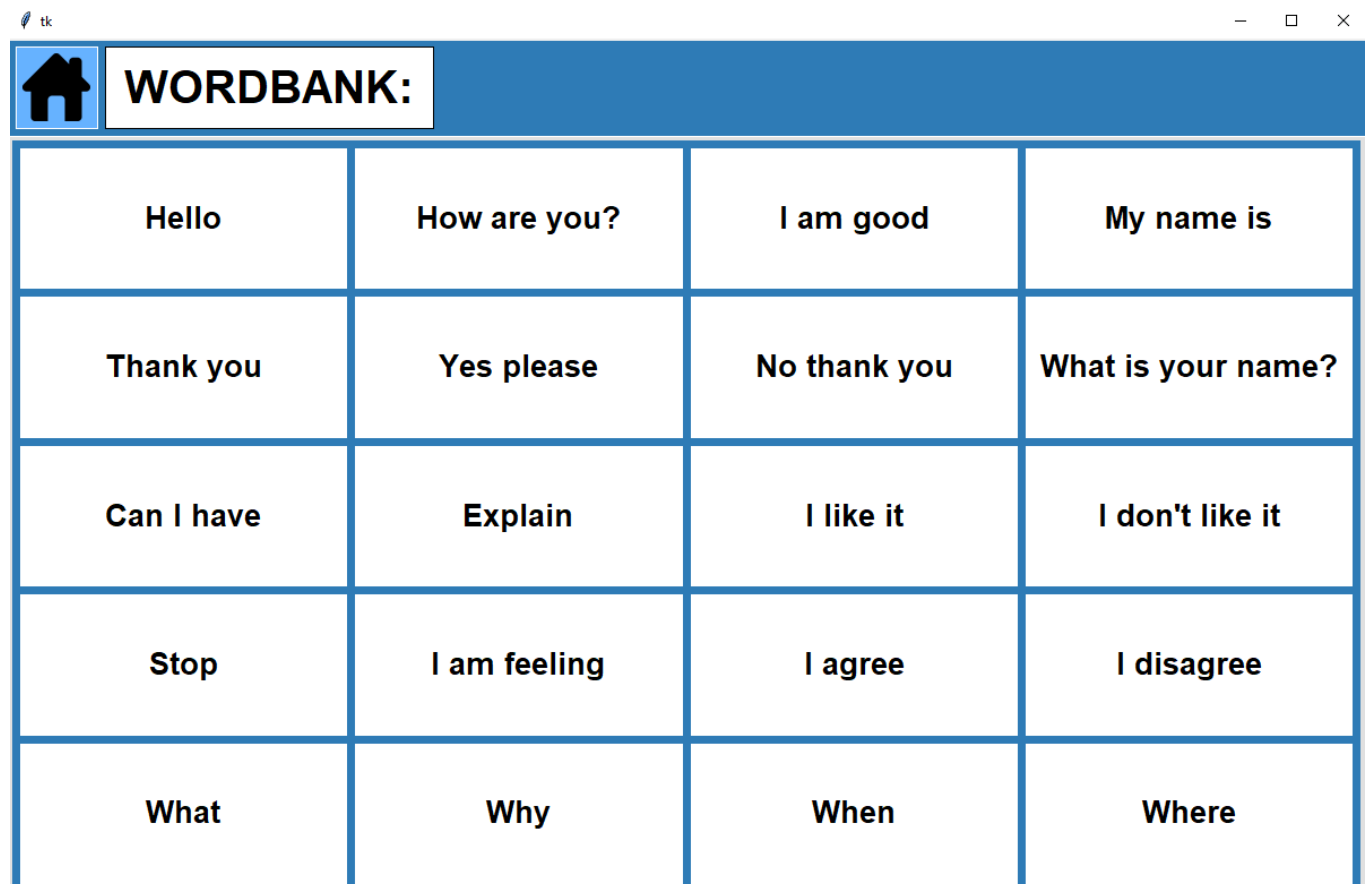
Then within the embedded for loop we can create the button using `create_rectangle()` function and place the phrase text inside it using `create_text()` function, for each iteration:

```

home.create_rectangle(x1,y1,x2,y2,fill='white', outline = '#2E7BB6', width = 7, tags='destroy')
home.create_text(x1+phrase_width/2,y1+phrase_height/2, #draw text centered on each key
                 fill='black',text='{}'.format(phrases), font='Ariel 20 bold',tags='destroy')
y += 1
x += 1

```

X and y are incremented at the end of each iteration of the loop they were initially defined so the phrase button rectangles can be draw for the next element/row:



Now we need to add the selector in, and like in the options window, the home button has different dimensions to the phrase buttons. So first we must check is the selected button in `phrase_position` array equals the last set of coordinates – which is the home button positions. If it does equal the home position then the bounding rectangle will have a width and height equal to the `button_width` and `button_height` variables defined outside the function. The top left corner coordinates of the home buttons bounding rectangle is equal to (7, 7).

If the position isn't the home position, then the top left-hand corner of the selector rectangle has an x coordinate equal to the x coordinate of the selected button*phrase_width plus the x offset. The y coordinate will therefore be equal to the y coordinate of the selected button*phrase_height plus the y offset. The bottom right coordinate will be equal the top left corner coordinates plus the phrase width/height:

```
if phrase_selection == [-1,0]:
    home.create_line(7,7,button_width+7,button_width+7,button_height+7,button_height+7,fill='red',width=7)#create line with list of 5 coordinates
else:
    x1 = (phrase_selection[1]*(phrase_width)) + x_offset_wb
    y1 = (phrase_selection[0]*(phrase_height)) + y_offset_wb
    x2 = x1 + phrase_width
    y2 = y1 + phrase_height
    home.create_line(x1,y1,x2,y1,x2,y2,x1,y2,x1,y1,fill='red',width=7)#create line with list of 5 coordinates making a square
```



Now we have successfully implemented the wordbank window the final stage is to set the condition at the bottom of the program that if the current window equals word bank, then the word bank function is called.

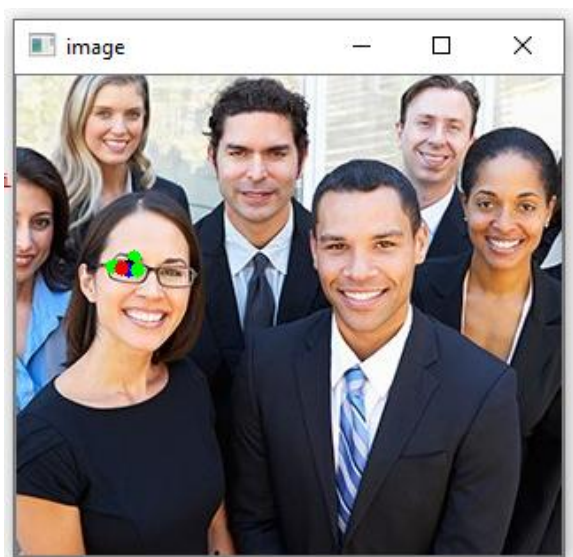
Now we have fully implemented the word bank windows function I am going to change the pupil detection algorithm so it only detects the pupil of the largest face in frame. To do this, outside of the loop iterating through the faces detected in the frame I create two empty arrays called face_size and tot_landmarks. For every face in frame the face height * the face width is multiplied by each other and stored in the face_size array. In addition to this, the landmarks of the current face being detected in the for loop is stored in the tot_landmarks array. Once all detected faces have been iterated through, the landmarks to be used in the pupil detection are set equal to the landmarks with the same index as the maximum value stored in face_size – as this should be the largest face in

frame. If there is not face detected in the frame (aka if face_size equals an empty array after the loop), the program will just continue and move onto the next frame:

```
face_size = []
tot_landmarks = []

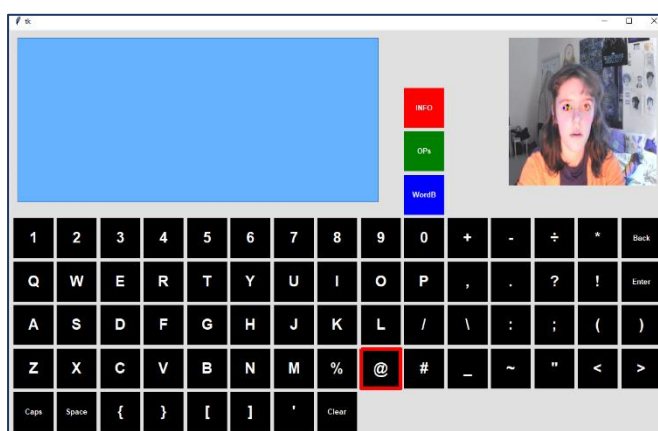
for face in face: #iterates through detected faces
    landmarks = (face_utils.shape_to_np(predictor(gray_im, face))).tolist() #predicts landmarks on face
    size = (landmarks[17][0]-landmarks[1][0])
    face_size.append(size)
    tot_landmarks.append(landmarks)

if face_size == []: #if no face detected the program continues onto next frame
    continue
else:
    landmarks = tot_landmarks[face_size.index(max(face_size))]
```

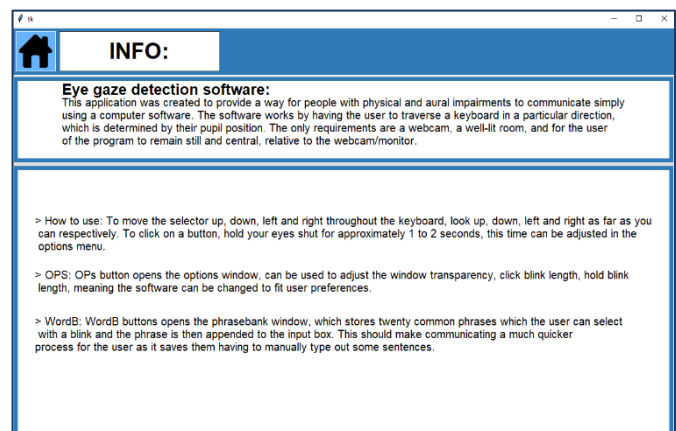


As the image shows, the face that appears the closest to the camera has had their pupil detected – as shown by the set of coloured dots on the image – meaning this method of detecting the closest face in frame works correctly. This means that if multiple faces are detected in frame, only one person's pupil will be used to determine the direction in which the selector will move.

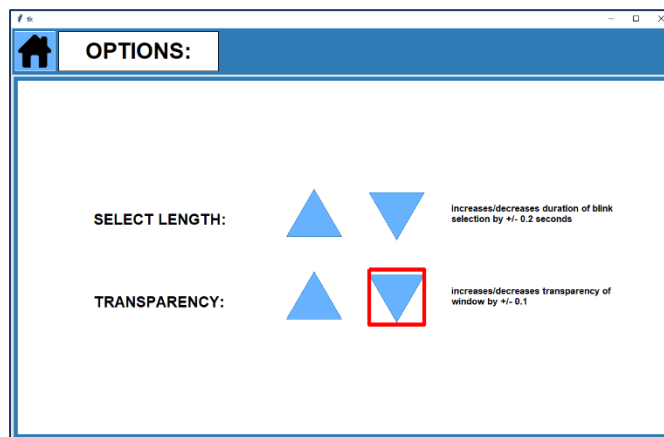
Lastly, I changed the colours of the home window background and input box so it matched more with the other three windows, to ensure the software looked consistent. These are the final versions of the windows:



Home window



Info window



Options window



Word bank window

Iteration 3 review:

Implemented:

In this section most program features described in the design section were successfully implemented, including:

- Home keyboard menu - containing mostly all of the original features stated in the design section (keyboard, input box, webcam feed, info, options and settings window) the only two features that were changed was removing the scroll bar from the info window as it seemed unnecessary given the large area of the output box to fill, and removing the calibration window button as a calibration process is no longer required given the changes made to the gaze estimation algorithm.
- Info window – includes a brief rundown on how the program works, and three points describing how to control the selector and how to use the options and word bank windows. The only main difference from the design is the removal of the calibration instructions and scroll bar.
- Options window – contains 4 buttons, and 4 text elements, two naming the software feature that each button changes (transparency and select length) with the other two describing by how much each click increases/decrease the values. The only feature removed was the hold length adjustment buttons, as the hold feature was removed due the fact that their main purpose was to allow users to control scrollbars within the program, which was also removed.
- Word bank window – this window contains 20 useful common phrases to save the user time in typing out certain words, it is also identical to the word bank windows initial design, with the exception of a scroll bar – for the purpose of the word bank window, I now think the scrollbar feature should have been kept in as the window only fits 20 phrases, which may possibly be of help to the user, but is still very limited.

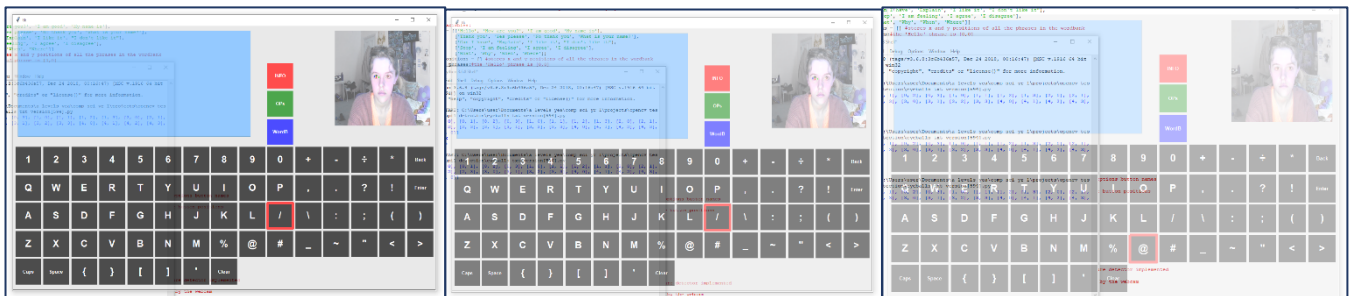
Changes to design:

Many features were changed from the design, including:

- Removing scroll bars/hold functions: scroll bars were removed from the program as the info window text and required number of phrases (20 phrases) could easily fit into the given window space, and the input box was so large it felt like an unnecessary feature. This meant the hold function was removed to as its primary purpose was for handling the scroll bars within the software. In addition, as we are no longer using tkinter widgets within the windows, this task would have been quite time consuming and difficult to implement well into each window within the time given.
- Removing calibration button/window/instructions: due to the changes made to the gaze estimation algorithm, a calibration process was no longer needed meaning all features relating to calibrating the software, stated in the initial design have been removed, as implementing them would have served no purpose to the software what so ever.
- Adding a clear button: a clear button was added to the home keyboard so instead of the user having to continuously blink of the back space or restart the program to remove a long string of text, they can just clear the output box with one button.
- Adding a selector: a selector was added due to the gaze estimation algorithm changes, as the cursor was no longer been controlled the user required a new method of navigating the software. The selector is a red boarder to each button that can move left, right, up or down depending on the user's gaze direction.

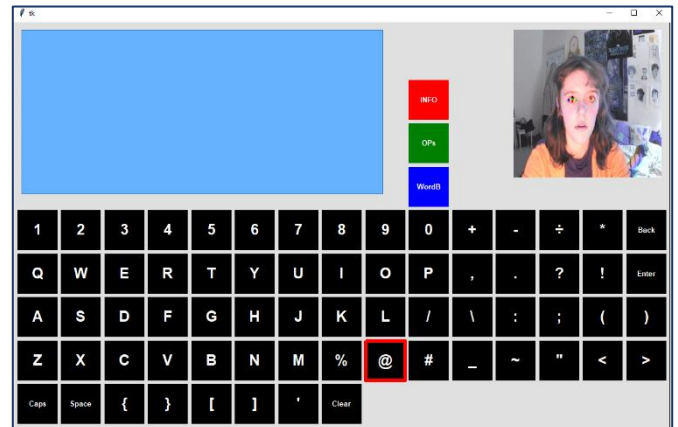
Success criteria:

- User settings window: this condition was met as the user can both vary the select length, and the transparency of the window, as shown below:



- User friendly info window: this criterion was also met, and evidenced in the third iteration, where the info window was shown to contain very clear, simple, comprehensive instructions for how the program should be used correctly.
- Common phrase word bank: criterion met as evidence from third iteration shows the word bank window containing 20 common phrases that can be output into the input box with a single click.
- Easy to find buttons: the writing within the buttons is bold and white so more contrasting with the background colours, which also match the home designs original button colours, making them easy to find and distinguish from the other keys on the home window.

- Virtual keyboard – large, bold keys: the writing within the buttons is bold and white so more contrasting with the black of the keyboard, so should be very easy to read for most users. The windows keyboard section takes up 0.509 of the window space meaning it meets the criteria of taking up at least a half of the main window.



- Uncluttered layout: the input box, the buttons and the webcam feed are all more than 1cm away from each other meaning the top half of the window is very uncluttered, however the distance between the keyboard and the input box and the buttons is less than a centimetre. Therefore, this criterion has not been met.
- Large input box: the input box takes up 0.2066 of the entire home window meaning this criterion has been met.
- Web cam feed displayed: as evidenced in the image above, the webcam feed is displayed in the corner of the home window, meaning this criterion has been met.
- Quick calibration (max 3 step of calibration)/Instructions with each step of the calibration: neither of these conditions have been included due to the removal of the calibration process, meaning these two criteria have not been met.
- Virtual keyboard/gaze detection can be used on 2 other applications: unfortunately, I have not had the time to apply the gaze estimation software to any other application, due to the software not controlling the cursor anymore as well as not having the time to complete this criterion, meaning it will not be met.

Evaluation:

Final testing:

Now we have fully implemented the program we can move onto fully testing it against the testing checklist and test plan. As the gaze estimation algorithm has changed significantly, certain elements of the test plan, such as the testing the accuracy of the gaze estimation on a virtual memory, will not be included in testing. Instead, as from testing the functionality of the gaze direction detection program, I know the movement of the selector can work correctly, however in this section I would like to test its effectiveness under different conditions. I will try moving to the word bank window button, opening it and outputting the 'explain' phrase under these conditions to test whether the program still works effectively:

- In a dimly lit, well-lit and an over lit room

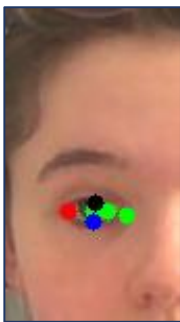
- Wearing glasses, no glasses
- 0.7m, 0.5m, 0.2m away from camera

In addition to testing the code under specific conditions, I will do another run through of every function in the program to ensure all/most items on the testing checklist have been completed or completed in an alternative format.

Beta testing: finally, once the final testing is complete and all potential errors have been addressed and noted, I am going to have one of my stakeholders test the program to see if the program works effectively for other people's faces and eye movements.

Destructive testing – robustness of program:

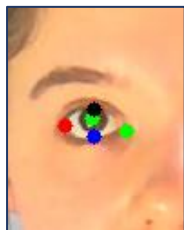
Lighting testing:



In the dimly lit (one light on to the left of the user) room moving around was still possible however unwanted movements were significantly more common as due to the lower light pupils were being detected in the corner of the eye, causing the selector to move right very often. Adjusting the position of the head so it was closer to the screen helped due to the source of light. The word was opened very easily and phrase was printed out in the output box, meaning the clicks function still worked effectively in low lighting. The time taken to open the word bank window and print the phrase was approximately 1.5 mins.



In the over lit room (two bright lights shining directly at the user) due to the extreme brightness, the program couldn't effectively detect features of the face. This made moving the selector around impossible, meaning for the software to work, the user should not have lights shining directly at their face.



Whereas in the well-lit room (one room light and a low intensity light to the left of user) the program worked very well, moving in all the correct directions. An issue that did occur a few times is in a well-lit room, clicking is slightly more difficult than in a low-lit room, as the program twice did not detect the users' blinks. However, the time taken to open the word bank window and print the phrase was approximately 45 seconds, which is a significant improvement from the dimly lit and over lit rooms.

Glasses testing:

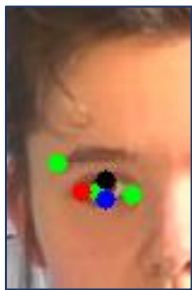
As a large portion of the population wear glasses, I thought it was important to test whether the program even works for this particular demographic. The testing will take place in a well-lit room and the time taken to output 'explain' from the word bank window whilst wearing glasses will be assessed against the time taken without glasses.



unfortunately, the software doesn't work at all when the user is wearing glasses as the pupil was either detected on the rim of the glasses or not at all, due to the glasses obscuring certain facial features meaning the facial landmark detector couldn't work correctly. Therefore, this program can only be used by individuals who don't need glasses to see.

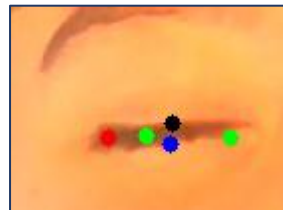
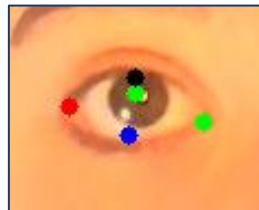
Distance testing:

As the user's head is bound to move whilst using the software, it is necessary to test how the users head position/distance affects the accuracy of the program. The testing will take place in a well-lit room and the time taken to output 'explain' from the word bank window will be recorded for each distance.



At a distance of 0.7m from the screen, much like in the dimly lit room, the clicks were very effective but movement was not accurate at all, due to the distance the pupil was being detected too far to the left making it very difficult to move left, as well as causing the selector to move right incorrectly. It took about 2 minutes to print the word 'explain' out into the output box, due to the difficulties when moving the selector.

At a distance of 0.2m from the screen, the location of the eye was detected very well however due to the lighting from the screen the pupil was always detected, even when the user had their eyes closed meaning the word bank window couldn't be opened as a click couldn't be invoked:



At a distance of 0.5m from the screen, the program worked most effectively, printing 'explain' after only 38 seconds with no issues.

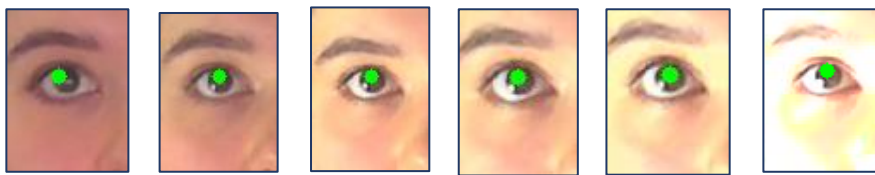
From testing the program in these different environmental conditions, we can conclude that:

- The program doesn't work at all with lights shining directly at the front of the user's face.
- The program doesn't work accurately in a low-lit room (one low intensity light to the side or behind the user).
- The program doesn't work if the users face is too close to the screen (<0.2m)
- The program doesn't work accurately if the users face is too far from the screen (>0.7m)
- The program doesn't work as accurately when the user is wearing glasses.

Testing checklist:

Cycle 1: Pupil detection:	
Pupil detection program can detect pupils	✓
Pupil detection program can detect clicks and hold commands from blinks	✓

Evidence of the pupil detection algorithm being able to detect pupils is evidenced through images from the first development iteration:

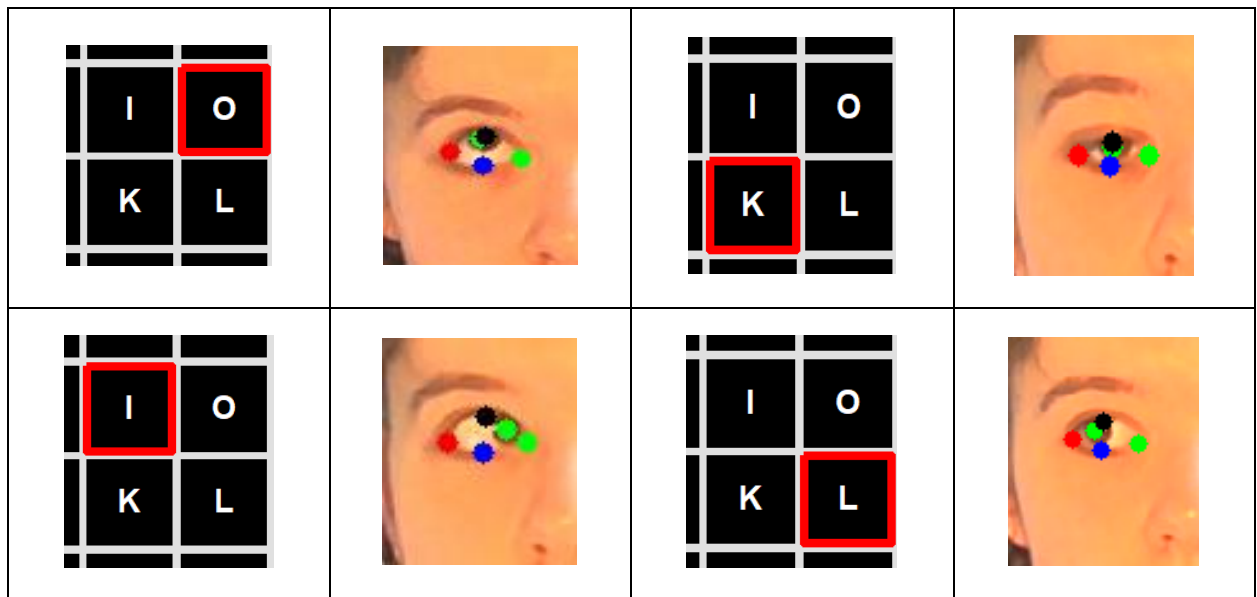


As this shows the pupil detection algorithm accurately detection the pupil in each image, shown by a green dot, meaning the first condition to test can be checked off. The second condition to test was both moved to the next development stage to be created with the gaze estimation algorithm and was also altered later in development. It was altered in the third iteration when the hold command was removed from the program, as scroll bars were no longer required as everything could be easily fitted into the window area. In addition to this algorithm for detecting blinks has changed as it no longer uses the width and height of the user's eye, but rather the fact that a pupil isn't detected in frame, so the initial way of evidencing this feature has to be changed. Instead, I will provide a screenshot of the user blinking showing that the pupil dot isn't present in the frame, meaning the it hasn't been detected so a blink has been detected - which can be further evidenced though a letter being entered into the keyboard menus output box:

Therefore, the second condition to test can be checked off as well.

Cycle 2: Gaze estimation:	
Gaze estimation program enables the cursor to be controlled by pupil movements	✗
Gaze estimation program accesses calibration data from external file and uses them correctly	✗

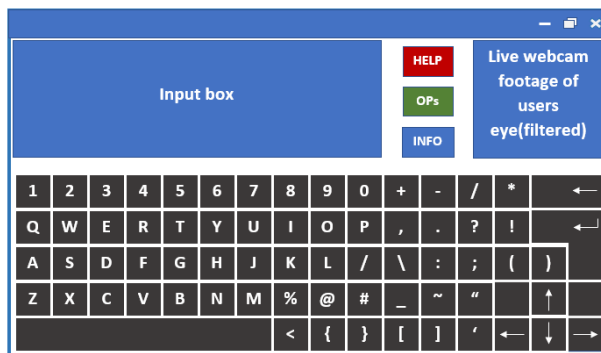
In the program, the gaze of the user isn't controlling the movement of the cursor so technically I haven't met this condition, as it is rather controlling the movement of a selector throughout the virtual keyboard/additional windows. However, this function can be evidenced through images of user's eye and the selector position within the same frame showing how it traverses through the keyboard under the control of the user's pupil:



As the calibration feature was removed, I have also failed to meet the requirement of having the calibration access the original focal length data from an external file, as it is no longer needed within the program.

Cycle 3: windows/layout:	
Correct formatting for each window	✓
Calibration window opens when program opened	✗
Webcam feed and instructions present in calibration window	✗
Input box can be written into on calibration window and inputs are stored in external file	✗
Menu screen opened when both input boxes in calibration window are filled	✗
Blinking for set click time enables a button to be clicked	✓
Blinking for set hold time enables a button to be held down and released	✗
'HELP' button opens info window	✓
'OPs' button opens settings window	✓
Arrow buttons on settings window alter value in text box	✓

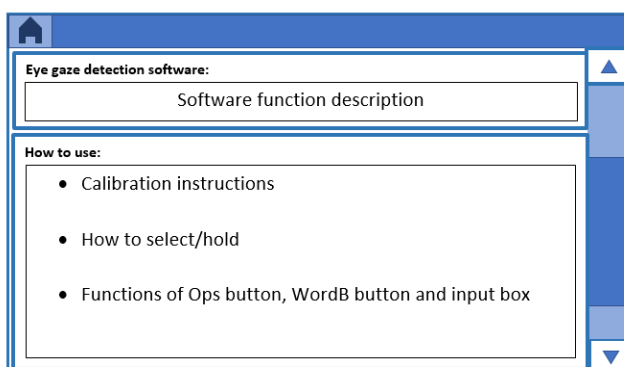
Error message displayed in front of settings window when values in text box go beyond or below boundary values	✗
'WORDB' button opens word bank window	✓
Hold function can be used on vertical scroll bar to move down a window	✗
When phrase button pressed in word bank window, menu window opened and phrase selected is displayed in input box	✓
'CAL' button opens calibration window	✗
Home button opens main menu	✓
Character buttons can be used to input text to input box	✓
Webcam feed displayed on menu screen	✓
'Cap lock' button capitalizes all text being entered into input box	✓



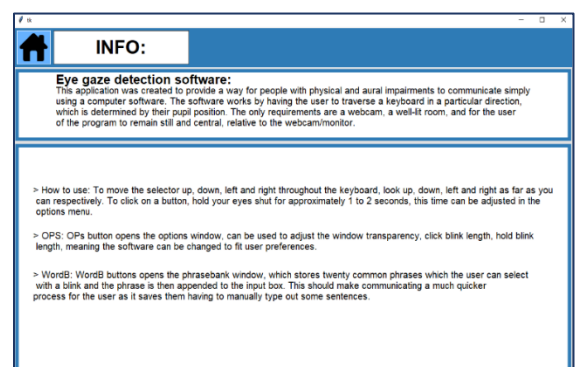
Home window design



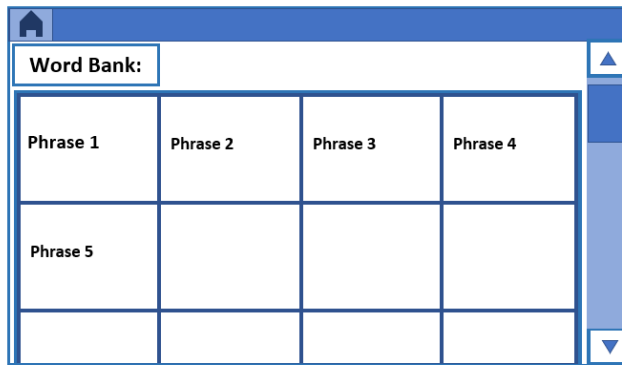
Home window implementation



Info window design



Info window implementation



Word Bank:

Phrase 1	Phrase 2	Phrase 3	Phrase 4
Phrase 5			

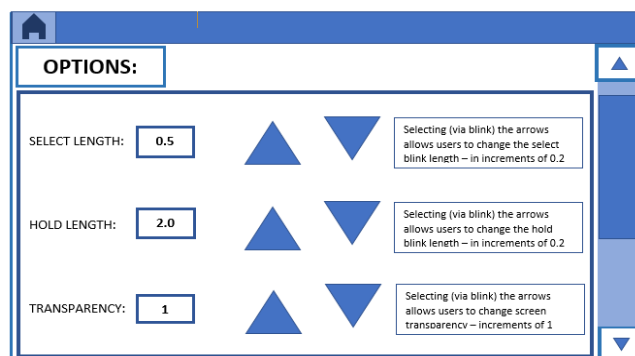
Word bank window design



WORDBANK:

Hello	How are you?	I am good	My name is
Thank you	Yes please	No thank you	What is your name?
Can I have	Explain	I like it	I don't like it
Stop	I am feeling	I agree	I disagree
What	Why	When	Where

Word bank window implementation



OPTIONS:

SELECT LENGTH: 0.5

HOLD LENGTH: 2.0

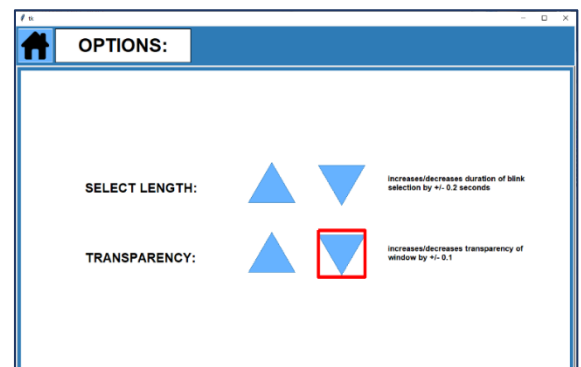
TRANSPARENCY: 1

Selecting (via blink) the arrows allows users to change the select blink length – in increments of 0.2

Selecting (via blink) the arrows allows users to change the hold blink length – in increments of 0.2

Selecting (via blink) the arrows allows users to change screen transparency – increments of 1

Options window design



OPTIONS:

SELECT LENGTH:

TRANSPARENCY:

Increases/decreases duration of blink selection by +/- 0.2 seconds

Increases/decreases transparency of window by +/- 0.1

Options window implementation

As evidenced by these pictures all of the windows contain the correct features and layout designs when compared to the design window. The only difference is the home windows colors and that the implementation doesn't include the scroll bar, which would've served no purpose to the code as everything could fit within each window easily. There are also two fewer buttons in the options window implementation than in the design, which was due to the removal of the hold feature. Therefore, the windows have the correct formatting so the first condition can be ticked.

The following 4 conditions all involve features of the calibration window which was never implemented as it no longer serves a purpose in the new gaze estimation algorithm. Therefore these 4 boxes can be crossed out as there is no way to evidence ever completing any of them.

The default select/blink time is 1 seconds however to prove the blinks only work after being detected for a set amount of time, first I will provide an image of the camera set up to prove that I am only using a webcam to detect these blinks. Then I will alter the code so whenever the pupil isn't detected in frame, it prints the duration that the pupil was not detected as well as whether a blink was invoked, as if the duration is less and a blink was still invoked, it means this condition has not been met:

```
0.08188414573669434
2.310473918914795
click
3.115948438644409
click
0.5728542804718018
0.08301281929016113
```

As shown in the image the only times a click was invoked was when the duration of blink was above 1 second, whereas when it was below, it was just passed as a natural blink, meaning this

function works correctly so this condition can be ticked.

However, as the hold command was removed from the program, this condition cannot be met as there is no way to evidence completing this criteria.

The next two conditions of having the info and options buttons open the info and options window have been met, as evidenced in the third iteration. Therefore these two criteria have been met so can be ticked of the checklist.

Although a text box was not included in the options window implementation, the arrows still altered the intended software features, select length and transparency, meaning this condition has, to an extent been met. To evidence this, I have altered the code to when the select length is increased and the window transparency is decreased the new select length and transparency are output at each click, showing that the user is successfully controlling the features in the options window:

```
click
0.9
click
0.8
click
0.7000000000000001
click
0.6000000000000001
click
0.5000000000000001
click
0.40000000000000013
click
0.30000000000000016
click
click
```

*Transparency decreasing with each
click of the bottom right arrow*

```
1.2
click
click
1.4
click
1.5999999999999999
click
1.7999999999999998
click
1.9999999999999998
click
2.1999999999999997
```

*Select length increasing with each click of
the top left arrow*

No error message was displayed in front of settings window when values in text box go beyond or below boundary values, but rather the program wouldn't decrease/increase the value of it exceeded the boundary value, meaning this condition, was not actually met.

Like the info and options window, the word bank window was evidenced in the third iteration however to further evidence that the wordbank button can be used to open the wordbank window, this video shows the user, opening the word bank window with the word bank button and outputting a phrase from the word bank window into the output box: [wordb test.mp4](#) from the videos folder. This provides evidence that the wordbank window and phrase buttons both work correctly, so these criterions can be checked.

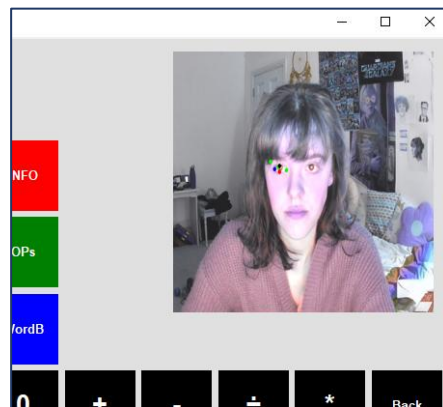
All calibration and scroll bar related criterions have been crossed as these two features were removed in the implementation stage.

The home button can also be used to open the home window from each of the additional windows (info, options and wordbank). This feature has been tested in the following window by opening the keyboard menu using the home button from the word bank window: [home test.mp4](#) from the

videos folder. Therefore as the video provides raw evidence that the home button works, we can check this criterion off.

Both the cap lock and character entry keyboard features were shown in the third iteration of the implementation section, this video, showing the user typing the capitalized letters into the input box, gives clear evidence that these features work and are only controlled by eye movements/blinks - [cap test.mp4](#) from the videos folder.

The final criterion to test is if whether the webcam feed is displayed in the corner of the home screen and, as seen in the third iteration and test videos, this feature has been successfully implemented:



The only issue with this feature is that in a well-lit room the image appears very blue and discoloured, which may be quite distracting for some users.

Stake holder testing:

Now the final testing is complete, I wanted to see how the software worked when one of my stakeholders used it. Therefore I sent the final prototype to Joss Sullivan and requested feedback points on how it worked for her, and these were the main points she made in her response:

- Very simple, fun and easy to understand.
- Instructions unclear – too concise.
- Is at first very difficult to get the hang of, does make the eyes hurt after a while.
- Looking right doesn't work all the time and had to keep moving the window up so I wasn't looking down at the keyboard window – as that was detected and the square kept on moving down.
- Blinks were sometimes not detected or were wrongly detected when my face was out of the camera frame.
- In general the eye movements were very accurate – especially up and right.
- Potential issue is that the software is very straining on the eye, so maybe reducing the amount the user has to look up and left, as they are already very accurate.
- The above point may also mean that the software isn't suited to individuals with lazy eyes or eyes that cannot move as freely due to certain conditions like cataracts.

Response takeaways:

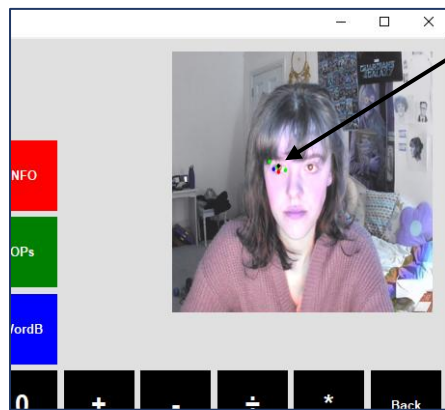
- Is at first very difficult to get the hang of, does make the eyes hurt after a while/ Potential issue is that the software is very straining on the eye, so maybe reducing the amount the user has to look up and left, as they are already very accurate:
 - o The monitor light combined with the user having to exaggerate their eye movements in order to navigate the screen mean after some time, it may put a great deal of strain on their eye - meaning this software may not be well suited to people prone to headaches.
- Instructions unclear – too concise:
 - o More detail within the info window on how to use the software is required, as the window was considered not user friendly and too concise.
- Looking right doesn't work all the time and had to keep moving the window up so I wasn't looking down at the keyboard window – as that was detected and the square kept on moving down:
 - o When the window is opened it appears near to the bottom of the screen meaning it is occasionally detected as a down command, causing the selector to incorrectly move down the keyboard. This issue may be very frustrating for the user; therefore, I have altered the root window so it appears in the top centre of the screen to remove this issue:


```
root.geometry('+200+0')
```
- Blinks were sometimes not detected or were wrongly detected when my face was out of the camera frame:
 - o Blinks are sometimes not detected due to the program predicting the user's eye landmark coordinates, even when its closed, meaning a pupil is wrongly detected. This can be avoided by ensuring the lighting surrounding the user is not too bright and that the user is not too close to the monitor screen.
 - o Blinks are also wrongly detected when the users face goes off screen as a blink condition is if no pupil is detected in frame. This can lead to issues such as letters being typed or windows being closed/opened without the user meaning to. To avoid this issue in the future, I could have implemented the blink detection algorithm so that the time a pupil is not detected in frame is only detected as a blink if a face is detected in frame.

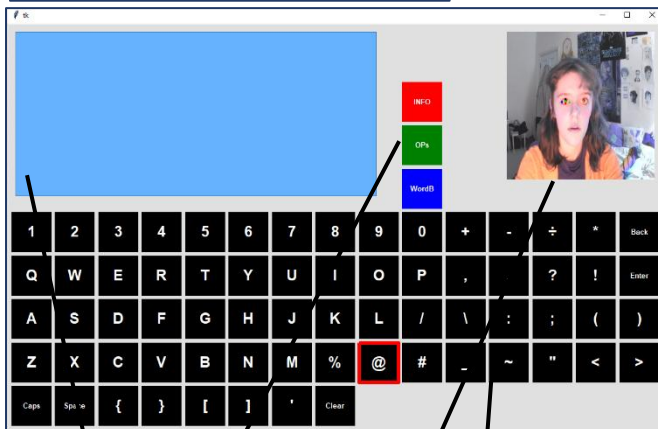
Success criteria met:

Criteria	
Webcam used to accurately control cursor movements	✓
Webcam used to control clicks	✓
User settings window	✓
User friendly info window	✓
Common phrase word bank	✓
Easy to find buttons	✓
Virtual keyboard – large, bold keys	✓
Uncluttered layout	✗
Large input box	✓
Web cam feed displayed	✓
Quick calibration (max 3 step of calibration)	✗
Instructions with each step of the calibration	✗
Virtual keyboard/gaze detection can be used on 2 other applications	✗

Success criteria evidence:

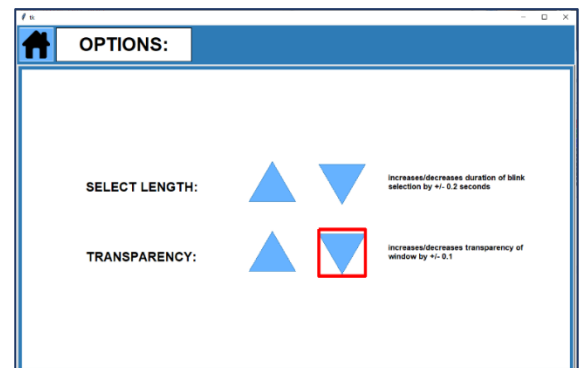


Webcam used to accurately control cursor movements + Webcam used to control clicks



- Easy to find (colorful) buttons
- Virtual keyboard – large/ bold keys
- Large input box
- Web cam feed displayed

Webpage was not uncluttered due to the small distance between the keyboard and other window features (input box, window buttons).



Options window implementation



Word bank window implementation

As the info window was considered too concise by Jos, this criterion is no longer met as it clearly isn't up to the stake holder's standards/requirements, which is what the success criteria was based off.

- Quick calibration (max 3 step of calibration)
- Instructions with each step of the calibration

These two criteria were not successfully implemented as the calibration process was no longer needed in the new gaze estimation algorithm.

- Virtual keyboard/gaze detection can be used on 2 other applications

This criterion could not be implemented to a good standard in the time given, therefore I decided to leave it out to avoid running into further issues at the end of the implementation stage.

Usability features:

The software windows were consistent, using similar layouts, colour and fonts. Each of the info, help and word bank windows contained a header and a home button in a banner at the top of the window, meaning it was very easy for the user to get back to the main keyboard as the window headers formatting was consistent across the three windows. The window buttons being very bold in colour made it very obvious to the user that they were separate to the keyboard keys and could be used separately, and due to them being different colours (red, green, and blue) it should be very easy and quick for a user to distinguish between the three buttons.

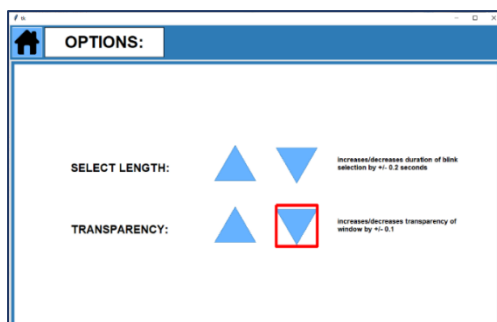
The keyboard takes up half of the menu meaning the keys a large enough space for the text inside to be bold and readable for most users. The key text is white, so the contrast of the text against the dark background of the keys should make reading the keys easier for most users.

The word bank window is another usability feature as it allows the user to output an entire word/phrase at once, without manually spelling it out, meaning it could speed up communication for the user.



WORDBANK:			
Hello	How are you?	I am good	My name is
Thank you	Yes please	No thank you	What is your name?
Can I have	Explain	I like it	I don't like it
Stop	I am feeling	I agree	I disagree
What	Why	When	Where

How to use the software is very simple and not difficult to understand, meaning most users should be able to grasp the basic principle of it fairly quickly. In addition, the info window provides a brief summary of the programs features and instructions for use, potentially helping the user understand how to use the software correctly.



The options window also gave the user the opportunity to change software settings like the window transparency and click length. This usability feature is what enables the software to be more tailored to a user's individual requirements/preferences.

Maintenance:

Limitation:	Solution:
Only 20 phrases, limits the number of word bank phrases the user can access to speed up communication:	With more development time, a scroll bar feature, along with the hold command, could have been implemented into the program. Meaning more than 20 phrases could have been stored in a single window and scrolled through using this feature.
Strain on eye as the pupil direction is how the user controls the selector movement:	Changing this feature would involve changing the fundamental purpose of the software which is to interact with technology using pupil movements in order to communicate. Therefore, this limitation can't be solved, however the user could be warned when the program first opens that it is not well suited to individuals prone to headaches/with eye conditions.
Can't move cursor freely around screen, meaning software cannot be easily applied to other applications:	In future developments to the program, a potential solution to this limitation is to try to re-implement the initial gaze estimation algorithm that controlled cursor movements. However, as this seems infeasible given the limited set up equipment (a webcam), using the current gaze estimation algorithm to move the mouse to the nearest button/widget in an application is another alternative solution to this limitation.
Look right sometimes undetected:	The ratios for looking left and right may vary significantly from person to person as everyone has different eye widths, therefore an initial calibration to personalise the left and right movement eye ratios to the user is a potential solution to this limitation.
Info window instructions too concise – unclear for stake holder:	The goal when implementing the information window was to make it as clear and concise as

	<p>possible, however more instructions must be given on features such as:</p> <ul style="list-style-type: none"> - How far to look up, down, left, right - Optimal lighting - Optimal head position - Functions of different keyboard buttons (caps, clear, back) <p>As now I realise, without this information, for someone who is not too familiar with technology, without the adequate guidance it could be very difficult to navigate a very technical software such as this one.</p>
Very particular lighting/facial position conditions must be met for the software to work correctly:	<p>This limitation is due to the inexpensive and low-quality webcam set up, meaning the webcam feed quality is very bad in poor lighting. Upgrading the set up would also change one of the fundamental points of the software which is to create an inexpensive, accessible software for those who need an alternative method of communication. Therefore, this limitation cannot be solved.</p>
Software very slow to use – slow method of communication may be very frustrating for some users:	<p>Like with the third iteration, a potential solution to this limitation is to try to re-implement the initial gaze estimation algorithm that controlled cursor movements. As the user is directly moving the cursor it would significantly speed up moving the selector/cursor across each window to get to a button. However, if this method is not feasible, implementing diagonal movements would make traversing the keyboard potentially faster.</p>

Almost every coded feature and development section has been commented/labelled in the program code, meaning in future maintenance of the software, it should be easier for other developers to understand and continue working with the code. In addition to this, the program is modular as it heavily utilises functions, meaning new features to minimise the current limitations of the software can be easily implemented as an additional function or by using the existing functions.

In further developments of the code, the initial gaze detection algorithm could be re-attempted using a slightly more advanced camera set up – as long as the total cost for the application to run is kept lower than the average price of commercial gaze estimation applications (<£200).