MyEye : Eye Capture Window Manager
Final Report

Group 31

11th January 2015

Contents

Chapter 1

Executive Summary

MyEye is a window manager navigation tool. It uses the head position to change window focus. This removes the need for using standard input peripherals, such as a mouse and keyboard, to complete this task. The goal of MyEye is to make navigating windows on a Linux OS easier and faster, allowing users to spend more time working on useful things, as opposed to managing windows. This application makes use of several existing libraries and techniques, and improves in some areas.

MyEye can also provide a benefit to users who have trouble using the standard window focus method, and could easily be incorporated into wider Ease-of-Access systems in computer use. For individuals who struggle to use a computer, MyEye provides a way to streamline the process without any additional learning - user's look at the screen they want to use anyway, so MyEye lines up with their current usage.

MyEye runs in the background, but has a UI that can be used to utilize some settings. The window switching can be turned on and off and you can switch between cameras if your computer has multiple web-cams, and there is a display showing what is being focused by the window manager. In order for the user to be able to see what MyEye is doing to track the user's gaze (and also to help know which camera to use), the UI shows the web-cam stream with some of the head tracking information displayed. This gives the user a simple visualization of some of the processes being used, and can also be used to help uncover problems if the arise.

Chapter 2

Introduction

2.1 Background

2.1.1 Input Methods

Window Managers are software designed for creating, displaying, positioning organizing windows on screens. A window manager will usually draw window decorations such as title bars and window borders which a user can interact with using a mouse to position and resize the window. The basic concept of windows decorated in this way has changed little since it's conception. It's only recently that less decorated windows have been introduced into the mainstream especially for devices where touch input is one of the main methods of interaction. Many window managers and Linux distributions use the X Window System; as such we decided to target systems that use the X Window System for our project.

X Window System Linux window managers generally use a combination of mouse and keyboard input, with keyboards being more often used. This is made available to them in the Linux operating system by the X Window System which is a windowing system for bitmap displays. The X Window System is used to draw and move windows on the display device and for interacting with the mouse and keyboard.

X is separated into a client-server model. A server provides the basic functionality of creating, moving and resizing windows as well as dealing with mouse and keyboard interactions. A client will send messages to the server formatted according to the X protocol. The transportation method of these messages is undefined by the X protocol, this means that a client and server can be one different machines and the messages sent other a network, or both be on the same machine and an Inter Process Communication method used.

If no window manager is used, windows will be displayed without any decorations meaning users can't move or resize windows using the keyboard or mouse. As such the window manager is required to send messages to the X Server in order to create child windows for the title bar and borders of actual windows. This means the window manager is a client of the X server, just like any other program that has a window.

Most clients don't directly create and send the messages to the X11 server, instead a library called Xlib is used, this removes the need for each application to know the details of the X protocol.

Xlib Xlib provides a large number of functions for interacting with the X server. Those we make use of both get and set attributes of the windows both visible and invisible. Rather than use Xlib directly many applications will use an Xlib backed framework, such as Qt, that provides higher level abstractions which are often based on the notion of widgets.

2.1.2 Eye Gaze and Face Tracking

Viola-Jones technique Viola-Jones technique is a widely used method for real-time face detection. It uses machine learning to select optimal classifiers and allocate weights to each classifier. Classifiers are acquired from labelled data, where the majority of data does not contain faces and about 0.002% data

contains faces. It uses "Integral Image", an image that computes a value at each pixel, which improves the calculation speed of detector. Also, it uses AdaBoost learning algorithm to select the most crucial classifiers from many potential classifiers. Moreover, it invents the "cascade" algorithm when combing classifiers, so that the background region of the image can be abandoned quickly and concentrate more on potential face area. As a result, Viola-Jones technique is most robust to detect faces and high time efficiency, which coincides with the requirement of eye tracking: identify face precisely and use few computations.

Figure 2.1: Haar feature used by the Viola-Jones technique

Histogram of Oriented Gradients Histogram of Oriented Gradients (HOG) is another method for detecting objects including faces. It splits the image up into cells, and for each cell calculates a histogram of gradient directions or edge orientations across the pixels in that cell. The descriptor that is used for matching is just the collection of these histograms. Groups of cells are called blocks and these blocks are used to normalize the cells within them.

OpenCV OpenCV (Open Source Computer Vision) is a popular computer vision library focus on real-time image processing. The library has many application areas, including facial recognition system, gesture recognition, human-computer interaction, etc. OpenCV provides an API for capturing video frames from cameras and video files, and methods to interact with the image data, such as applying filters, accessing pixel data and specifying regions of interest. Additionally it provides an implementation of the Viola-Jones technique and trained cascade classifiers for detecting the face, eyes and nose, among other objects. By specifying the face as a region of interest (effectively cropping the image to the face), the search area for the eyes and nose can be drastically reduced, meaning less time is needed to annotate the image with the location of the face and facial features. Because these annotations must be performed in every frame of the video, a relatively small reduction in time for each frame means a noticeable increase in frame rate of annotated frames.

dlib dlib is a C++ framework that has a large variety of different uses, although we are most interested in the Computer Vision parts. In particular, we are interested in the facial landmark detection which uses a 'Histogram of Oriented Gradients (HOG) feature combined with a linear classifier, an image pyramid, and sliding window detection scheme to detect a variety of facial landmarks'[3]. The dlib implementation of landmark detection has several advantages over the Viola-Jones technique:

1. It is far better at detecting features when the user's head is rotated, as Viola-Jones struggles with this.
2. It can find a large number of points (68 in total) whereas Viola-Jones finds regions. This allows us to track points with greater precision and means we can decide precisely which points we look at.
3. dlib's implementation of feature selection is faster than OpenCV's, which is very important when an application has to run in real time.

2.2 Motivation

2.2.1 Regular Users

Many users find it difficult to, or simply do not, learn the commands necessary for efficient navigation around desktop environments. This means their work throughput is negatively affected and stress levels

are raised unnecessarily. Hotkeys which are mapped to these tasks are also often in positions that mean moving hand positions or stretching from one part of the keyboard to another. This can increase the risk of RSIs and for non-power users means looking away from the screen. In cases where hotcakes are not available, point-and-click navigation is even slower.

2.2.2 Power Users

While power users may well have learnt how to use their window managers well, errors are still made, especially in multi-monitor set-up. Our supervisor gave us examples from his own experience, having different work present in several different monitors. Often he would find he had made irreversible changes by accidentally typing input into one program while looking at another.

Switching focus between screens and windows using a keyboard when only the keyboard required doesn't take too much time when it is done correctly. However, when mouse input is required most window managers do not move the mouse automatically, instead requiring the user to do this manually. This means moving the pointer through, potentially, several screens which does take a significant amount of time. Whilst this is only fractions of a second, when being done rapidly and repeatedly throughout the day, this time adds up to a significant amount. Also, mouse movement is limited to the space of the mouse pad and moving the screen pointer so far may exceed this space, making movement even more awkward.

2.2.3 Accessibility

Users with certain types of accessibility issues using a PC sometimes have problems with certain types of interface. This means desktop navigation can be made more difficult or in some cases impossible.

2.2.4 Solution

Tracking a users gaze and head position means implicitly following their focus around the screen or multiple screens. It also does not require any knowledge of key combinations, be them modifiable or not. This means it will be both easy to pick up for new users and give experienced users a faster way of navigating around their desktop.

Most households use web-cameras for some purpose on their personal computers, with almost all laptops now having integrated web-cameras. The hardware necessary for this type of window navigation is therefore also readily available and no extra purchases are needed. Part of the challenge of this project is to ensure that our software can function using cheap, low-resolution cameras that may or may not give us all the detail of a better camera.

2.3 Objectives

Since we did not know how time consuming or difficult the project would be, our primary objective was very simple:

- We should be able to detect which screen (out of two) a user is looking at and switch focus between them so that it feels responsive.

However, we had a variety of stretch goals should we achieve the main objective with time to spare. These include:

- Improving the accuracy of the gaze tracking to detect quadrants on the screens.
- If that is possible, then to increase the precision as much as we possibly could.
- To be able to use more screens (for example, 3 with left, right and middle)
- To be able to use an arbitrary number of screens in any configuration (although this would require a good deal of user input to configure)
- To add some additional features, such as controls that could be used by blinking, that would give the user even more control over their window environment.

With a simple main objective and a variety of stretch goals, we were not entirely sure at the beginning of the project how our final application would turn out.

2.4 Contributions

As MyEye will automatically change focus according to head movement of user, this function ensures user won't be distracted during work by busy changing the focus when using multi-screen configuration. Therefore, usage of MyEye would significantly increasing the working efficiency of those power users and regular users using multi-screen.

MyEye gives users freedom to continue to use their window manager. Instead of build window manager based on open source window manager, we intentionally use X server to control the focus of screen, ignoring normal window manager functionalities. As a result, user can enjoy the convenience of eye tracking while having the useful features of a variety of window manager in the market.

MyEye consumes low computing power, although a normal graphics-based software requires high computing power. We optimised the head detection by calculating a rough area that head is most possible to appear. As a result, we can find head position in that area find, instead of finding head through the whole graphics.

MyEye will improve the productivity of users who use multi-screen computer intensively and require faster speed to switch screen, such as trader in investment banking. In addition, the UI of MyEye is intuitive to use for everyone. User can stop and restart the program easily. The equalise function makes the software runs effectively in high light environment.

Chapter 3

Project Management

3.1 Group Structure

3.1.1 Work Allocation

We realised from the start of our project that it was clear there were members that would be more suited to certain areas of work. The members of the group taking the computer vision course were more suited to the vision aspects of our the software development. This meant it was very easy to determine an initial group structuring and allocation of work accordingly. We have split the group into 2 even sized sub teams (3 members each), one working on the eye tracking and the other working on switching windows/window management and user interface in the initial stage.

This division lead us to the conclusion we should create two fairly separate programs, sharing very little, if any, internal details. The window team will have to provide an interface to change the focus of windows, and will design the communication between the sections. The eye tracking library will detect the user gaze, and convert its conclusions into messages. A third component sits between the others to tie them together, translating messages from the eye tracking system into commands to the window system. Splitting the implementation in this way allows the potential to use the components separately, for instance using the eye tracking system with other operating systems or other display servers, or building a system that detects another physical input which can change the focus of windows and be managed in the same way.

3.1.2 Group Meetings

Weekly meetings always took place in the labs. We did not always have the resources available to test our software during the meetings, as we required multiple monitors. There were always opportunities in the quieter periods to do this. We also did not have a set meeting time every week, as the availability of members often changed with deadlines and other responsibilities. We did however, always manage to meet at least once a week to:

1. Discuss new features to be added to the Kanban board
2. Review the previous week's work
3. Organise times for meetings with our supervisor
4. Schedule sub-group meetings if necessary

Minutes were kept from every group meeting so we had a record of what was discussed and which members were present.

3.1.3 Sub-group Meetings

Sub-group meetings did not always happen in person, as they happened much more frequently than other meetings. Being less formal, less organisation was required and were scheduled when needed. They existed mainly to collaborate if any member was struggling to implement a particular feature or had questions on other members' implementation. They were also useful when deciding how to move forward generally within a particular subgroup (such as discussing the algorithms we were considering for improving the accuracy or efficiency of the vision component). Like the main group meetings, we would add items to the Kanban board as necessary.

3.1.4 Supervisor Meetings

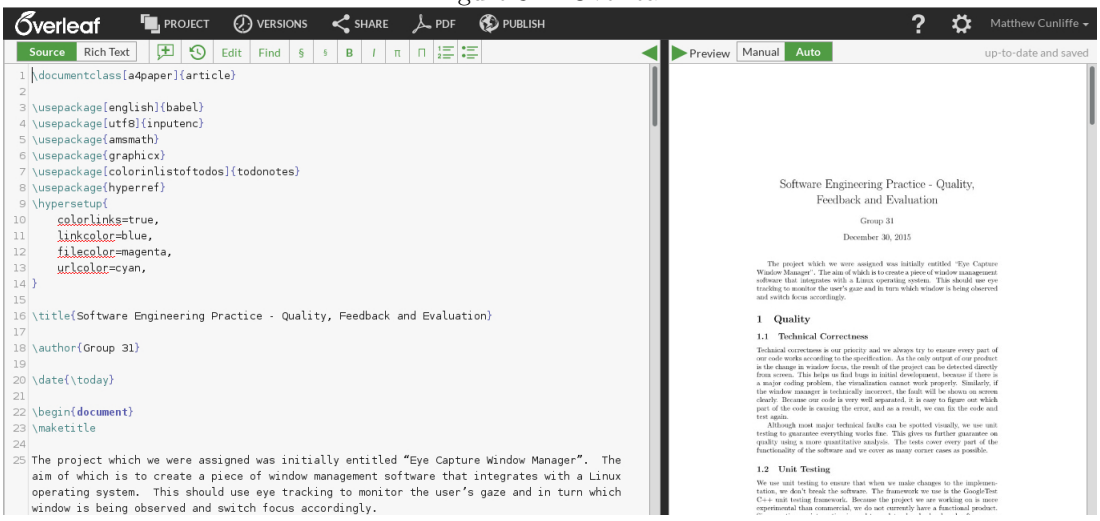
Meetings with our supervisor happened less often: every couple of weeks. Therefore, these meetings were longer and there was more to be discussed. They generally consisted of:

1. Showing our supervisor the features we had implemented.
2. Discussing how to best improve the software.
3. Receiving advice on our current progress.

3.1.5 Overleaf

Initially, for writing the reports we chose to use Google Docs, which allows collaboration on a rich text document and is stored on Google Drive. After finishing the document on Google Docs, we would transcribe it into a LaTeX document. To remove the need of transcribing, we used Overleaf which allows real-time updates and collaboration on a LaTeX document. The website also renders the compiled `.pdf` which allows all members to see in real-time the compiled document.

Figure 3.1: Overleaf



3.2 Development

3.2.1 Methodology

We chose to subscribe to the Kanban methodology from the start of our project to its finish. Kanban is mainly a change management system with an emphasis on just-in-time delivery. Releases are made after features have been added, meaning that the software slowly evolves over time. There are no set work

periods with certain features expected after each, for example, sprint. Development is constant and a Kanban board is used to track features through the work flow; practices can be optimised to decrease the time a feature is on the Kanban board.

The potential scope of our project was very large, however we had a limited amount of resources to complete it with. Kanban allows us to limit the amount of work that is being completed at any one time. This means we don't put too many demands on ourselves in one go.

At the start of our project, both ourselves and our supervisor were unsure how difficult or time consuming each feature we implemented would be. As Kanban encourages small changes over time, our group was agile to changing requirements while we decided which way our project would go, and which direction would allow us to produce the best product.

Through the development process, we followed the guidance of Kanban and frequently analyzed our workflow, trying to improve the efficiency. We aimed to get the general structure of the project first, so we analysed and developed the face recognition part (finding facial features, but doing nothing with them yet), putting other goals in the TODO part. After finishing this, we moved it to QA and started to analyze if the user was looking the left screen or right screen using facial features. Meanwhile, we incorporated the eye tracking part with the window manager part. As a result, we had a minimally functional eye capture window manager. At that time, the accuracy of eye tracking was low, the application was slow and unresponsive, and it had many restrictions on how users should sit, so the next target was improving the accuracy and efficiency of eye tracker and making the system robust. Since our group was clearly subdivided, it was easy to split into our subgroups and improve each part, as the overall architecture was built.

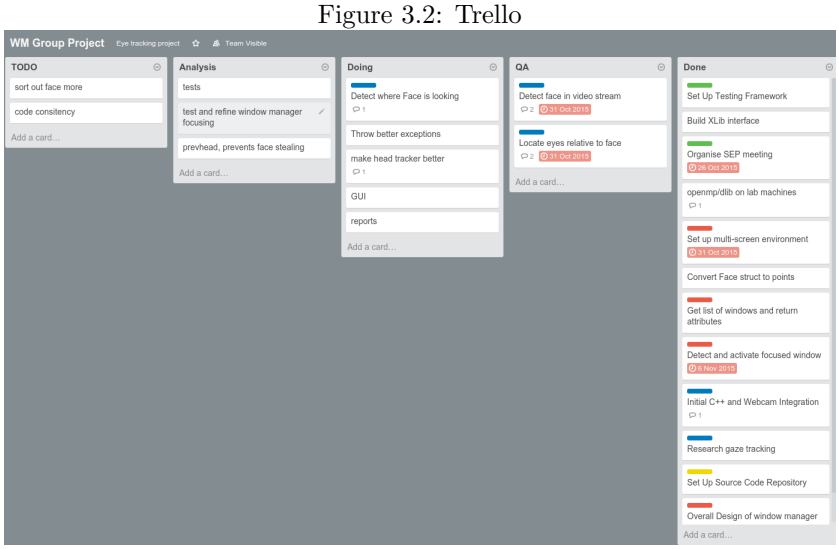
By this time, the deadline of the project approached, we did not finish all features we plan to achieve, for example, we hope to make it compatible with multiple screens instead of two. However, we have a proper product to release and we can continue to develop it agilely in the future.

3.2.2 Trello

As a substitute to a physical Kanban board, we used a website - Trello. Trello gave us the functionality of a physical Kanban board and could be accessed from any location. It also includes the ability to colour cards, making it easier to distinguish which tasks were associated with each sub-group.

In addition to this, deadlines for features were simple to add to the cards. Meaning, at a glance, we could view whether we were keeping to the deadlines we had set for ourselves; how many features were late and by how much this was the case.

Figure 3.2: Trello

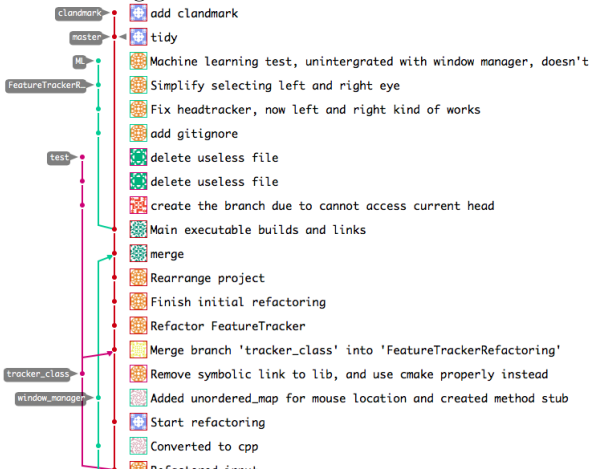


3.2.3 GitLab

We used git on the Department of Computing GitLab servers `https://gitlab.doc.ic.ac.uk/` for version control while developing our software. We made extensive use of branches in git. Branches were used whenever we had a new feature to implement, which allowed us to still have a working master branch, and also a branch in development. Further sub-branches were also useful in cases were errors need to be corrected to supposedly finished features, or if features were particularly complex. Each sub-team also had their own branch. At the point where a feature had been successfully developed, the new branch is merged with the master branch.

Another feature of git we used was sub-modules. Using sub-modules meant we could keep another git repository in a subdirectory of our repository, in our case GoogleTest. By doing this, we didn't need to add the GoogleTest code to our global repository, it could be downloaded into any local ones when needed. This means space is not taken up storing this code, it was not mandatory to clone the code when not needed, and updates to the GoogleTest framework can be downloaded independently.

Figure 3.3: GitLab commit tree



3.2.4 CMake

CMake is a tool used to manage the build process for a program (in a UNIX environment this usually means generating makefiles). Since we are using a wide variety of external libraries to provide extra functionality to our project, we decided to use CMake to create our makefiles. This was extremely useful, since we did not have to manually change them whenever we altered our dependencies in the code, and we saved a great deal of time and effort by automating the process.

Chapter 4

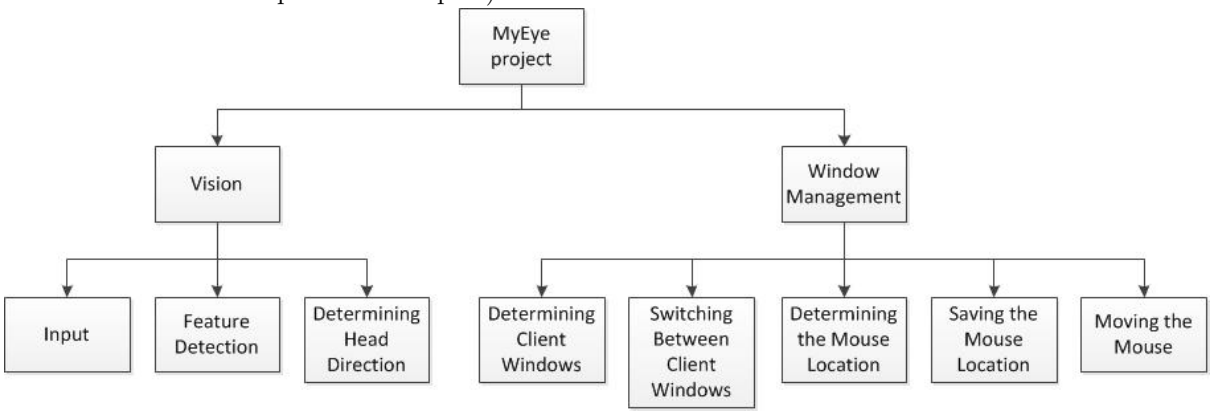
Design

4.1 Back-End Design

Our design process started upon our initial meeting with our supervisor. We knew that our software had two parts: vision and window management. We were concerned to keep each part as separate as possible, so that our two sub-teams could work effectively without needing constant communication with each other. We decided we could achieve this separation by each part being a self contained library. This was important as each sub-team could work fairly independently, and only a short time was needed to work together to join the two components. Within the Vision part of the project, we were similarly concerned with keeping the code as modular as possible. The idea here was that we could add new parts to the process to refine the accuracy and precision of the gaze tracking without having to disrupt other parts of the code. An additional benefit of this modular design was that it made using version control easier. Our issue was we had not had much experience with designing software of this kind.

The back-end design of the whole program is divided as following graph, there are functionalities under each part to fulfil:

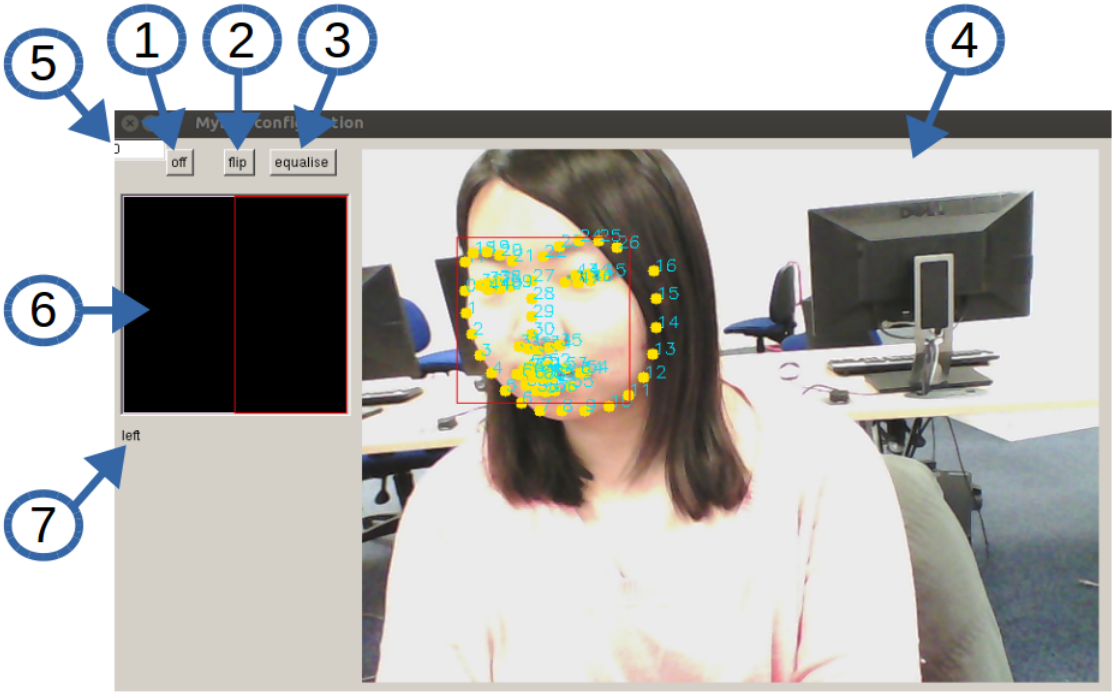
Figure 4.1: An overall view of the back-end design (Detailed implementation of each functionality will be introduced in the implementation part)



4.2 Front-End Design

The front-end of MyEye is the user interface. Due to the nature of our software it required very little UI, because when the user starts working, he/she can just let MyEye running in the background and let it automatically switching the focus accordingly. Therefore, our principle to design UI would be concise and easy to understand. Here is our UI design with functionalities labelled from 1 to 7:

Figure 4.2: A screen-shot of the UI in use



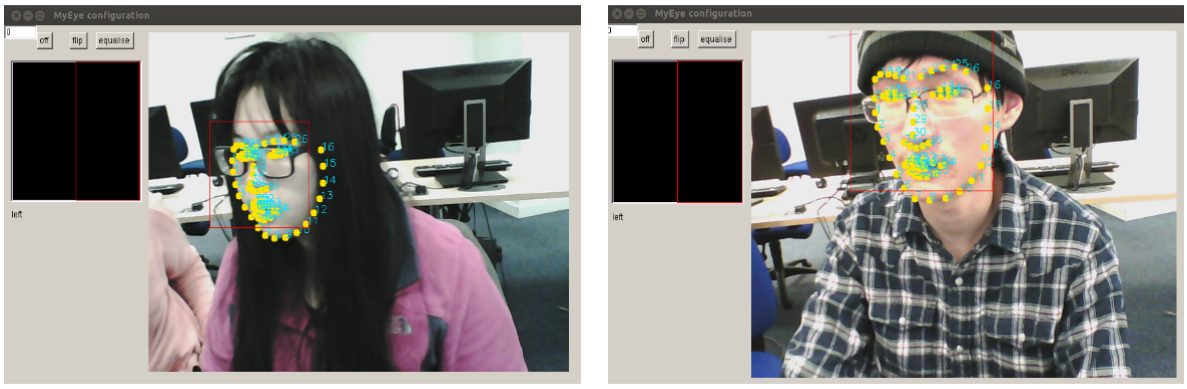
1. ON/OFF button: Click to turn on/off the detection process. If the detection is turned on, the system will change focus according to gaze motion, vice versa.
2. Flip button: Click to change the left and right screen. Should only be used when the setting of two screens is inverse of the detection.
3. Equalise button: Click to equalise the graph. While detecting under some high light environment, the normal setting would not be accurate and may not be able to detect the features, in this case, user may click equalise to let the detection work as usual.
4. Real-time graph from camera: This rectangle will show the graph transferred from the current camera. If MyEye detects a human face, it would label all 68 features(including mouths, noses... etc) with yellow points and green numbers it detected according to the face.
5. Camera selection: A listbox which shows the identifier of the current camera selected by MyEye.
6. Screen setting visualisation: The focused screen is sunk. According to the example graph, the user is staring at left screen, so the left rectangle is the focused screen which is the sunk.
7. Current focused screen: Use text to tell user which is the current focused screen.

4.3 User Experience

Since the target users of our program are power users, we found some classmates of computing department who needs our product to improve their working efficiency and they are willing to provide feedbacks. Our users are full of different characteristics, we have males and females, people wearing glasses and not, European and Asian faces. During the project, we were required to make a video pitching the software, and during the making of this we discovered some issues to do with the User Experience. Mainly, we found that there are some situations when the user does not want MyEye to be running, such as when copying from one screen to the other. This led us to add an on/off toggle so that the user is not inconvenienced in this way.

Here are pictures of two example users who are using our program:

Figure 4.3: Some examples of users testing the software



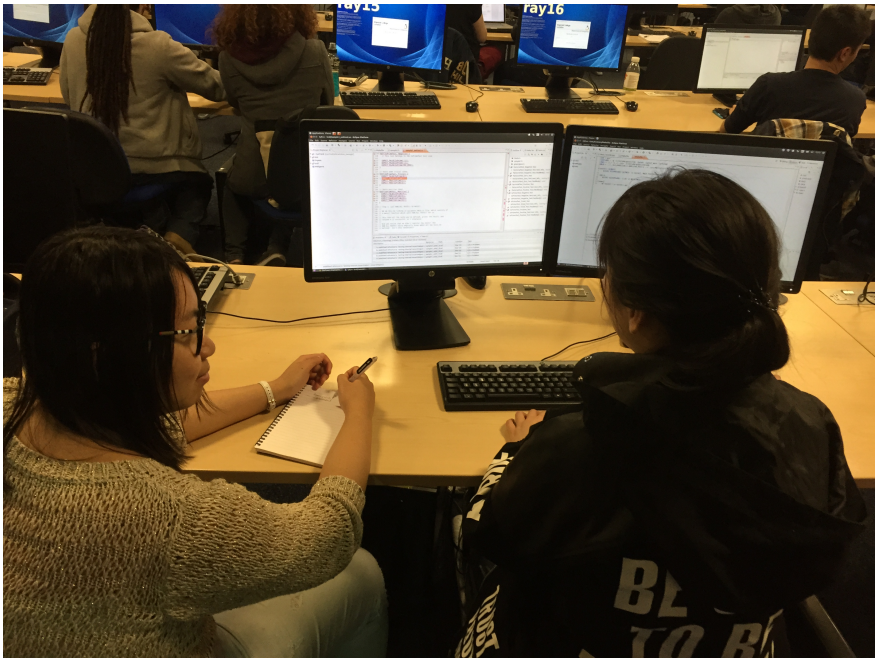
In order to get feedbacks from users, we interviewed them after letting them have a try and perform a series of tasks to test our program.

Figure 4.4: An example of user feedback

MyEye Group: How do you feel after using MyEye for an hour? Talking about accuracy and sensitivity of the program.

User: Well, it can definitely define which screen I am looking at correctly at most situations, even when I was staring at somewhere close to the middle. Also, I'm quite satisfy with the sensitivity of program, because the focus was never changed automatically when I keep my vision in the same screen. However, it can't detect my face if it can only see my side face, but since I will always face to the screen it doesn't matter in real use.

MyEye Group: Thank you very much!



4.4 Risk

4.4.1 Security Risk

Many web-cams, including ones built into laptops, have an indicator light that turns on when the camera is being used. As MyEye uses the camera constantly, this light will also be on constantly. This means if another program starts to use the camera the user won't notice as the light will have been on already due to MyEye.

Risk cannot be prevented entirely, because we cannot predict what other software will do. However, we can guarantee the users we will not record any video nor send the video through network. Meanwhile, we suggest users to use standard techniques to protect themselves, such as using firewalls and install software from reliable source only, so that the security risk can be minimised.

4.4.2 Performance Risk

MyEye runs in the background as long as it is opened. It is quite computationally extensive although we optimise the algorithm by storing the previous position of the face.

The software put risks to normal computers' running speed. However, for computers with limited computing power, the software will take longer to detect the user's face and facial features. This will reduce the frame rate, and create lag between the user looking in different directions and the software switching focus giving a suboptimal user experience. It would also degrade general system performance.

This risk does not exist for most computers because the newly developed computer all have decent computing power. If a user finds their computer runs slow without using MyEye, then they should not install and use it. Users can update their hardware if they need to.

4.5 Technical Challenges

- Users may have their preferred window manager because window managers all have different features and ways to manipulate.

  To address this we decided to interface with the X server directly through Xlib, rather than target a specific window manager and use it's API. This is because not all window manager's have an API, and each one is different. This means the user can use almost any X11 based window manager that suits their personal preference while using MyEye.

- Initially, our algorithm did not always detect the correct direction of users' gazing direction.

  We had to experiment with various methods of feature detection and gaze tracking, and in the process switched from using openCV's implementation of the Viola-Jones technique to dlib's landmark detection algorithm as our primary method of feature tracking. This is because the Viola-Jones technique is primarily useful for detecting whether there is a face in an image - this becomes more obvious when we consider that only 0.002% of images used during machine learning component actually contain a face. What this means is that the technique fails to find features accurately in real time with a moving face. dlib is far more accurate for this, and so we decided to switch.

  Similarly, when detecting gaze direction, we spent some time deciding which landmarks to use. Initially, we only used the bridge of the nose for the left/right direction, but it quickly became clear that we needed more. We decided to use the eyes as well, and to use a 'voting' scheme to decide on the direction.

- We needed the software to run at a sufficient speed to feel responsive to the user.

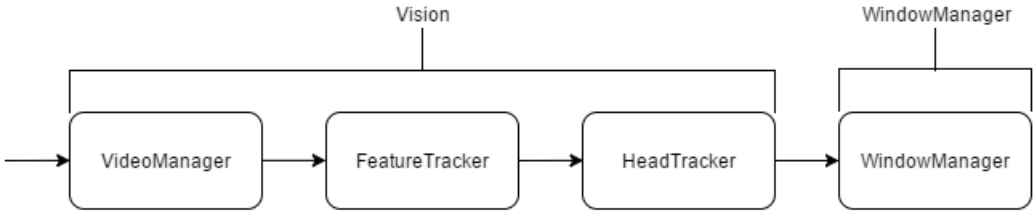
  If the software ran too slowly, the user would notice lag when switching between windows. We made sure the software was as fast as possible by minimising the search area during the feature detection component (i.e. remembering the location of the head between frames).

Chapter 5

Implementation

5.1 Overall Architecture

Figure 5.1: A flowchart showing the classes used for the back-end implementation



The program is split into components matching the way we split the workload. We abstract the interaction with the window manager to a library, called WindowManager, and a separate library for the vision part. These libraries are then statically linked to the main application, which is the executable that is run to use MyEye.

Splitting the program in this way provides the advantage that dependencies are contained to the libraries. The WindowManager library requires X11, which is not used on all systems. To support other systems all that needs to be done is a new library created that implements the correct interface, using whichever dependencies it requires. For example we have the wm-nox11 library which has an empty implementation of the required methods. This was used in development so that the entire program could be compiled on systems like OSX that don't use X11, but now serves as an example of how the window manager library can be swapped out for another.

5.2 Main Application

The Main Application combines the components provided by the Vision and WindowManager libraries to produce the full application. It is also responsible for managing settings and creating the graphical user interface.

Video input is managed by the `VideoManager` class, which uses the `changeVideo()` method to switch between a Camera input `CameraInput`and an input from a file `FileInput` which is very useful for testing. The use of this class is to abstract away the input method from the main code and therefore the subsequent parts of the program, since it is not relevant to the rest of the code.

The GUI displays a window containing the live video from the camera, annotated with the face rectangle and facial landmarks. It also displays controls to change various settings.

The main loop of the program succinctly shows the overall flow of the program.

```
while (!window.is_closed()) {
       try {
           frame = videoManager.getFrame();
           Face face = featureTracker.getFeatures(frame);
           face.drawOnFrame(frame);
           Direction dir = headTracker.getDirection(face);
           window.image.set_image(frame.dlibImage());
           window.set_focus(dir);
           if (settings->getTrackingState()){
             windowManager.set_focus_screen(dir);
           }
       } catch (const char * e) {
           window.image.set_image(frame.dlibImage());
       } catch (NoInput e) {
           cout << "No Camera detected, exiting.." << endl;
           exit(0);
       }
   };
```

5.3 Window Management

Our implementation of window management using the coordinated of the users focus works with the current window manager. That is to say it does not replace it, but calls to X Server in addition to the running window manager. Calls to change focus of the screen and modify or get pointer coordinates.

There are some issues which are encountered using this method. We had to tailor our solution to certain WM due to certain quirks in different software.

5.3.1 X Server

The **X Server** represents all items on all screens as 'windows'. This can make it quite difficult to distinguish between the type of information that is being displayed in each one and to see how they are related. i.e. Which window is contained within another and is a part of the same process. This is necessary to determine which windows we should switch focus between. For the most case, sub-windows (those which are contained within those we want to switch between) are not able to be focused on, so the program would throw errors if this was attempted. However, if it was the case that we could do this, it would still be unwanted behaviour, as we are not attempting to control a users interaction within applications. So, in both cases this needs to be avoided.

Extra work needs to be done if we were allowing windows to overlap or exist but not be visible. Therefore, we have assumed the restrictions of:

1. There will be no minimised windows
2. Windows must not overlap

5.3.2 Xlib

Properties Each window has 'properties' attached to it, an example of this being the window 'name'. Properties are named, typed data The purpose of these properties is to associate extra information that can be used by the window manager or for other purposes. We can create these ourselves, however the one we use is already existing in most window managers.

Atoms Each property has a atom associated to it which is a unique identifier. Atoms are used for efficiency reasons, and the atom associated with a property name can be obtained using the `XInternAtom()` function.

Root Window The root window is the window which contains all other windows. In the default case, most of the time, this window will stretch across as many monitors as exist. This is what we have assumed when building the software has the ability to switch between multiple monitors.

5.3.3 Determining Client Windows

The root window contains a property `_NET_CLIENT_LIST` which is a list of all the client windows. (*This is dependent on the window manager that is being used, but in our implementation we assume the window manager provides it.*) Here client windows refer to visible 'top-level' windows or in other words, those in which we are interested and need to switch focus between. The function necessary to return this list is `XGetWindowProperty()`. Once these client windows have been discovered, we can find their size and location.

In some window managers, we discovered there is an arbitrary number of client windows which we are not interested in at the start of this list. We found this to be either 1 or 2 generally. To circumvent this problem, this number of windows needed to be hard coded into the code depending on the window manager. This introduced a dependence on the window manager which is something we were trying to develop our software to avoid.

In addition to the previous point, some window managers do not make use of `_NET_CLIENT_LIST` which narrows down the number of window managers we could tailor our software to.

5.3.4 Switching Between Client Windows

Once the list of client windows has been successfully discovered, switching focus between them is a fairly trivial process. This is done using the `XSetInputFocus()` function. The input window parameter to this function is determined in `pointInPolygon()`. By cycling through the currently open windows and determining whether the coordinates of the users current focus, which is determined separately, is in each window we can find this window. This is only a possible solution because we have restricted our desktop environment so it cannot have overlapping windows.

5.3.5 Determining the Mouse Location

The location of the mouse can only be determined through the function `XQueryPointer`. The mouse location only needs to be determined when we have detected that their focus has moved to a window different to that where the mouse is placed. This is for the purpose of saving the mouse pointer so that it can be reloaded when the same window is activated. `XQueryPointer` takes a window as one of its arguments and returns the pointer location if the pointer is within this window. This means in a set-up in which there are no hidden windows, we simply need to query all the current windows to determine where the mouse currently is. We cannot simply assume that the pointer coordinate will be contained within the previously active window. There were 2 options we had in the coding at this point (if the mouse was not in the previously active window). Simply leave the pointer where it was or snap the pointer to an arbitrary point in the activated window, most likely the centre. We chose to leave the pointer unaffected. The reason for this being, it is more likely the user was busy elsewhere and would still be working in another window, than him wanting to snap to a new window which has no previously saved mouse location.

5.3.6 Saving the Mouse Location

The Mouse Position was saved in the Main function. There were four fields, leftX, leftY, rightX, rightY, recording the coordinates of the mouse in left and right screen respectively. As the focus was only changed when the the user's focus changed from left to right or right to left explicitly, that was if user was looking in the middle, the focus would not change. The mouse position was saved everytime before the the focus was shifted. We used XQueryPointer to get the coordinate, and update leftX, leftY or rightX, rightY, depending which direction the user was looking. If the user was looking right, left coordinate were updated to the real time coordinate returned by `XQueryPointer` . By using this function, we can

always save the last position of pointer in a given screen and switch the focus effectively. Meanwhile, when user focused on one screen, he could move mouse freely, MyEye would not track pointer position in this case, saving the computer efficiency.

5.3.7 Moving the Mouse

Moving the mouse was achieved by `XWarpPointer`, which was a function in Xlib. `XWarpPointer` takes many arguments such as `display, src_w, dest_w, src_x, src_y, src_width, src_height, dest_x, dest_y`. `dest_w` specified the destination window or None. When set it None, because according to Xlib Manual, when `dest_w` was None, `XWarpPointer` moves the pointer by the offsets `dest_x, dest_y` relative to the current position of the pointer. As a result, we just need to calculate the offset between current pointer position and target screen's saved pointer position. Moving the mouse to the screen you're looking at is a feature that can be enabled if the user wants it.

5.4 Vision

This is the section of the software used to read in images from a webcam, detect features and find the direction a user is looking.

5.4.1 Input

We then use functions contained in OpenCV to read in the video file frame by frame. Two classes are used for input, `FileInput` and `CameraInput`, both inherit from the abstract class `Input` which has a pure virtual method called `getFrame` that returns a struct called `Frame`. This struct wraps the OpenCV matrix that stores the frame. This achieves two things, the Main App doesn't need to know anything about OpenCV matrices and so if we change from using OpenCV the main application doesn't need to be changed, and methods have been added to the struct to return the frame in the type needed by dlib which is different to OpenCV. By overloading the implicit conversion operators for `Frame`, it can be passed to OpenCV and dlib methods, making the code more readable as no explicit conversions to non-simple types are needed.

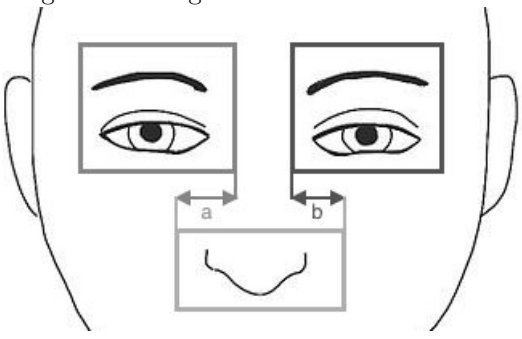
5.4.2 Feature Detection

The frame is passed into the feature detection class `FeatureTracker`. Initially we implemented the FeatureTracker using OpenCV's object detection, the `detectMultiScale` which is based on the Viola-Jones technique and uses Haar cascades specific to the object to be found. We detected the face and then cropped the search area for the eyes to the face and for the nose set the search area half the width of the face centred around the middle. We found we achieved less false positives and missed less features without a noticeable reduction in performance, this way rather than searching for the eyes and nose in the entire frame.

The `getFeatures` method returned a struct called `Face`, which initially contained fields for the rectangles that locate the face itself, the left eye, the right eye and the nose.

However we found that the classifiers in OpenCV weren't very reliable, often there'd be frames where the eyes and nose were not found, and even the face wouldn't be found. This meant the method we initially used to determine the direction of the user's head didn't work well.

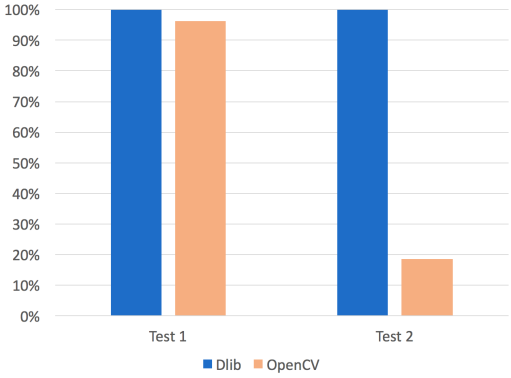
Figure 5.2: Diagram showing our initial method for feature detection



a and b will change when the user rotates their head due to perspective. If a is greater than b then the user is facing that direction and vice versa.

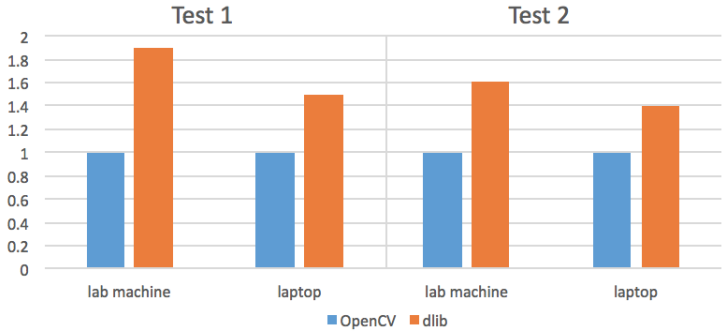
We compared dlib and OpenCV's detection of the facial features by recording video of a face and using it as input for the program. The first test video was of a face that barely moved, whereas the second was of a face looking left and right. The tests were run 3 times for each implementation and test video and the results averaged. OpenCV found all of the left eye, right eye and nose in 174 frames out of a total 181 in the first test video, however it only found the features in 15 frames out of 81 in the second test video. Dlib performs much better, in the first video it finds the facial landmarks in all 180 frames, and in the second it finds them correctly in all 81 frames.

Figure 5.3: Graph showing the performance of the two feature detection methods



When timing how long each implementation took to process the two test videos, dlib was 1.9x faster than OpenCV for the first test video on a lab machine and 1.5x faster on a laptop . For the second test video dlib was 1.6x faster on the lab machine and 1.4x faster on the laptop.

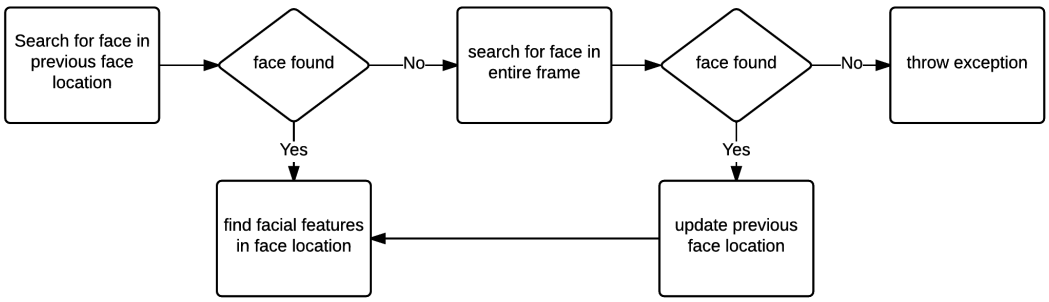
Figure 5.4: Graph showing the performance in terms of time of the two feature detection methods



Due to these results we switched to using dlib completely for the implementation of the`FeatureTracker` class.

First, we use the dlib `frontal_face_detector` class, which uses the Histogram of Oriented Gradients (HOG) technique, to detect the location of the whole face within this rectangle we use the `shape_predictor` to find 68 points, called facial landmarks, on the face. As the user's head will not move much from frame to frame we store the rectangle containing the face when it is initially found and search for the face in just that rectangle rather than the entire frame. If no face is found, then we search the entire image and if a face is found continue processing it as normal and update the stored location of the face. If no face is found after searching the whole frame, we throw a relevant exception as we have nothing to return. We do not update the stored location of the face every time we find a face as this leads to the facial rectangle getting smaller and smaller, and means the entire image has to be searched every few frames which is less optimal.

Figure 5.5: Flowchart of the feature detection algorithm

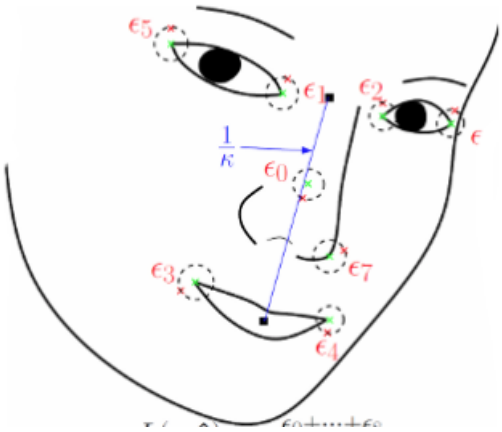


5.4.3 Determining Head Direction

Head Direction is calculated in the `HeadTracker` class. Here, we take the landmarks from `FeatureDetector` and use two methods of judging the horizontal direction of the face. The first is to use the eyes and edge of the face. We check the horizontal distance of the corner of the eye to the edge of the face for both sides. If one is more than double the other, then we judge that the face has turned in that direction. Otherwise, we return middle.

The second method is to use the bridge of the nose. We calculate the gradient of the line made by the nose ridge. This is done by calculating the difference in the x coordinate between the bottom and top of the nose, and dividing it by the difference in y coordinates. If the value we get is greater than a certain threshold (or less than minus that threshold), then we return that direction, or middle otherwise. This is the same as calculating the angle of the nose bridge and comparing it to a threshold angle. We also have to account for head tilting by rotating the face to vertical, whilst still maintaining the offset generated by the face being turned. To do this, we calculate the angle of the line formed by joining the eyes, and rotate the nose by this angle. We use the top of the nose as the origin of rotation so that only the bottom point of the nose has to be rotated .

Figure 5.6: Diagram showing the points of interest in head tracking

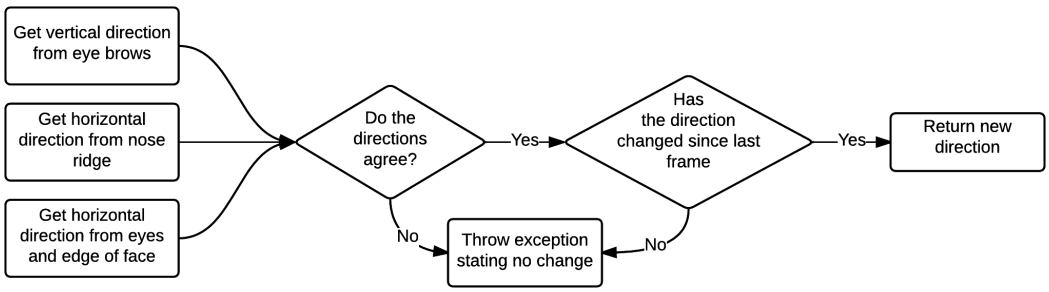


In order to determine final direction in the x axis, we combine the two. If they both return the same result, then we update the direction to the new one. However, if they do not agree we leave the direction as it was in the previous frame.

As part of a stretch goal, we decided to also calculate the vertical direction. To do this we get the apparent distance between the eyebrow and upper bound of the eye. As the user's head changes direction vertically this distance will decrease when they look down, and increase when they look up due to perspective.

Finally, the vertical and horizontal directions are combined and if the overall direction has changed since the previous one, we return the new direction and if the direction has not changed we throw an exception that states this.

Figure 5.7: Flowchart of the head tracking algorithm



Chapter 6

Evaluation

6.1 Testing Evaluation

6.1.1 Unit Testing

We used unit testing to ensure that when we made changes to the implementation, we didn't break the software. The framework we used was the GoogleTest C++ unit testing framework. Because the project we were working on was more experimental than commercial, we did not have a functional product for the first half of project. Since continuous integration was used to update already developed software, we did not use continuous integration in the beginning stage.

We used unit testing to test our project before it was added to master branch. The unit tests focused on two areas: eye tracker and window manager. The eye tracker tests ensured eye direction was captured correctly. Through the continuous development, the accuracy of eye tracker improved. The window manager tests ensured window focus was switched as expected.

Unit testing helped us during development of the software. Firstly, by using test-driven development, it was clear what the code we were going to write would do. After writing tests, we had a clearer expectation of the code and we could think about how to overcome corner cases before we code. Also, we could write the simplest code to make the test past, which breakdown a complex problem into several small ones. Secondly, unit testing gave us confidence on how good the software was. It showed the progress quantitatively. Thirdly, the error messages from the unit tests helps us debug and saves us time, as it clearly indicates which section of code needs to be fixed.

Here are some example usages of GoogleTest C++ unit testing framework in our program:

An example initial configuration of the test suits

```
\\ Initial configuration of window manager tests

TEST (window_manager, changes_pointer) {
    Display *d = XOpenDisplay(NULL);
    int set_x = 100;
    int set_y = 100;
    int ret_x;Graph showing the performance of the two feature detection methods
    int ret_y;

    wm* w_m = new wm();

    w_m->set_focus_to(d, set_x, set_y);
    w_m->get_pointer_location(d, &ret_x, &ret_y);
```

```
    ASSERT_EQ(set_x, ret_x) << "Pointer x location was not set correctly. Set: " <<
     set_x << " Actual: " << ret_x << "\n";
    ASSERT_EQ(set_y, ret_y) << "Pointer y location was not set correctly. Set: " <<
     set_y << " Actual: " << ret_y << "\n";

}
```

An example test

```
\\To test if our window manager can set the focus to left screen

TEST(window_manager, can_repeatedly_set_focus_screen_left){

    wm w_m;

    for (auto i = 0; i < 5000; i++) {
        w_m.set_focus_screen(wm::Left);
    }

}
```

6.1.2 Integrated Software

After both the eye tracker and window manager were functionally correct, we tested them as integrated software. Meanwhile, we improved its responses so that the system would not be over sensitive. This part of testing required a great deal of end-user input, as we needed to be able to test the product as if it were in daily use by a user.

Due to the nature of the software, the integrated software was intuitive to test directly using the GUI. When using the software, the mouse should always move to screen that users was looking at. If the mouse was not moved as expected, it indicated there was a bug in the program. Since the direction the of the head is printed to the terminal, we can know easily if the bug was caused the by eye tracker or window manager.

Apart from finding bugs, testing the integrated software gave us more inspiration on how we could improve the project. When running the software, we treated ourselves as users and thought about which areas in the software were difficult to use or had unexpected results. As a result, that part would be added to the to-do list on Kanban Board.

6.2 Gather Feedback

During the meeting with our supervisor Dr. Mark Wheelhouse, he mentioned the problem with our project is users like him will most likely use window managers which are inappropriate for our software i.e. non-tiling. We knew from research at the start of our project that we would only be aiming our software at tiling window managers. We then used this group of target users to refine and extend our software depending on their requirements and suggestions. Hopefully we would be able to extend our project to include non-tiling window managers.

Besides our supervisor, being in a computing department, we had access to lots of power users who could potentially test our releases and provide feedback. The problem we encountered was it is difficult to gain qualitative feedback quickly due to the nature of our software. We could test the project's basic functions are working fine(i.e. change focus of windows when the eyes of user change focus). However, it needs to be used in their normal day-to-day tasks to gain feedback of how well it worked for them. Issues to do with responsiveness and accuracy may seems acceptable in a short time frame to users, but dragged out over the course of a day it is likely that these problems can have a much greater effect. This sort of feedback will take a much longer time to gather than testing of basic functionality.

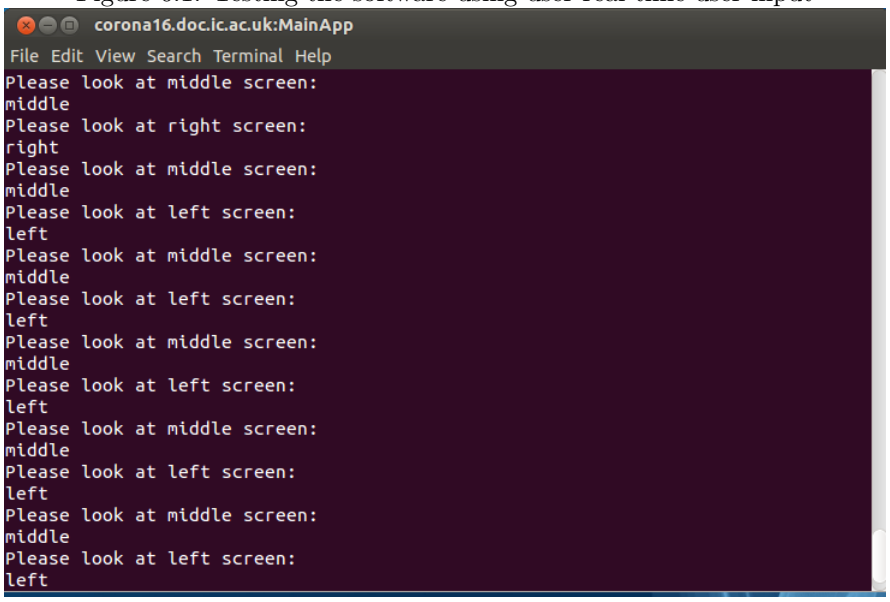
6.3 Deliverable

6.3.1 Software Performance

The algorithm we've written was able to provide a success rate of finding facial features which was expressed as a percentage, currently we use a threshold of 95% for success. By recording videos of moving faces, and annotating them with the direction they are facing, and location of their features (as it is not enough to know a feature was detected in the frame, we need to know if the feature was correctly located) we have a qualitative way of evaluating our software. This is done by comparing our annotations with the output of the algorithm used to identify the face and output direction vectors.

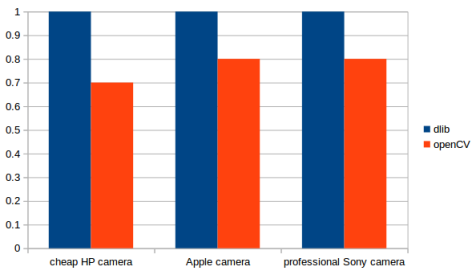
When the product was completed we had users test the software by them using it during everyday use of the computer. They were asked to record when the software incorrectly switches focus and when it doesn't switch focus yet it is desired. Details about the user was noted down, such as whether they wear glasses. A good threshold for success will be matching the user's expectations 95% of the time, and with a delay of at most of 1 second.

Figure 6.1: Testing the software using user real-time user input



We aimed to have this software work on a variety of hardware, especially cheaper off the shelf components. We acquired a collection of cameras, varying in expense and quality. The product was tested against all these cameras, and the results plotted, showing the expense/quality VS performance of our product.

Figure 6.2: Graph showing the influence of camera quality on performance



6.3.2 Project Evaluation

Overall, we think the project is successful, because we achieve the initial goal of the project, which even our supervisor Dr. Mark Wheelhouse did not know how hard it was. The tracker can detect directions effectively. The window manager can switch the focus of different screens, getting hardware configurations from X server, and moving focus according to eye directions. Also, the software is compatible with other window manager software. That is user can continue to user their preferred software while using MyEye to facilitate the screen navigation. Meanwhile, the efficiency of MyEye is very high, instead of computation intensive like most graphics-based software, because we optimised the eye tracker algorithm by filtering an area to search head, rather than search head from the whole photo. In addition, the UI is intuitive for users to use, helping users to configure the software better.

However, we still have many areas to improve. The current software only works with two screens, where our final goal is to make it compatible with up to nine screens. Because we used Kanban methodology, we broke larger tasks into different stages and achieved the goal stage-by-stage. We realized it was hard to achieve switching between nine screens from the beginning, but we wrote the code such that it can be extended without rewriting it all. Apart from that, the software has some restrictions on minimised windows, which could be improved in the future.

We think this software is very useful in improving working efficiency. We hope to develop the nine screen version in the near future, so that more people will be benefited.

Chapter 7

Conclusion and Future Extensions

7.1 Conclusion

The X Server and windows set-up is a simple system but that makes it quite complex and buggy to do certain things. A prime example being trying to change focus between windows. It is not a trivial task to determine the number of windows that we want the eye tracking software to distinguish between. The window tree is a good representation, but different software can include arbitrary windows and there is no concrete way to differentiate between what appears as a normal window to the user and what is technically a window but doesn't appear as one on screen.

The Vision part of the software is in theory fairly application independent, as all we are doing is determining in which direction the user is looking. However, the basic left/right as designated in the main objective has limited applications, and a much more precise method of determining gaze direction would be needed for this to be useful in other ways.

7.2 Future Work

Machine Learning:

Currently to derive direction we only consider some landmarks. One improvement we could look into would be to use machine learning to better recognise directions. While the Viola-Jones technique uses trained classifiers to recognise parts of the face, we could take all the landmarks provided by the HoG method and use machine learning to derive direction. This could increase both the accuracy and precision of the gaze tracking, and given enough training data could become more robust to different situations, such as users with glasses.

Develop software/games based on the whole Eye Capture Window Management System:

Develop multi-screen based games using the eye capture window management system, This kind of game is very attractive due to its high difficulty and the delight brought by the large screen.

Increase accuracy of Eye Capture function:

Currently the algorithm that implements gaze tracking does not take into account the pupils. The reason for this is that we were able to achieve our objectives by only tracking other points on the face, since when users change focus from window to window, or screen to screen, they move their head, not just their eyes. If we were to track the pupils relative to the position of the eye socket, or other features, we could potentially provide a greater precision based off of our existing knowledge of quadrants. However, one potential issue with this is being able to reliably detect the pupils at low resolution, which becomes a significant issue, since if we are working at the higher precision pupil tracking would afford us, we also need a greater deal of accuracy if the user is not to notice errors.

We use the dlib library in this project to detect features, but it detects far more features than we use. A potential improvement we could make would be to decrease the number of features we track to increase the efficiency of the program. Since the program is designed to run in the background with multiple other applications running, it is always valuable to increase the efficiency, since if the program runs too slowly, the window manager won't feel responsive.

Develop software/games based on the Eye Capture part:

Eye Capture is a functional part of the whole project used to capture gaze which is quite useful as an individual system in some situations. Nowadays, car manufacturers are developing anti-fatigue driving systems. They are our potential user of the Eye Capture part since their first aim is checking the gaze motions, and then gather this data to analyse if the driver is fatigued. Some manufacturers developing smart home functions (e.g. turn on the TV/ open the freezer when you look at it) would also be our potential users. Therefore, we can isolate the eye capture part as an individual software and make suitable interface for our potential users.

We have mentioned the possible use of this project in games, but another potential use for the Eye Capture part of the project is in advertising. It is useful for companies advertising over the internet, which is an increasingly common source of revenue for businesses, to know when and which adverts are being looked at. However, this would require a greater deal of precision than simply being able to determine quadrants.

As another instance of possible future work based on eye capture part, we could connect the music player with our eye (also face) capture system. Implementing the machine learning functions to analyse users' face to get the current emotion of user and play the appropriate music at real-time situations. For example, when the system detects the tired face of user, it plays some energetic music to wake the user up; when the system detects the focusing face of user, it decreases the volume of music to avoid the distraction and etc.

7.3 What We Learned

7.3.1 Technically

- We gained experience of handling real-time images by learning and using OpenCV and Dlib libraries.
- Made use of Xlib to control the hardware of X Window system.
- Practised programming in C and C++.
- Used CMake tool to build a compilation environment.
- Became familiar with the GoogleTest C++ unit testing framework.
- Gained experience using test-driven design pattern to make sure the program achieved expected features.

7.3.2 Project Management

- Learned agile programming, in particular Kanban.
- Learned how to make our product releasable at every stage.
- Used Trello to keep track of the process at every stage of development.
- Used a divide and conquer method to manage people: Divide the whole project to groups first and combine when both groups finished developing features.
- Learned to manage a project where different members of the group had differing levels of expertise and interest in various parts (ie only some of the group studied computer vision).

Chapter 8

Bibliography

Thanks very much for the help from our supervisor Dr. Mark Wheelhouse!

1. OpenCV
   http://docs.opencv.org/master/d7/d8b/tutorial_py_face_detection.html#gsc.tab=0
2. Viola-Jones technique
   http://www.vision.caltech.edu/html-files/EE148-2005-Spring/pprs/viola04ijcv.pdf
3. dlib
   http://dlib.net/face_landmark_detection_ex.cpp.html
4. Xlib
   https://tronche.com/gui/x/xlib/
5. Unit testing C++ with GoogleTest
   https://blog.jetbrains.com/rscpp/unit-testing-google-test/
6. CMake
   http://doc.qt.io/qt-5/cmake-manual.html
7. Trello
   https://trello.com/guide
8. Kanban and agile development
   http://www.doc.ic.ac.uk/~rbc/302/
9. Facebook
   https://en-gb.facebook.com/
10. Overleaf
    https://www.overleaf.com/
11. GitLab
    https://gitlab.doc.ic.ac.uk/