Gaze-Based Paint Program with Voice Recognition

by

Jan van der Kamp, BA NUI Maynooth

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

September 2010

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Jan van der Kamp

September 13, 2010

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Jan van der Kamp

September 13, 2010

Acknowledgments

The author would like to thank Dr. Veronica Sundstedt for all her time, help and guidance during the project and also Jon Ward for both kindly lending a Tobii X-120 eye-tracker for use and providing support with its operation. The author would also like to thank Paul Masterson for all his help with any hardware issues encountered and all the participants who contributed their time to assist with the evaluation of this project.

Jan van der Kamp

University of Dublin, Trinity College
September 2010

Gaze-Based Paint Program with Voice Recognition

Jan van der Kamp

University of Dublin, Trinity College, 2010

Supervisor: Dr. Veronica Sundstedt

Modern eye-trackers allow us to determine the point of regard of an individual on a computer monitor in real time by measuring the physical rotation of eyes. Recent studies have investigated the possibility of using gaze to control a cursor in a paint program. This has opened up the doors for certain disabled users to have access to such programs which may not have been possible via the traditional input methods of keyboard and mouse. This dissertation investigates using gaze to control a cursor in a paint program in conjunction with voice recognition to command drawing. It aims to improve upon previous work in this area by using voice recognition rather than 'dwell' time to activate drawing. A system of menus with large buttons to allow easy selection with gaze is implemented with buttons only being shown on screen when needed. Gaze data is smoothed with a weighted average to reduce jitter and this adjusts automatically to account for saccades and fixations, each of which requires a different level of smoothing. Helper functions are also implemented to make drawing with gaze easier and to compensate for the lack of sub-pixel accuracy which one has

with a mouse. The application is evaluated with subjective responses from voluntary participants rating both this application as well as traditional keyboard and mouse as input methods. The main result indicates that while using gaze and voice offers less control, speed and precision than mouse and keyboard, it is more enjoyable with many users suggesting that with more practice it would get significantly easier. 100% of participants felt it would benefit disabled users.

Contents

Acknowledgments **iv**

Abstract **v**

List of Tables **x**

List of Figures **xi**

Chapter 1 Introduction **1**

1.1 Motivation 1

1.2 Objectives 3

1.3 Dissertation Layout 3

Chapter 2 Background and Related Work **4**

2.1 The Human Visual System 4

2.2 Eye Movements 6

2.3 Modern Eye Trackers 7

2.3.1 Eye Tracking Techniques 8

2.3.2 Video Based Eye Tracking 9

2.3.3 Calibrating 10

2.4 Modelling Eye Movements 10

2.5 Using Gaze in Paint Programs 10

2.6 Gaze and Voice Together 12

Chapter 3 Design **13**

3.1 Hardware 13

3.2 Main Development Tools . . . 13
3.2.1 COM Objects . . . 14
3.2.2 Microsoft Speech . . . 14
3.2.3 Tobii SDK . . . 15
3.3 Application Development Tools . . . 16
3.4 Application Design . . . 17
3.5 Application Layout . . . 20
3.5.1 Gaze System . . . 20
3.5.2 Voice Recognition System . . . 20
3.5.3 Drawing System . . . 21
3.5.4 Menu system . . . 22

Chapter 4 Implementation **23**
4.1 Framework . . . 23
4.1.1 Main Application Functions . . . 25
4.2 Voice Recognition . . . 26
4.2.1 `DrawingTools::InitSpeech` . . . 26
4.2.2 `DrawingTools::ProcessRecognitionEvent` . . . 27
4.2.3 `DrawingTools::ExecuteSpeechCommand` . . . 27
4.3 Gaze System . . . 28
4.3.1 `cTetClientSink::OnGazeData` . . . 29
4.3.2 Smoothing Gaze Data . . . 30
4.3.3 Modelling Eye Movements . . . 31
4.3.4 Calibration . . . 32
4.3.5 `DrawingTools::CleanUpEyeTracker` . . . 32
4.4 Drawing System . . . 32
4.4.1 Lines . . . 33
4.4.2 Rectangles and Ellipses . . . 34
4.4.3 Curve . . . 36
4.4.4 Drawing Tools Handler . . . 38
4.4.5 Flood-Fill . . . 39
4.4.6 Undo . . . 41
4.4.7 Helper Functions . . . 43

4.4.8 Menu System 45
4.5 Colouring In Mode 46
4.6 Data Output 47

Chapter 5 Evaluation **49**
5.1 Design of the User Evaluations 49
5.1.1 Participants 50
5.1.2 Questionnaires 51
5.1.3 Setup and Procedure 51
5.2 Results 53
5.2.1 Rankings Obtained 54
5.2.2 Participant Comments 57
5.2.3 Drawing Tools 59
5.3 Appraisal of Results 60

Chapter 6 Conclusions and Future Work **65**
6.1 Conclusions 65
6.2 Future Work 66

Appendix A Questionnaire **70**

Appendix B Instruction Sheet **74**

Appendix C Participant Data **78**

Bibliography **81**

List of Tables

3.1 Voice Commands. 18

5.1 Ease of navigating menus. 54
5.2 How much control was there? 55
5.3 How fast did you draw? 55
5.4 How much precision of the controls was there? 56
5.5 How enjoyable is it? 57
5.6 How natural are the controls? 57

C.1 Group One Participants. 79
C.2 Group Two Participants. 80

List of Figures

2.1 The Eye. Reprinted from [2], copyright [1]. 5
2.2 Peripheral Drift Illusion. Reprinted from [5]. 6
2.3 Example of electro-oculography (EOG) eye movement measurement. Reprinted from [2], copyright [12]. 8
2.4 Example of apparent pupil size. Reprinted from [2], copyright [12]. . . . 9
2.5 Left, EyeDraw. Reprinted from [4]. Right, EyeArt. Reprinted from [3] . 11

3.1 Setup for portable eye tracker. Reprinted from [27] 14
3.2 Drawing a rectangle 19

4.1 Main framework classes. 24
4.2 Assigning weights to gaze points for averaging 30
4.3 Left shows line being rendered in wireframe mode, Right shows line rendered with solid fillstate. 33
4.4 Top Left, thin rectangle. Top Right, thick rectangle. Bottom Left, thin ellipse. Bottom Right, constructing a thick ellipse. 36
4.5 Left, Catmull-Rom interpolation given 4 points. Right, modified version. 37
4.6 Left, first 3 points on curve. Right, finished curve. 38
4.7 Top Left, Tools Menu. Top Right, Start-Up Menu. Bottom Left, Pictures Menu. Bottom Right, Colouring In Mode. 46

5.1 Eye-tracker position. 52
5.2 Ease of Navigating the Menus. 54
5.3 How much control was there? 55
5.4 How fast did you draw? 55

5.5 How much precision of the controls was there? 56
5.6 How enjoyable is it? 56
5.7 How natural are the controls? 57
5.8 Ease of giving voice commands. 58
5.9 Favourite and Most Difficult Drawing Tools. 59
5.10 Group One Pictures. 63
5.11 Group Two Pictures. 64

6.1 Left, not drawing. Middle, drawing a line. Right, finished drawing. . . 67
6.2 Grid points to aid drawing 68

Chapter 1

Introduction

Eye trackers work by measuring where a person's gaze is focused on a computer monitor in real-time. This allows for certain applications to be controlled by the eyes which benefits disabled users for whom keyboard and mouse are not an option as input. This dissertation presents an application used for drawing on screen using an eye tracker for control. Voice recognition is also used which offers a method of confirming actions which is separated from eye movements.

1.1 Motivation

Throughout history human beings have learned how to deal with various disabilities in order to improve the quality of life that is possible. One of the harder things to deal with is the loss of one of the five senses. Imagine if you are blind, and unable to see the many different sights and colours of everyday life and imagine that that is the case for the rest of your life. People have learned how to deal with something like this by either using some sort of tool or helper aid to improve quality of life, or by passing some responsibilities for functions left out by this missing sense to one of the other five senses. For instance, a blind person may use a stick or a guide dog to help them navigate the world around them, or they may develop improved hearing in order to help with understanding what is going on. When braille was invented, the doors were opened for the blind to access the written word, which would have been unheard of previously. Similarly, a deaf person may use the new skill of sign language in order

to communicate with people quickly without having to write things down. This is a combination of using both hands and sight to make up for the inability to hear what is being said. For people with certain disabilities however, these already developed skills are not enough to improve quality of life and allow communication or expression.

These include sufferers of cerebral palsy, motor neuron disease, multiple sclerosis, autism, amputees and other physical paralysis. Since eye trackers allow us to determine where a person is gazing at on a computer screen, they open up new opportunities for such sufferers to interact with computers and control where they would like a cursor to be placed.

By using both eye tracking and voice recognition to control a paint program in this project, it is intended to replace the sense of touch and feel. This will be done by using the movement of a subjects eyes to control the position of the cursor, and voice commands to activate drawing. Using an eye tracker to control a cursor can be thought of as a new skill since we are used to using our eyes to look around, not meticulously place objects on a virtual canvas. It is the intention that this new skill coupled with the modern advent of high quality eye tracking and voice recognition systems will open up the doors of communication and expression to people with certain disabilities in a way that was not previously possible.

One of the main problems involved in gaze based interfaces is that of the "Midas Touch". This arises because our eyes are used to looking at objects rather than controlling or activating them [8]. When using gaze as input for a drawing program, this can lead to frustration as drawing can be activated without a user intending it to be. In previous work in this area, this has been overcome by using *dwell time* to activate drawing, which works by requiring a user to fixate their gaze at one point for a particular amount of time for confirmation. However it was found that this is not a perfect solution to the problem, due to both the delay involved and the possibility of drawing still being activated without intent [7]. The application developed for this project proposes a novel approach of using voice commands to activate drawing. It is intended that removing confirmation of drawing from gaze data will lead to improved user experience and drawing possibilities. It should also be quicker to draw since users will not have to wait for a *dwell* to be picked up in order for drawing to be activated.

1.2 Objectives

As mentioned, the approach of using voice recognition to activate drawing is novel as none of the previous work in the area of gaze based paint programs incorporates voice recognition. The application should be user friendly and improve upon previous work in terms of drawing experience. Gaze data is received from a Tobii X-120 eye tracker and interpreted in real-time using the Tobii SDK (Software Development Kit). Voice commands are interpreted using the Microsoft Speech SDK 5.1. The program should have many of the useful tools contained in basic paint programs such as Microsoft Paint. It is intended to implement line, curve, rectangle, polygon, ellipse, rounded rectangle, select, flood-fill, and text. Users will be able to save pictures once they are finished and all common image formats should be possible. The program will also contain some helper functions to make drawing with gaze easier and an option to colour in line drawings with gaze which may appeal to younger users. This is a novel feature, and will increase the overall appeal of the program.

The program will be evaluated by user groups who will test out the different features, and complete a questionnaire on their findings. The aim of these user trials is to gain an understanding of whether this program would be a valid alternative to keyboard and mouse for users that cannot use these traditional forms of input, and whether using voice recognition for confirming drawing can be done without many issues.

1.3 Dissertation Layout

The following chapter presents the background and related work important for this area of research. Chapter 3 introduces the design of the application and Chapter 4 describes the implementation. Chapter 5 examines the user evaluations and results and Chapter 6 continues with conclusions and future work.

Chapter 2

Background and Related Work

This chapter begins by giving an introduction to the human visual system which is relevant to eye-tracking. It then discusses different eye movements which are of interest, before moving onto different methods of eye-tracking in Section 2.3. Section 2.4 discusses how eye movements are modelled briefly before moving onto some previous work in Sections 2.5 and 2.6.

2.1 The Human Visual System

The human eye consists of several parts which act together to interpret visual data. This data is sent along visual pathways where it is processed in other regions of the brain. Incoming light enters through the cornea, which helps to focus it. The iris is in front of the lens and reduces the amount of periphery light coming through which reduces spherical aberrations which can occur [2]. It then passes through the lens, which is shaped by the ciliary muscles surrounding it. It is pulled flat to see objects far away, and the muscles relax to thicken the lens when focusing on closer objects [25]. After light passes through the lens it hits the retina where it is picked up by photoreceptors, which transform light energy to electrical impulses. The retina is curved to compensate for planar objects appearing rounded [2]. The photoreceptors in the retina can be split into two types, rods and cones. In total there are approximately 120 million rods, which are sensitive to dim light, and 7 million cones, which are sensitive to bright light [2].

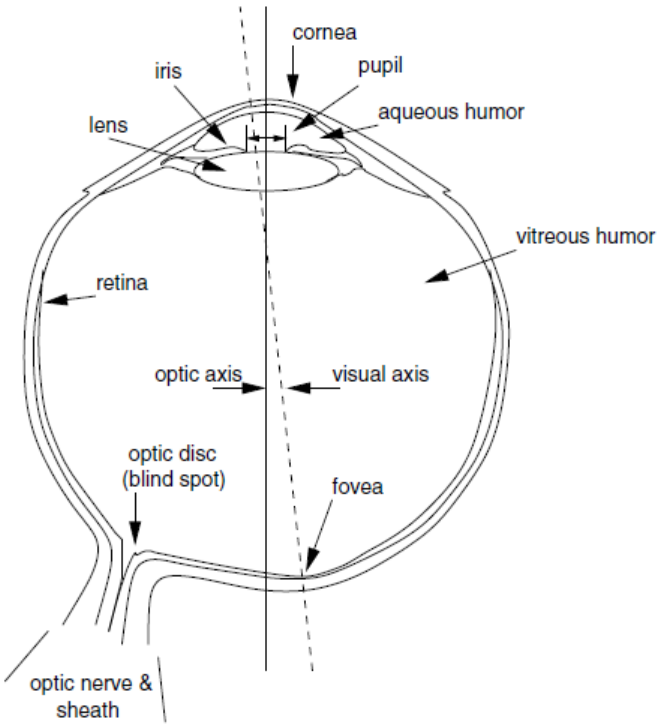


Figure 2.1: The Eye. Reprinted from [2], copyright [1].

If the light hits the small region known as the fovea, (see Figure 2.1) the highest level of visual acuity can be achieved, resulting in fine detail being noticed. If it falls outside this region, it lies in the area of peripheral vision, where large portions of a scene can be noticed, but with much less detail [2]. Figure 2.2 is an example of the how our peripheral vision can be less accurate than foveal, and is called the peripheral drift illusion [5]. No motion is perceived when looking directly at the image, but when looking in the area around the image, or by blinking rapidly, the circle appears to be turning slowly. Central foveal vision extends from about 1 to 5 degrees. This allows us to build up a picture of our surroundings by inspecting small areas of our field of view with brief fixations of gaze, which make up about 90% of viewing time [2].

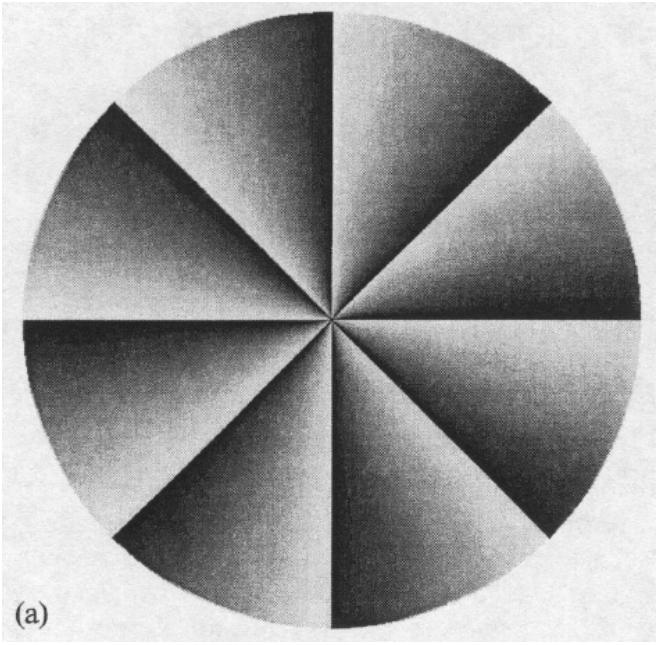


Figure 2.2: Peripheral Drift Illusion. Reprinted from [5].

2.2 Eye Movements

Information on eye movements as presented in this section was investigated in [2] and [28]. As mentioned in the previous section, when light hits the fovea, the highest level of detail is achieved, so when the eye looks around, light is repositioned on this, which corresponds to what we are looking at. The way that the fovea is repositioned is made up of five basic types of eye movements: saccades, smooth pursuits, vergence, vestibular ocular reflex (VOR), and optokinetic reflex (OKR). There are other non-positional types of eye movements such as those necessary for pupil dilation and lens focusing, but these are of secondary importance when it comes to eye tracking.

- Saccades are fast movements of the eye which are used to direct attention to a new part of the visual environment and they are voluntary and reflexive actions. The time taken can be between 10 and 100 ms. and virtually no visual information is obtained in this short time [2]. Saccades are also ballistic, which means that once the saccade has begun, the destination cannot be changed. This might be because there is not enough time for visual feedback during execution, or because instead of visual feedback, an internal copy of head, eye and target position are

used to guide movement.

- Smooth pursuits occur when the eyes are tracking a target which is moving visually. The velocity is smooth because a target which is moving will not, in the majority of cases be able to stop and start quickly enough to cause saccadic movements. It follows that the eye is capable of matching the velocity of the target in most cases [28].

- Movements of vergence are used to focus the eyes when looking at a target from a distance. There are two types, convergence which happens when looking from far to near, (the eyeballs are converging) and divergence which is the opposite [28].

- Vestibular ocular reflex movements are similar movements to smooth pursuits where the function is to keep the image still when the head moves. These movements still work when the eyes are closed, and this can be seen if one closes their eyes and moves their head while placing their fingers on their eyes [28].

- Optokinetic reflex movements assist VOR movements when the movement of the visual field is slow, and a sense of self motion is perceived. An example of this can be when sitting in a train and the train beside begins to move [28].

Of these five basic movements, only saccades and smooth pursuits are really of interest in the area of eye tracking. One other type of behaviour of the eye that is particularly of interest is the fixation.

Fixations are collections of movements that stabilize the retina over an object of interest which is stationary. They are characterized by tremor, drift and micro-saccades which are miniature eye movements. They work somewhat counter-intuitively in that these small movements move the image on the retina slightly, but this is necessary since if the image was completely stationary on the retina, it would fade away after about a second, but these small movements keep the image in view [2].

2.3 Modern Eye Trackers

The desired output from an eye tracker these days is estimating where a user is gazing at on screen, in x and y coordinates, which is referred to as the point-of-regard (POR).

Figure 2.3: Example of electro-oculography (EOG) eye movement measurement. Reprinted from [2], copyright [12].

First and second generation eye tracking systems which used eye in head measurement and photo and video-oculography respectively generally do not provide this data [2]. This section will describe various eye tracking techniques and discuss how modern ones can now be used to retrieve POR data in real time. Information on different eye-tracking techniques as presented in this section was investigated in [2].

2.3.1 Eye Tracking Techniques

Electro-oculography was the most widely used method for eye-tracking roughly 40 years ago and is still in use today. It involves measuring the skin's electric potential differences of electrodes which are placed around the eye [2]. A picture of someone wearing the apparatus can be seen in Figure 2.3. This method measures the position of the eyes relative to head position however, and so is not acceptable for POR measurements. It is also fairly intrusive, since it involves placing electrodes on the subjects face. The sclera contact lens or search coil method of eye tracking is one the most precise and involves placing a reference object which is mounted in a contact lens, and placing this on the eye. The movement of the reference object is then measured. The main method uses a wire coil which is measured in an electro-magnetic field. While this is very precise, it is naturally very intrusive, and also only measures eye position relative to the head.

Photo and video-oculography contain many different techniques for recording eye

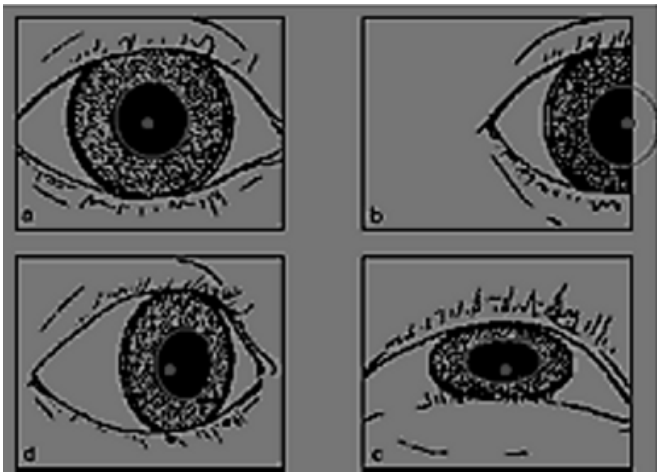


Figure 2.4: Example of apparent pupil size. Reprinted from [2], copyright [12].

movements and involve measuring distinguishable features of the eyes under rotation or translation, for example, the shape of the pupil as it appears to a camera (see Figure 2.4). Measuring these features can either be done automatically or manually, which is very tedius and prone to errors. In general these techniques do not provide POR.

2.3.2 Video Based Eye Tracking

Since the previous techniques do not provide the point of regard, this problem can either be solved by fixing the head in such a way that the eye movement matches up with the POR, or using computer vision techniques to separate eye movements from head movements. The latter is the more desirable option, since it is less intrusive. The apparatus used to retrieve image data can be either head or table mounted. In order to achieve this, the corneal reflection (known as the Purkinje reflection or image) of the light source (which is typically infra-red due to this light source being invisible to the subject) is measured with regards to the pupils centre [2]. By calibrating the system correctly, eye trackers which locate the Purkinje image are capable of retrieving a subjects POR on a planar surface such as a computer monitor. This technique of video based corneal reflection is the most widely used today for determining a subjects POR [2].

2.3.3 Calibrating

In order to calibrate these systems, it is necessary to prompt the subject to concentrate on a series of dots on the screen which appear at various points across the region of viewing. They are spaced out to cover as much of the region as possible, and the relative observed position of the pupil and corneal reflection at these locations can be interpolated across the whole viewing area to account for all gaze possibilities. A secondary use for the calibration procedure is to adjust the thresholds for pupil and corneal reflection detection. This ensures that these features are still detected while avoiding detection of artifacts such as eyelashes or contact lenses etc. [2].

2.4 Modelling Eye Movements

When modelling eye movements, one can look at the signal of movement over time. By summing the temporal signal and averaging the result, one can tell if the signal belongs to a fixation if little difference is found, otherwise it is likely to be a saccade [2]. Another method involves getting the velocity of the movement by subtracting successive samples [2]. If the velocity is below a certain threshold, then a fixation is likely to be happening, otherwise a saccade. The latter method is implemented in this project to decide to what extent the gaze signal should be smoothed by. More smoothing is applied during fixations, and less during saccades.

2.5 Using Gaze in Paint Programs

There are two significant gaze based drawing programs in existence at the moment, EyeDraw [7] and Eye Art [13], screenshots of these applications can be seen in Figure 2.5. Both programs have certain drawbacks that will be addressed in this project. EyeDraw is simplistic in operation, and in its current version (2.0) has drawing options for lines, squares, circles and certain clipart pictures. In [7], the authors mention that they have started working on version 3.0, but even though this was in 2005, version 3.0 does not seem to be published, or available. The fact that icons for selecting tools or menus are on screen at all times while staying large enough to select comfortably with gaze puts a limit on the amount that can be on the screen at once. As a result

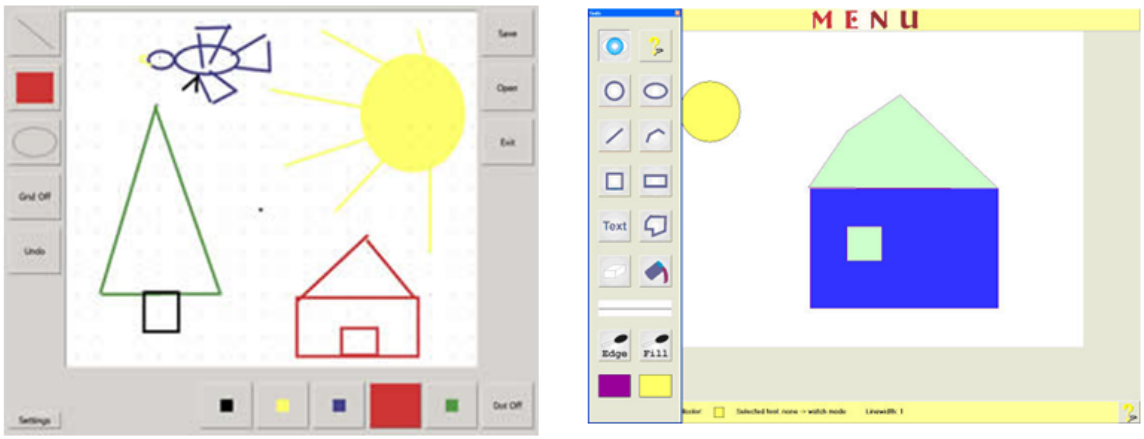


Figure 2.5: Left, EyeDraw. Reprinted from [4]. Right, EyeArt. Reprinted from [3]

this limits the scope of the application.

Also as they are along the side of the screen, sometimes users were found to be choosing them by accident [7]. In order to choose a drawing action, users needed to dwell their gaze on a point on the screen for 500 milliseconds, and for the same amount to confirm it, which led to frustration when trying to draw. This was due both to the inherent delay for each drawing command when using dwell time to activate drawing, and also drawing was sometimes activated by mistake if users gazed at a point for too long.

EyeArt was developed in response to EyeDraw, which was found to be missing some essential parts of a drawing program [13]. While it is a more substantial program, it seems that from watching a video on the EyeArt wiki page [3], it is also frustrating to use. Users still need to dwell their gaze for a fixed amount of time in order to confirm drawing so it is still a time consuming process. There are more drawing options such as fill, erase, text and polygon. This means scope for more complicated drawings, but since the icons are still constantly visible along the left hand side of the screen this requires them to be smaller. As a result, this introduces further frustration, since they are difficult to choose with gaze. There were also problems with accuracy in both programs, if a user wants to start a line from the endpoint of another line, the chances of hitting the point spot on with gaze are minimal.

2.6 Gaze and Voice Together

To date, there has been no published application that uses both gaze and voice recognition to control a paint program. In [22], a game using gaze and voice recognition was developed in 2009 . The main concept of this was to escape from a maze while collecting coins and shooting evil rabbits. When being controlled by gaze, a cross hair appears where a user's gaze is fixed on the screen, and by gazing towards the edge of the screen, buttons to change the orientation of the camera are activated. While one user thought that using voice commands to move felt slow, gaze was found to be an easy method for aiming the cross hair, and overall gaze and voice was found to be the most immersive form of interaction as opposed to keyboard and mouse. There were some issues with voice recognition where some words had to be substituted in order to be recognized properly. The word 'maze' had to be substituted for 'map,' and 'select' was also found to be inconsistent as a word to choose menu options.

Chapter 3

Design

This chapter aims to provide an understanding of how the project was designed and the tools which were necessary for this. Section 3.1 discusses the hardware used in the project, and Section 3.2 discusses the main development tools used with Section 3.3 covering the programming language and graphics API (Application Programming Interface) that were chosen. Section 3.4 explains the main design of the application, while Section 3.5 describes the layout of the overall program.

3.1 Hardware

A Tobii X-120 portable eye tracker was given to the college on loan from Acuity-ETS (a global reseller of Tobii eye trackers), for the duration of the project. This eye-tracker is positioned below the computer monitor. When provided with measurements of the computer screen, its height and tilt angle and the position of the eye tracker relative to the screen, accurate data pertaining to the users POR on screen can be obtained. A basic USB microphone targeted at speech recognition was used for interpreting voice commands. Figure 3.1 shows a basic setup with the portable eye-tracker.

3.2 Main Development Tools

This section begins by discussing COM objects which are used by the Tobii SDK to communicate with other software components. It then moves onto introducing both

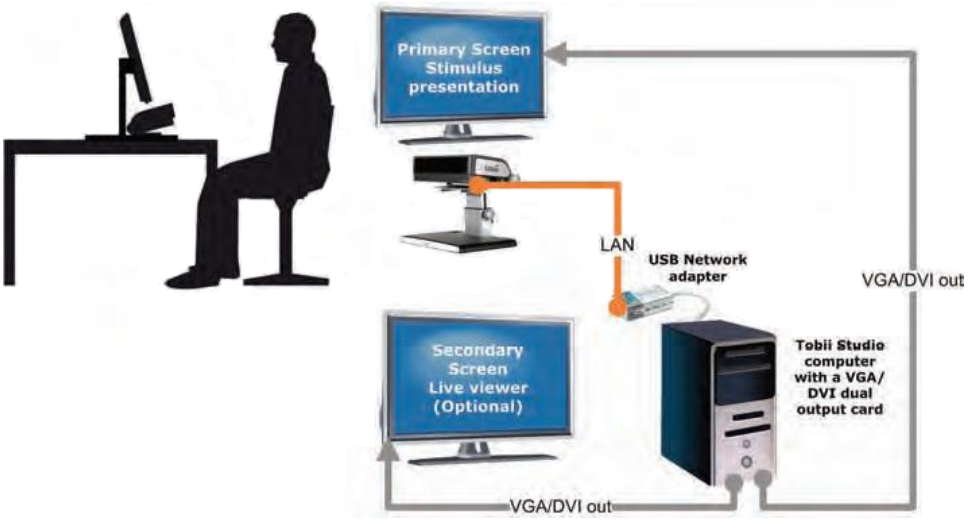


Figure 3.1: Setup for portable eye tracker. Reprinted from [27]

the Microsoft Speech SDK, and the Tobii SDK.

3.2.1 COM Objects

COM stands for component object model, and allows different software components to communicate with each other. They can be created with different programming languages and object oriented languages such as C++ simplify implementing COM objects [20]. COM does not indicate how an application should be structured in anyway, it is a standard that indicates how COM objects or components should interact with one another. Since it is a binary standard, this refers to the application once it has been translated into binary machine code, and thus is language independent [19]. It has one language requirement, which is that the language must be capable of creating structures of pointers and calling functions through pointers. COM objects control access to their data through sets of functions known as interfaces, (of which the functions are known as methods) and access to these methods is provided by a pointer to the interface [19].

3.2.2 Microsoft Speech

The Microsoft Speech SDK was used to recognize voice commands in the program. It provides a high level way to interface between an application and a speech engine, of

which there are two basic types, text-to-speech (TTS) and speech recognizers. The latter type is what is used in this project, and converts human spoken audio into readable text [18]. In this API, the main interface for speech recognition is `ISpRecoContext`, and there can be two types. The one which is used in this project is a shared recognizer, which can be shared with other applications should they need to use speech recognition. The other type is more appropriate for bigger applications that would run alone. Once a recognizer has been set up, it can register its interest in an `SPEI_RECOGNITION` event, which indicates speech has been recognized. Now the application can be set to recognize either dictation, where every word spoken is taken into account, or command and control, where only commands that are part of a file are interpreted. This project uses the latter, and words are input into an XML file, which is compiled to a binary grammar file and read by the application on startup.

3.2.3 Tobii SDK

The Tobii SDK is a collection of programming interfaces that allow different levels of access to the eye tracking hardware. Which API to use depends on the level of control required. As mentioned, a Tobii eye tracker retrieves gaze data from a subject in real time. The eye tracker hardware is controlled by the Tobii Eye Tracker Server software (`TETServer`) and access is given by either the Tobii Eye Tracker Components API (`TETComp`) or the Tobii Eye Tracker low level API. `TETComp` is a component object model (COM) implementation with lots of useful tools (for example, calibration). In contrast the low level API is just a dynamic link library (DLL) file which can be accessed from any programming language [26]. There is also the Tobii ClearView application which is a suite of tools used for gaze data recording which in turn can be used by clicking buttons in its GUI (Graphical User Interface), or programmatically by using further APIs. It was not difficult to choose which API to use for this project, the aim was to retrieve from the eye tracker the point of regard of the subject while avoiding unnecessary complexity. This definitely ruled out the low-level API which does not include any GUI tools itself. `TETComp` proved more than adequate for the task since it was an option to use COM. The main component necessary to retrieve gaze data is the `TetClient` [26]. This is a COM object which is a wrapper for the low-level API, but retrieves data in the form of firing events. This means that the programmer can

define an event to occur whenever gaze data is retrieved from the eye tracker (this is usually at the highest frequency supported by the hardware) and in this event update a variable defining where the cursor should be placed for example. This provides a convenient way of encapsulating this important piece of functionality.

3.3 Application Development Tools

It was decided to use C++ as the programming language due to both its high performance, and the author's familiarity with the language. Several solutions were considered for rendering the graphics, namely OpenGL, DirectX, SDL, and Windows GDI. Windows GDI stands for Graphical Device Interface, and allows applications to use graphics and text on the video display [21]. As a paint program would be developed, Windows GDI was considered first due to its inbuilt support for drawing primitives quickly. A line can be drawn with one simple command. This was quite positive due to the fact that shapes could simply be started whenever voice commands were given. It was also found to be possible to perform a flood-fill operation with a couple of lines of code, which would have simplified the implementation of this drawing tool considerably. However, there seemed to be a relatively low level of fine-grain control over the graphics, and once moving images were necessary (such as the cursor) it became very awkward to use.

The next possibility investigated was SDL which stands for Simple DirectMedia Layer. SDL offers low level access to a video framebuffer, amongst other things with support for many operating systems [10]. It was investigated due to it being a 2D graphics API and good support for texturing operations, which would be handy for menu buttons etc. However due to the fact that it does not take advantage of hardware acceleration and performs all drawing on the CPU (Central Processing Unit), it not chosen. The OpenGL API was also investigated. This is a software interface which allows application to communicate with the graphics hardware [17]. It has support for drawing lines with simple commands, but in order to take advantage of GPU (Graphics Processing Unit) optimization it would be necessary to draw thick lines as narrow rectangles and pass the vertices of these rectangles to the GPU, so this was not an advantage. OpenGL allows communication with the GPU via the shading language GLSL (OpenGL Shading Language) [23] but it is not near the level of stability of its

counterpart developed by Microsoft, HLSL. Due to both this and the fact that the author had experience working with HLSL, it was decided to use the DirectX API.

3.4 Application Design

As mentioned previously in Section 1.2, the main objective of this project is to create a paint program that is created by gaze and voice that is both intuitive and easy to use, with the aim of overcoming some of the problems encountered by EyeDraw and EyeArt. The application should use voice recognition to process commands, and when a user is drawing or examining what is on screen, the menu should not be visible. This would overcome the problem of having menu buttons constantly on screen which could be distracting for the subject and may be activated by accident. If the menu command is spoken, a menu containing buttons for selecting drawing tools would become visible, where a button could be chosen by hovering the cursor over a button and saying a word of confirmation.

As mentioned in Section 1.2, initially the plan was to include line, curve, rectangle, polygon, ellipse, rounded rectangle, select, flood-fill, and text. During early development rounded-rectangle was decided against due to its relative similarity to rectangle. Select was also left out as it was thought that this would be difficult to use with gaze and that it would be easier for users to rely on and get comfortable with using an undo command. This also removed the prospect of unnecessary complexity from the application. Text had been viewed as one of the tools with a lesser importance than most, and towards the end of the project, this was also excluded in favour of concentrating on curve, which had not been implemented in a gaze based paint program previous to this. Other tools present in Microsoft Paint like free-form select, erase, and brush were not considered due to the lower precision afforded by current eye tracking solutions. It was also desirable not to have too complicated an interface to retain ease of use. It was important that all aspects of the program worked in a similar way to avoid confusion for subjects so other menus were designed to be used similarly to the tools menu. There should be a menu for colours, and a menu for changing line thickness, each of which would be easily accessible by speaking a voice command, after which a button could be chosen by giving a confirmation command. If no change was required, the user would need to be able to exit from the menu and go back to drawing with

another voice command.

Voice Command	Action
Start	Starts drawing a shape
Stop	Stops drawing a shape
Snap	Starts drawing a shape at nearest vertex
Leave	Stops drawing a shape at nearest vertex
Undo	Removes the most recent drawing action
Fix	Fixes the current line to being vertical/horizontal
Unfix	Allows line to be drawn at any angle with x axis
Open Tools	Opens the tools menu
Open Colours	Opens the colours menu
Open Thickness	Opens the line thickness menu
Open File	Opens the file menu
Select	Chooses a menu button
Back	Exits from current menu screen

Table 3.1: Voice Commands.

When considering the actions necessary for drawing the shapes themselves, it was decided to keep them relatively similar to those in mainstream paint programs where possible. For example, a line would be started by saying 'start' whereby a line would be added to the screen going from the point at which the user was gazing when saying 'start', to the current gaze point. Saying 'stop' would stop the current line. Rectangle would work by the user giving two points by saying 'start' and 'stop' and a rectangle containing horizontal and vertical lines would be drawn based on these two corners, as seen in Figure 3.2. Ellipse would work by drawing an ellipse in the area described by an imaginary rectangle from these two points. Polygon is just an extension of the line command, so whenever a user said 'start' while drawing a line, the current line would stop, and a new line would start at this point. During development it was decided to implement polyline instead of polygon which just meant the shape would not be closed when finished giving more choice to users. A table containing all possible voice

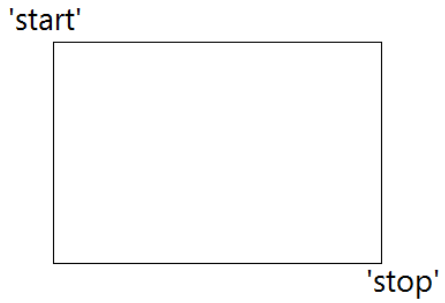


Figure 3.2: Drawing a rectangle

commands is shown in Table 3.1

Curve needed careful consideration due to the fact that gaze was being used as input. In mainstream paint programs, bezier curves are usually implemented, where the user draws a line, then drags two control points from either side of the line to produce a curve. It was thought that this would prove too difficult to accomplish with gaze however, and a simpler method of control was decided on. Users would specify four control points and a curve would pass through all four, with the first and fourth being start and end points on the curve. By having the curve pass through all control points, it would be easier to visualize how the curve would look when finished, which hopefully would avoid further modification. There is a function in DirectX called `D3DXVec3CatmullRom` which uses Catmull-Rom interpolation to generate points along a smooth curve passing through the control points which met this requirement.

It was also desirable to have an option for colouring in line drawings with gaze as it was felt that this would be a very enjoyable feature for younger users. This type of feature would allow users to complete nice looking pictures in a much shorter period of time then using the drawing tools, which would help in acclimatizing oneself to this new input method for the first time. This feature is implemented as a separate mode, which can be chosen when starting up the application. After choosing this mode, users would be able to choose a line-drawing which would take up the whole screen. The drawing tool would then be set to flood-fill, and users would be able to fill in the line drawing with different colours, and save the picture when finished. The program was also designed to output data at certain time intervals which could be looked at to analyze user behavior while evaluating the project.

3.5 Application Layout

As mentioned in Section 3.3, the DirectX API was used to render graphics. In order to provide a display, windows programming is used. A custom framework was investigated in [11] which allows encapsulation of the basic code required to run a Direct3D application without having to start everything from scratch. It includes functions for creating the main window for the application, running the message loop, handling messages to the window, initializing Direct3D, handling devices which are lost, and also enabling full-screen mode [11].

3.5.1 Gaze System

As mentioned in Section 3.2.1, `TetCOMP` is a component object model implementation, and once it has been set up and registered, events are fired which can be subscribed to. Two COM objects in `TetCOMP` are used in this application. `ITetClient`, and `ITetCalibProc`. `ITetClient` facilitates communication with the `TETServer`, and in turn the eye tracker hardware. `ITetCalibProc` allows an application to perform calibration of an individual, and save this calibration to the eye-tracker in use. In order to interface with these objects, 'sinks' must be implemented which allow the application to subscribe to events which are fired. Two are implemented, one for each of the COM objects mentioned, and the first contains is responsible for updating a variable which holds the screen position whenever gaze data is received. It is also responsible for smoothing the gaze signal to reduce jitter, this will be discussed in further detail in Section 4.3.2. The other event sink facilitates performing a calibration of nine points when the relevant menu button is selected. It was decided to keep the number of points to calibrate nine which would provide the most accuracy.

3.5.2 Voice Recognition System

This project required a few simple voice commands to activate drawing and menus, so a simple command and control system was necessary. An XML file is filled with the relevant words or sentences, and this is compiled to a grammar file using the grammar compiler tool which is bundled with the Microsoft Speech API. A new windows event is defined and when audio occurs, the message loop of the main window will fire this

event. From here, the main application calls the relevant functions which process the audio.

3.5.3 Drawing System

It had been decided to use the GPU to perform drawing in order to take the load off the CPU. Each drawing tool is implemented with its own class, and contains variables for its vertices, which are sent to the GPU to be drawn. The shapes that have been already drawn by the user are kept track of by the `cTools` class, which holds containers of each shape, each of which is added to when a new instance of a shape is required. For example, when the current drawing tool is line and a user says 'start,' `cTools` adds a new line to its collection, with the first point being where the cursor was at the time of selection. The second point of the line is going to be where the users POR is now, so while they look around they can choose a point where to end the line, and the line will move around to join the first point with the current gaze point until the word 'stop' is uttered. To facilitate moving the line, a function is implemented which locks the shapes vertex buffer and updates the position of its vertices based on the current gaze point. This works in a similar way for rectangle and ellipse which are drawn by giving two points. Each time their second point is given, the positions of all vertices on that shape are re-computed based on the point given with 'start' and the new gaze point, and the GPU is updated with these new positions, which allows fine-grain control over the exact appearance of shapes without sacrificing performance.

As mentioned in Section 3.4, the curve is implemented by giving four points, after which the curve is calculated. This means no locking of the vertex buffer is necessary as with the other shapes. In order to implement the flood-fill tool, it was necessary to access pixel data of the screen at the time of selection. Since it is not possible to access the front or backbuffer with reasonable performance with DirectX, it is necessary to render the whole screen to a texture and read the data from this texture. This process was investigated in [11] which contained code which performed everything necessary to implement this and is implemented in the `DrawableTex2D` class. The pixel data from the texture can then be updated with a flood-fill algorithm and a new updated texture can be displayed on screen underneath any shapes which are to be drawn.

3.5.4 Menu system

The menu system was investigated in [6]. Even though this book was aimed at C sharp/XNA developers, the author found it useful as there was a chapter dealing just with implementing a menu system. There are three classes which make up the menu system. The first one is the **Button** class which is used to represent any of the buttons which are used to activate parts of a menu. There is also a `MenuScreen` class which is used to represent a series of buttons which make up a menu. Lastly there is the `MenuSystem` class which is in charge of all the menus and is responsible for drawing them. When an instance of a **Button** is created, it is either passed the address of another `MenuScreen` (if navigation to another menu is required) or the address of a function to perform (for example, change drawing tool to rectangle), or both. It is also passed coordinates of where to appear on screen and texture coordinates specifying where in an image file to sample image data. A `MenuScreen` holds a collection of `Buttons`, and is responsible for performing its `Button's` function if it has one.

Chapter 4

Implementation

This chapter starts by giving a high level overview of the main application class along with short descriptions of its functions. It then proceeds to describe the implementation of the voice recognition system, the gaze system, the drawing system, the colouring in-mode and the data output feature.

4.1 Framework

As mentioned in Section 3.5, a custom framework is in place which the main application can inherit from. The main class in the application is called `DrawingTools`. This inherits from the custom framework class D3DApp from [11]. It is responsible for creating all relevant objects, running the update and message loops, and has a function to draw to the screen. Since the application is a Windows program, it starts off in the `WinMain` function where an instance of the `DrawingTools` class is constructed. Drawing objects involves specifying vertices for shapes in 3D space, and giving them all a z-coordinate of 0. This has the effect of having a virtual canvas on which drawing is performed. The camera is positioned ten units in front of this looking in the positive z direction, in order to view what has been drawn. Figure 4.1 shows the main classes in the application.

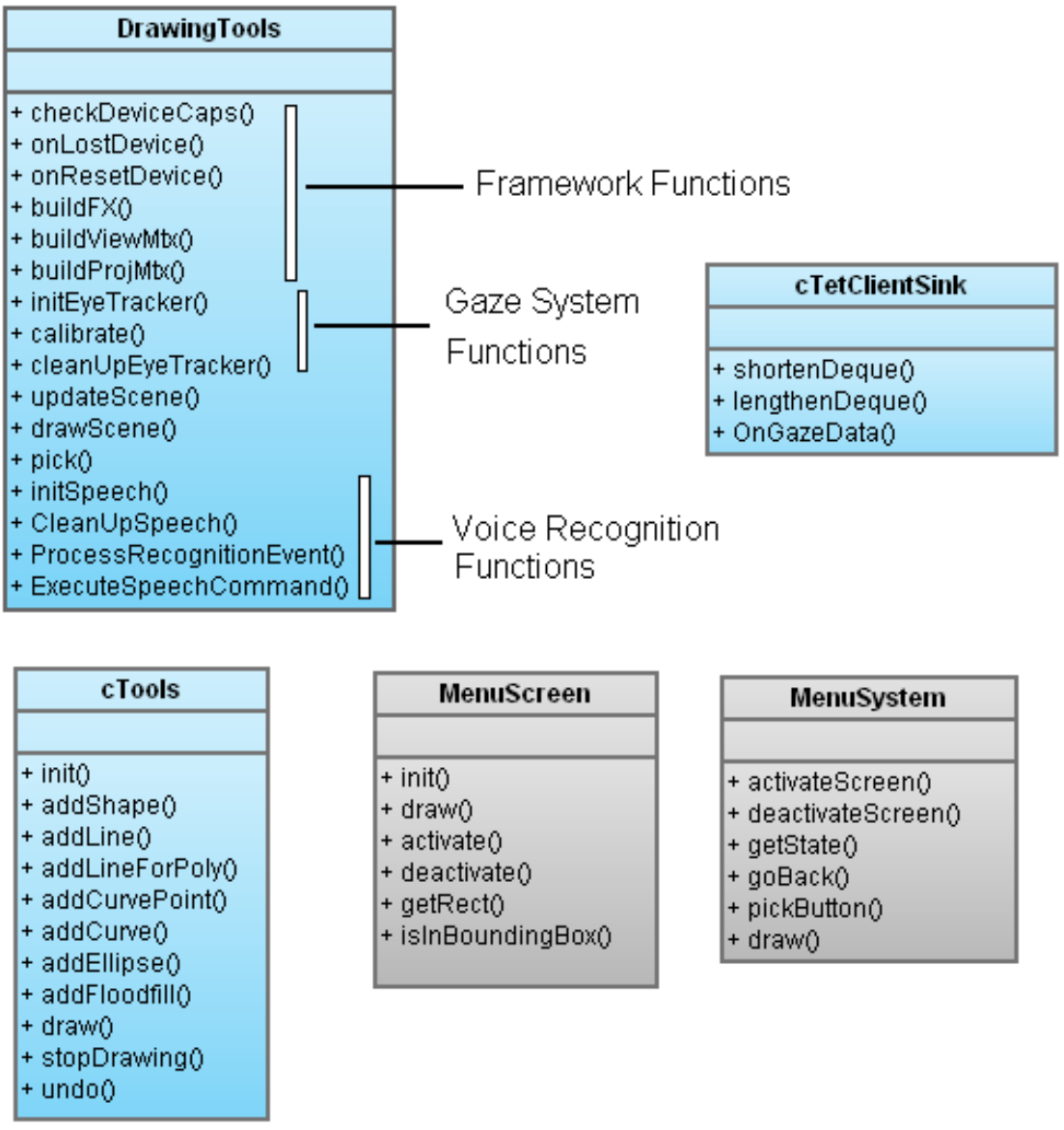


Figure 4.1: Main framework classes.

4.1.1 Main Application Functions

Since the main application inherits from the framework class `D3DApp`, the first and foremost purpose of it is to run the framework of a Direct3D application. A lot of its member functions are there for framework initialization (`::checkDeviceCaps`, `::buildFX`, `::buildViewMtx`, `::buildProjMtx`). For example, `D3DApp` has its own function for listening to messages (`D3DApp::msgProc`) sent to the window such as when the user tries to resize it or if they press the 'X' button in the top right hand corner to exit. This function is overridden by `DrawingTools::msgProc` to also listen to messages more specific to this particular application such as a spoken word being recognized. Other important framework functions include `DrawingTools::updateScene` which runs the main update loop of the application and `DrawingTools::drawScene` which draws all objects and menus and performs any rendering to textures that are necessary. (The flood-fill operation involves rendering what is on screen to a texture and altering this texture, see Section 4.4.5.)

As mentioned in the previous section, drawing is performed in 3D on a virtual canvas. In order to convert screen coordinates (which refer to the pixels on screen) to coordinates in 3D-space, `DrawingTools::pick` was written. This involves getting the point on the projection window which corresponds to the screen-coordinates, evaluate the ray which passes from the camera through this point, which is then transformed from view space to world space. Then it is just a matter of following along this ray from the camera, until the z-coordinate equals 0. This is the point on the virtual drawing canvas which corresponds to the screen-coordinates given. This function can then be called whenever screen coordinates such as the position of the gaze point need to be converted to positions on the virtual canvas.

The main application class also contains functions which initialize and process voice commands, these will be discussed in the next section. The gaze system involves implementing event sinks which listen for events. The main application class sets up what is necessary to do this, and interfaces with these sinks, and will be discussed in Section 4.3.

4.2 Voice Recognition

The Microsoft Speech SDK comes with comprehensive documentation which provides samples and tutorials explaining how to build speech applications. This provided useful information which helped with the task of integrating voice recognition into this project. As mentioned in Section 3.5.2, a simple command and control recognition system is necessary to interpret the few voice commands that would be used in this project. The main work involved was initializing the speech system and getting ready for recognition, after which it could be tweaked and updated with new words very easily. The voice recognition system is part of the overall application class, and the functions pertaining to it are:

```
DrawingTools::InitSpeech
DrawingTools::CleanupSpeech
DrawingTools::ProcessRecognitionEvent
DrawingTools::ExecuteSpeechCommand
```

4.2.1 `DrawingTools::InitSpeech`

The `InitSpeech` function initializes everything necessary for the voice recognition system. It begins by creating a recognizer object and uses this to create a recognition engine with `CLSID_SpSharedRecognizer` as an argument, which creates a shared recognition engine relevant to this project. By creating a shared instance of a recognizer object, resources such as recognition engines, microphones and output devices can be used by different applications at the same time [14]. The next step is to create a recognition context which will be used to interpret commands. A context is a part of an application that is used to process speech. In this application, one context is used, so at any point in the application, all possible voice commands can be interpreted and are handled by the same procedure [14]. Since this application uses a relatively small number of voice commands, this was found to be satisfactory.

`SetNotifyWindowMessage()` is then called by the recognition context, and this function tells the context that the application should send events to the main applica-

tion window, using a user-defined message `WM_RECOEVENT`. The next step is to set the interest for the recognition context. If nothing is set, the speech API sends all events back, which is more than 30 [14], and in this application the only event that is of interest is successful word recognition, or `SPEI_RECOGNITION`. Once this is done, a grammar is created and loaded. A grammar specifies which words will be recognized in command and control recognition, and is compiled into a binary .cfg file to be used by the API. This is done by compiling an XML file with the Grammar Compiler tool which ships with the Speech SDK. The grammar is then set to active and the application is ready to interpret speech.

4.2.2 `DrawingTools::ProcessRecognitionEvent`

Once the user-defined message `WM_RECOEVENT` has been defined, this can be used in the main window's message handler in the same way as events such as `WM_MOUSEMOVE`. The latter notifies the main application window that the mouse has moved, and similarly, `WM_RECOEVENT` is called in the message handler whenever a successful word recognition occurs. At this point, only the fact that a recognition has occurred is known, the application does not know which word has been recognized. In order to investigate further, `ProcessRecognitionEvent` is called following `WM_RECOEVENT` being fired, which extracts the recognition and processes it. Since there can be more than one recognition event in the queue (for instance if a person said many words quite quickly), this function loops through the whole queue in order to get through them. It extracts what type of event it is, and in the case of a word or sentence from the grammar being recognized (`SPEI_RECOGNITION`), calls `ExecuteSpeechCommand` on this word.

4.2.3 `DrawingTools::ExecuteSpeechCommand`

When voice commands are input into the XML file which is compiled to a binary grammar file, each one is given a code number, or Rule ID to define itself. Once the recognition event has been processed and flagged as being a successful recognition it is processed in the `ExecuteSpeechCommand` function. All that is necessary now is to perform a switch statement on the Rule ID to identify exactly which voice command from the grammar has been spoken. If the Rule ID corresponding to the word 'start' has

been recognized, the application starts drawing a shape depending on which drawing tool is currently selected. An example of this can be seen in the following code example:

```
#define VID_Start 4


switch ( pElements->Rule.ulId )
        {
            case VID_Start:
                    // start drawing
                    break;
        }
```

The only other function related to voice recognition is `CleanUpSpeech` and as expected, cleans up the voice recognition system. It is called in the destructor of the main application class. It checks whether the grammar, recognition context, and recognition engine have been loaded, and if so, releases them.

4.3 Gaze System

As mentioned in Section 3.2.1, two COM objects from the `TetCOMP` API are used in this project, `ITetClient` and `ITetCalibProc`. Each has its own event sink implemented, `cTetClientSink` and `cTetCalibProcSink`. Each of these was researched from one of the example programs provided by the Tobii SDK, C++ components example. To implement the gaze system, the main application class has pointers to both COM objects and their respective sinks. The function `DrawingTools::initEyeTracker` initializes what is necessary to start receiving gaze data from the eye tracker hardware. First, `TetClient` and `TetCalibProc` objects are created followed by their event sinks. The function `AtlAdvise` is used to subscribe on each sink's events, and then the `TetClient` connects to the eye tracker itself using the eye tracker's address.

During the course of this project, the eye tracker address was hardcoded inside the `initEyeTracker` function, since only one eye tracker was being used. It would be quite straightforward however to make this changeable in a settings menu. Each of these commands is checked for success using the `HRESULT` returned back, and if they have been successfuly so far, the `TetClient` is set to start tracking. At this point the eye tracker starts shining infra-red light and interpreting the gaze data from the subject. It starts receiving gaze data regardless of whether it has been calibrated yet or

not, and it can be calibrated from with this application or from an external application with the same result.

4.3.1 `cTetClientSink::OnGazeData`

As discussed in Section 3.5.1, the `TetClient's` event sink is used to update a variable representing cursor position, and this is done in the `OnGazeData` function. Once the application has subscribed to the `cTetClientSink's` events, this function will be called with the maximum frequency allowed by the eye tracker hardware. It accepts a pointer to a `TetGazeData` structure, which contains information on one gaze data sample. It first checks to see that the validity code for both eyes is 0, which indicates that the eye tracker is certain that the data received for an eye definitely belongs to that eye [26]. This means that if one eye is closed or not found, the application will not process the data and update the cursor. This limits the effectiveness of the application somewhat, but was not found to be an issue while testing and thus was not pursued. It could be modified without much difficulty to accept data from one eye if this was all that was received.

If the validity of the left and right eye are both valid, then the average is taken of the X and Y coordinates of both eyes, and this gives coordinates in the range of zero to one. In the case of X, zero indicates the very left of the screen with one indicating the right, and with Y zero indicates the top of the screen and one the bottom. Coordinates of (0.5, 0.5) would indicate the very middle of the screen. By multiplying these coordinates by the current screen dimensions, screen or pixel coordinates are found, which is what is needed to update the cursor on screen. If the main application window is smaller than the monitors screen dimensions this would introduce errors and would need to be taking into account. Due to the fact that user studies were intended to be run with the application in full screen mode and that full-screen mode suited the application best due to having less distractions present, it was decided not to account for having a smaller window. When the application starts it enters full-screen mode immediately and cannot be set to a smaller window.

4.3.2 Smoothing Gaze Data

The `OnGazeData` function was left in this state for some time during implementation, with the intention that a solution would be found to smooth the gaze signal. As mentioned in Section 2.2, the human vision system naturally has quick micro-saccadic eye movements that are necessary to keep the image still on the retina. Together with other noise this resulted in jittery movements of the cursor which made drawing difficult. A method of smoothing was researched in [9] which used a weighted average to smooth the gaze data.

$$P_{fixation} = \frac{1P_0 + 2P_1 + ... + nP_{n-1}}{1 + 2 + ... + n} \tag{4.1}$$

This is shown in equation 4.1 which is reprinted from [9]. It can be visualized in Figure 4.2, where the five most recent gaze points are given weights depending on how recent they are. In this case they are given their respective weights, and averaged by adding together and dividing by 15, which is the sum of the weights. In order to maintain a list of constant length of the most recent gaze points, it was necessary to add the current gaze point to one end of a container and delete the least recent one from the other end once the container had reached a pre-defined limit. To do this efficiently, a container from the C++ standard library known as `std::deque` which stands for double ended queue was used.

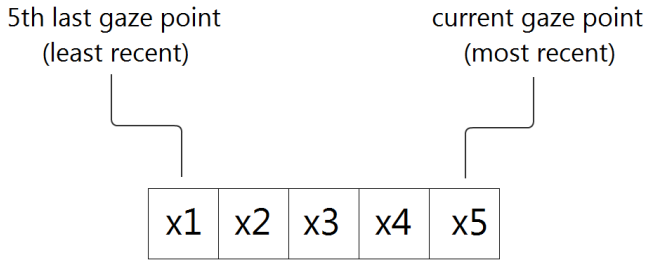


Figure 4.2: Assigning weights to gaze points for averaging

This container allows elements to be inserted or deleted with the same efficiency at both the front and back. This facilitated efficiently adding recent gaze points to one end, and removing the oldest points from the other end to maintain a constant amount. By limiting the length of this container to 5, stability of the cursor improved

dramatically, while increasing it to 15 gave a good tradeoff between stability and responsiveness. If it was increased to 20, the cursor began to drag somewhat which hampered drawing when quick eye movements were necessary.

4.3.3 Modelling Eye Movements

When considering the two extremes of this weighted average (with no averaging producing jittery but immediate response and averaging with at least 20 points producing delayed cursor movements with no jitter), it was noticed that the former case suited quick saccadic eye movements where speed of response is of primary importance. The latter case suited fixations, where the cursor would not be moving far or fast, and stability of movement was of primary concern. The application needed to know when fixations or saccades were being made, and seamlessly modify the smoothing algorithm to take these changes into account. As mentioned in Section 2.4, measuring the velocity of the cursor across the screen was used to interpret which movements were being made. The program was already keeping track of roughly the 15 most recent gaze points, but these were being fired with such high frequency that the velocity had to be measured separately.

The program was modified to keep track of gaze points in discrete intervals of 100 milliseconds. By measuring the distance in pixels between the most recent gaze point and the gaze point from 100 milliseconds previous to this, the velocity of the cursor in pixels per 100 milliseconds could be evaluated. A velocity of 100 pixels per 100 ms was arbitrarily chosen as a threshold to separate fixations from saccades. If the velocity was above 100, the movement was flagged as a saccade and the amount of gaze points to average was set immediately to 15. Fixations were harder to account for. If the amount of gaze points to average was simply set to a large amount straight away, there would be a lot of empty elements which made the cursor unstable for a short length of time. By simply incrementing this amount by one every 100 milliseconds, a smooth increase was achieved. The result of this was that by fixating gaze within a window of 100 pixels by 100 pixels for one second would increase the amount smoothly by ten. An upper limit of 50 was put in place and this provided a seamless way of providing both quick response and great stability automatically when necessary.

4.3.4 Calibration

An event sink is implemented to listen to events during calibration. The only event which is relevant is `cTetCalibProcSink::OnCalibrationEnd` which removes the calibration window when it has finished. Actual calibration takes place in the main application in the `DrawingTools::calibrate` function. When a user chooses to calibrate the eye tracker, this function is called, which starts by pausing the main application. It then checks to see if the eye-tracker is connected, if not it attempts to connect to it. The number of points to calibrate with is set to nine, before setting the calibration window to being visible. The eye tracker then starts calibrating by calling `TetCalibProc's StartCalibration` function. This consists of a blue circle moving across the screen to nine different points. When it reaches a point on the screen it shrinks in size in order to attract the attention of the user, and it stays at a point until it has received enough gaze information from the user [26]. Since the high-level `TetComp` API was being used this was possible to implement with minimal difficulty as the GUI is taken care of by the Tobii SDK.

4.3.5 `DrawingTools::CleanUpEyeTracker`

This function cleans up and releases anything related to the gaze system in the application. It is called in the main applications destructor directly before CleanUpSpeech and begins by stopping the eye tracker from tracking the gaze of the user. The application then unsubscribes from the events of the `cTetClientSink` and `cTetCalibProcSink`. The `TetClient` and `TetCalibProc` are disconnected from the eye tracker before releasing all four COM objects. The pointers to these four objects are then set back to NULL before the `CleanUpEyeTracker` function returns.

4.4 Drawing System

The drawing system for the paint application is implemented in 3D on a virtual canvas, with all shapes being given a z coordinate of zero. The shapes are implemented by specifying individual vertices which are then sent to the GPU to be drawn. There is a separate class for lines, rectangles, ellipses, and curves. The polyline tool is simply made up of several lines. Lines will be discussed first, as they are the basis for much of

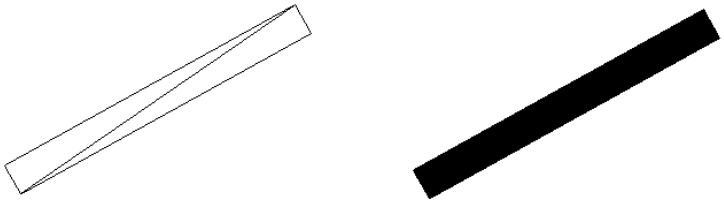


Figure 4.3: Left shows line being rendered in wireframe mode, Right shows line rendered with solid fillstate.

the other drawing tools. This section will then describe some of the other drawing tools before discussing the drawing tool handler which keeps track of the shapes. The flood fill tool will be discussed last as it was the most difficult drawing tool to implement and took longer than any of the other tools.

4.4.1 Lines

Lines were the first tool to be implemented and only thin lines of one pixel thick were possible at first. At this time lines were drawn as a line list primitive between two points specified by the user. In order to implement thicker lines, it was necessary to specify the four vertices of a quad, and index the two triangles which make up this quad.

This is shown in Figure 4.3, where the two triangles making up a thick line can be seen on the left, and a filled in thick line is shown on the right. Lines are implemented with the `CLine` class. There is only a default constructor available for this and other shapes, and whenever one has been created, the function `::init` is called. This function takes as arguments a pointer to the main `IDirect3DDevice9` for the application, a D3DXVECTOR3 (3 dimensional vector) specifying the first point of the line, a D3DXCOLOR specifying line colour, and a `float` specifying thickness. It was attempted to draw the thinnest lines with the more elaborate method of two triangles which would have simplified the code somewhat, but it was not possible to make them exactly one pixel thick as it had been with the original method of using a line list primitive.

In order to achieve this functionality, a check was put in place at the start of the `::init` function. If the variable `thickness` is equal to `0.0f`, the number of vertices

and indices are set to two each, and `mPrimType` is set to `D3DPT_LINELIST` for drawing lines. If `thickness` is greater than **0.0f** however, the number of vertices is set to four, the number of indices is set to six and `mPrimType` is set to `D3DPT_TRIANGLELIST` in order to draw the two triangles which make up a thick line. Once the internal variables for drawing the relevant thickness of line have been set up, a lines own vertex buffer is created and then locked in order to write the vertex data. An arbitrary second point is chosen which is more less right beside the first point provided to the function. (This point will be overwritten almost immediately with the `::setPoint2` function which is called continually while the user is choosing where to finish the line) If `thickness` is equal to zero, the 2 vertices are simply set to these two points. If not, the four points on the quad need to be found. This is done by first getting the vector from point1 to point2 and getting the perpendicular vector to this which is then normalized and multiplied by `thickness`. This perpendicular vector is then both added and subtracted to both end points on the line to find vertices on the quad. The next step is creating the lines own index buffer which is trivial for the thinnest lines, only two indices are needed to index the one line primitive. For thicker lines two triangles need to be indexed.

So at this point the line has been created and initialized with its first point defined by where the user was gazing at on screen when saying 'start'. Almost immediately the second point needs to be overwritten with where they are gazing at currently, which is done with the `::setPoint2` function. This function simply takes the current gaze point and repeats the process of locking the vertex buffer and updating all vertices based on the 2 end points of the line which took place in the `::init` function. It could be said that there is not much point in specifying vertices based on an arbitrary second point in the `::init` function. However it was necessary to specify some vertices in order to be drawn, even if they were going to be overwritten almost immediately with `::setPoint2`. Also, during development the process of making sure the correct vertices were being specified for the thick version of a shape was made easier by doing this in `::init`. These could then be carried over with confidence to `::setPoint2`.

4.4.2 Rectangles and Ellipses

Section 4.4.1 discussed how lines were implemented in great detail. It is not necessary to discuss Rectangles or Ellipses in as much detail since they work very similarly

to lines. Instead, the differences in creating these shapes will be discussed. When `CRectangle::init` is called, `thickness` is checked first as with lines, to see if the shape should be one pixel thick or more. If it is `0.0f`, the shape will be drawn with the line list primitive, other wise four quads will be made up out of 8 triangles. When writing vertices to the vertex buffer, an arbitrary second point is chosen as with lines, and the other points on the rectangle are evaluated from this. (see Figure 4.4, Top Left, x and y). It was less complicated to generate the vertices on a thick rectangle than it was for lines, since rectangles are made up of only horizontal and vertical lines. Once the four corners of a thin rectangle were evaluated, the vertices of the triangles in Figure 4.4, Top Right, could be found by simply adding or subtracting the x or y vector (which had been multiplied by `thickness`). The same code was then used in the `::setPoint2` function to update all vertices based on a new second point provided. `CRectangle` also contains the function `::setPoints` which is similar to `::setPoint2` except that it takes 2 points and updates the whole rectangle. This was implemented in order to update the position of the cursor, which is made of a red rectangle and a black rectangle, and requires all corners to be updated based on the current gaze position.

As mentioned, `CEllipse` works similarly to `CRectangle` and `CLine` and encompasses the area described by an imaginary rectangle given by 2 points. (see Figure 4.4 Bottom Left) If `thickness` is `0.0f`, it is drawn with the line list primitive and vertices are evaluated using the equation of an ellipse, which is given below. They are evaluated at intervals of 64, which gave a good approximation of a smooth ellipse. If a thick ellipse is required, then points are first found along the equation of the ellipse as with the thin version. Next, the vector from the centre of the ellipse to the current point on the ellipse is normalized and multiplied by `thickness`, and then added and subtracted from the current point on the ellipse to give the vertices of a quad which would make up 1/64 of the thick ellipse. This can be seen in Figure 4.4 (Bottom Right) where the red lines show what 2 very thick quads would look like in wireframe rendering mode. It would have been much more effective to get the vector which was perpendicular to the ellipse at any one point to displace the vertices. The method chosen resulted in some ellipses that were not of uniform thickness if elongated ellipses were being drawn. However this only occurred in some cases and did not affect the experience of drawing significantly.

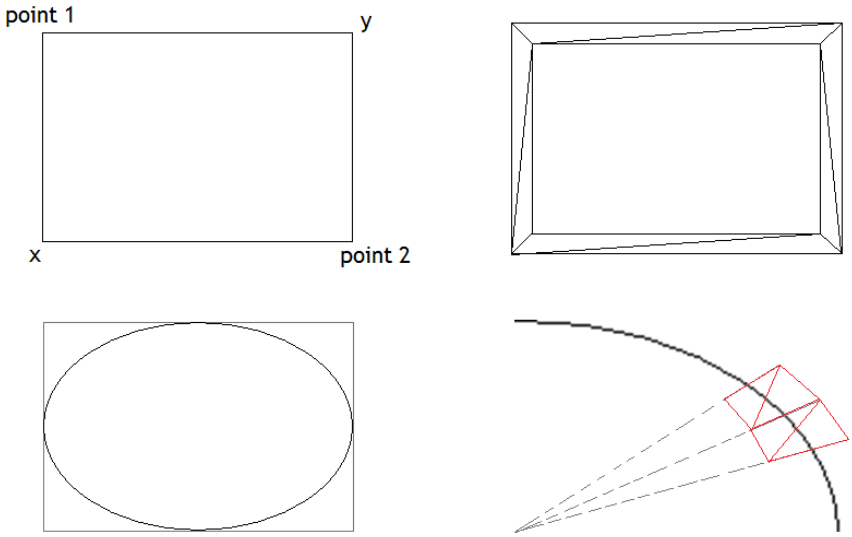


Figure 4.4: Top Left, thin rectangle. Top Right, thick rectangle. Bottom Left, thin ellipse. Bottom Right, constructing a thick ellipse.

4.4.3 Curve

A brief introduction to the curve tool was given in Section 3.4. It had been decided not to allow modification of the curve following its completion. In order to be able to visualize the curve while providing the control points, Catmull-Rom interpolation was used which made it possible to have the curve pass through the control points. If a bezier curve was used however, users would have to place control points some distance away from the path of the final curve and a lot of trial and error would have been necessary. The main drawback to using Catmull-Rom interpolation however was that by providing four control points, the resulting curve would only pass from the second point to the third point. This can be seen in Figure 4.5, Left.

Since it was desired to have the user supply 4 points, and have the curve start on the first point, pass through the second and third point, and finish on the fourth point, it was necessary to find two other imaginary points. These can be seen in Figure 4.5, Right, where the first imaginary point is calculated based on the angle a. The distance from point 1 to imaginary point 5 is the same as the distance from point 1 to point 2. A similar process is followed to get the new point 6. This is implemented in the `CCurve` class with the `::getExtraPoint` function. This function will be explained with

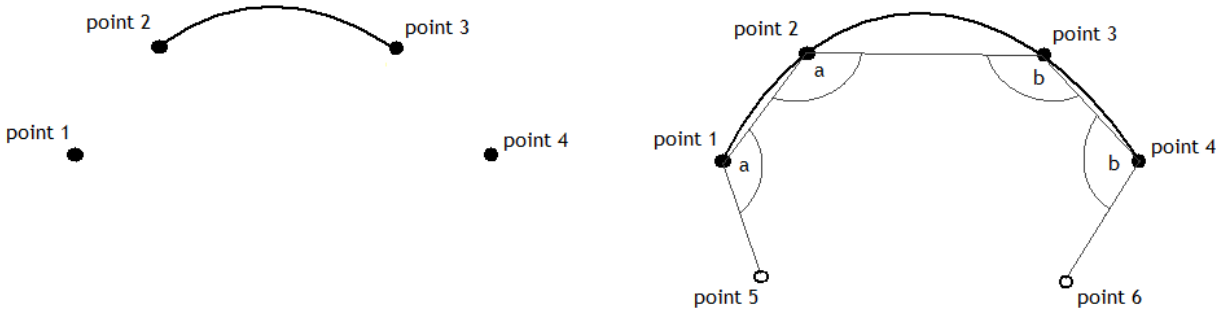


Figure 4.5: Left, Catmull-Rom interpolation given 4 points. Right, modified version.

regards to getting point 5 in Figure 4.5, Right. It starts by getting the vector from point 2 to point 1, and the vector from point 2 to point 3. The angle between these 2 vectors is found by first normalizing them and then getting the arc cosine of the dot product of them. A rotation matrix is then evaluated based on rotating around the Z axis by this angle, and finally the vector from point 1 to point 2 is multiplied by this matrix to find point 5. The function will work for point 6 by calling it with points 2, 3, and 4.

Once these extra points have been found, it is possible to generate three curves. In Figure 4.5 these can be seen going from point 1 to point 2, from point 2 to point 3, and from point 3 to point 4. The function `D3DXVec3CatmullRom` generates a point on a curve between 2 inner points when provided with 4 points and a parameter t, which controls how far along the curve the point should be. This function is called in 3 separate loops in order to generate points on all three curves. If `thickness` is greater than `0.0f`, it is necessary to displace vertices from the sides of the curve using the vector which is perpendicular to the curve at that point. This vector is normalized and multiplied by `thickness` in order to give the correct thickness. The curve can then be drawn by correctly indexing triangles which join up these vertices. Once the vertices of the curve have been provided, its vertex buffer is set and cannot be modified by the user.

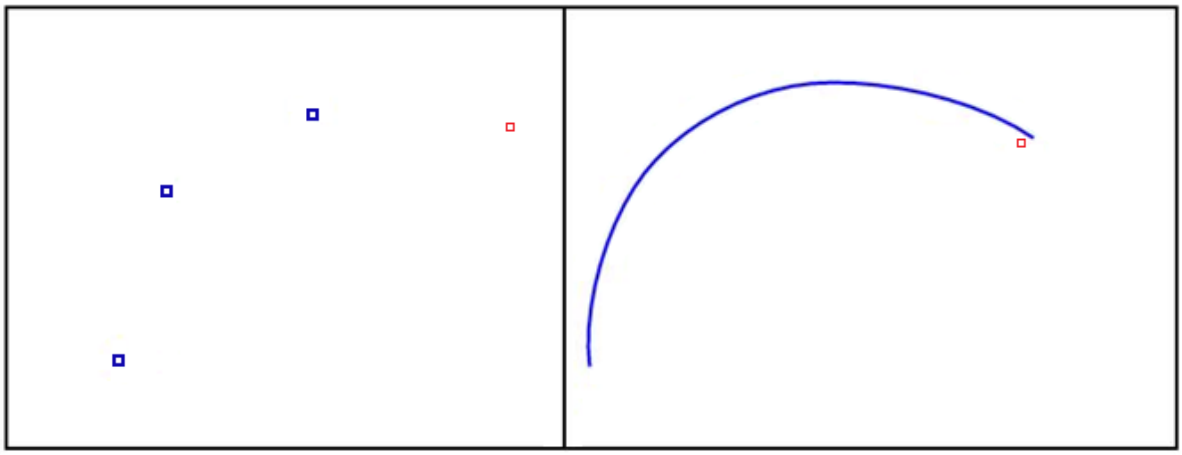


Figure 4.6: Left, first 3 points on curve. Right, finished curve.

4.4.4 Drawing Tools Handler

Up to this point, all shapes that are self contained have been discussed. Lines, rectangles, ellipses, and curves all have their own class, and own vertex buffer and are distinct objects. The remaining drawing tools that need to be discussed are poly-line and flood-fill, which deserve individual discussion due to them being implemented in a different manner to the previous tools discussed. In order to discuss poly-line, the drawing tools handler will be first introduced.

This is implemented by the class `cTools`. One instance of this class is created for the main application and it was designed to keep track of all shapes that need to be drawn, draw them, and interface with commands to add shapes. It contains several vectors (in this case referring to the C++ container `std::vector`) of pointers to different shapes, for a example a vector of pointers to the `CLine` class.

When a user says 'start' and the current drawing tool is set to line, a new line is added by pushing back a new pointer to a `CLine` to this vector, and initializing what it points to with the start position of the line, the current colour, and current thickness. This occurs within the `::addLine` function of `cTools`. Rectangles and ellipses are added with similar functions. `cTools` interfaces with the main application with the `::addShape` function which calls `::addLine`, `::addRectangle` or

`::addEllipse` depending on what the current drawing mode is. `::addShape` is called in `DrawingTools::ExecuteSpeechCommand` if the words 'start' or 'snap' are recognized. Since curves are only initialized when the user has specified four points, these need to be gathered prior to calling `::init` on the curve. `cTools` has a vector of points which represent these 4 points. If the user says 'start' or 'snap' and the current drawing tool is set to curve, then `::addCurvePoint` is called. The current gaze point is added to this list, and a rectangle is also added and modified to look like a small square at this point. These rectangles can be seen in Figure 4.6, Left, where the cursor is the small red rectangle. They provide feedback on where the user has chosen to place a curve point. Every time `::addCurvePoint` is called, it checks to see whether a fourth point has been added, and if so it removes the rectangles from the list to be drawn, calls `::addCurve` with the vector of current curve points, and then empties this vector. `::addCurve` simply adds a new curve to be drawn and initializes it with the 4 control points that it should pass through, a curve will then be drawn on screen similar to Figure 4.6, Right.

Poly-line is just a series of lines, and is controlled by `cTools`. Poly-line works by adding a new line to the system every time a user says 'start' or 'snap'. If they are in the middle of considering the end point of one segment of the poly-line and say 'start', the current segment will stop at the current gaze point and a new segment will start at this point. This is implemented in the function `::addLineForPoly` which works similarly to `::addLine`.

4.4.5 Flood-Fill

This tool was definitely the most difficult to implement due to the fact that DirectX was being used for the graphics. At the moment, shapes were being drawn in 3D space, on a virtual 2D canvas. When pixel coordinates were given to draw a shape, they were converted to points in 3D space and shapes were drawn based on these. However, in order to implement flood-fill, it was necessary to access the colour of virtually any pixel on screen. DirectX contains a function titled `GetFrontBufferData` which allows access to this data, but it is very slow and should not be used if good performance is necessary [15]. The solution to this was to render what is on screen to a texture and then access the colour of individual texels. This is done by the `DrawableTex2D` class

which was used from [11], and encapsulates what is necessary to initialize everything for rendering to texture in DirectX, and starting and stopping rendering. Rendering to a render target is done in the `::drawScene` function of the main application, where everything is drawn first to the render target, and then drawn a second time to screen.

The process is further complicated by the fact that render target textures are created in the default memory pool, and cannot be altered. A second texture is created when the application starts with the same dimensions as the render target texture. It is created in the system memory pool which allows for its data to be altered. Flood-fill is implemented in the `::copyScreenToFreshTexture` function of `cTools`. If the current drawing tool is set to flood-fill, and the user says 'start' this function begins by getting the top surfaces of both textures, and copying the data from the render target's surface to the other with the function `::GetRenderTargetData`. At this point the top surface of the texture which was created in system memory can be altered, which is done by calling the surfaces `::LockRect` function, which in this case locks the whole texture for editing, and returns a void pointer which points to the surfaces data. This pointer is cast to a 32 bit pointer in order to work with it and points to the first element of the top surface of a texture containing data pertaining to what is displayed on screen. It can be thought of as the start of a one dimensional array where each row of the texture is placed side by side. To access an arbitrary element of this array one would access it by calling: `array[(surfaceWidth * y) + x]` where x and y are the coordinates of the surface one needs to access. The data in this array is converted to a 2 dimensional array of colours, where it is now convenient to perform a flood-fill on it.

Various different flood-fill algorithms were investigated, and C code for a scanline recursive algorithm was found [16] which was robust and gave good performance when tested. It works by scanning to the left of the seed point testing each pixel to see if it is the same colour and setting to the new colour if it is, then scanning to the right and doing the same. The algorithm then recursively calls itself one pixel above and one pixel below the seed point and repeats for every horizontal line that is in this enclosed space. After this has been performed, there is a 2D array of colours which contains the image data, but which has been modified by the flood-fill algorithm. This 2D array is then copied back to the top surface of the system memory texture which now contains what was rendered from screen and then modified by the flood-fill. In order to be displayed on screen again, this texture needs to be copied to another texture which is

created in the default memory pool (similarly to the render target texture) in order to be able to be read by the GPU. At first this was implemented with the function `::UpdateSurface` which copies a surface from system memory to default memory. The texture could be saved to an image file correctly which showed that the flood-fill worked, but kept showing up as completely black on screen. A lot of time was spent on this problem and it is thought that this was because the GPU was not able to read the texture even though the correct function was being called to copy from system memory.

In the end, using another function `::D3DXLoadSurfaceFromSurface` solved the issue, and it is thought that this was because this function is a software wrapper around the lower-level Direct3D calls. It does a lot of checking and is not as direct as calling `::UpdateSurface`. Since it was only being called whenever a flood-fill was performed and not every frame, it was more than ideal due to its robustness. Once this problem had been solved, it was possible to render the altered texture to a screen aligned quad, which displayed pixel for pixel, exactly what had been rendered FROM the screen, with the addition of the flood-fill operation. Individual shapes were then still drawn in front of this texture. This introduced other issues, which will be discussed in the next section.

4.4.6 Undo

In order to implement the undo function, some research was done online and it was found that in many cases, paint programs use a stack of drawing operations which are cycled through when drawing. If a user wishes to perform an undo, the most recent drawing operation is simply popped off the top of the stack. To implement this, the draw function of `cTools` was modified. It had been looping through every single shape that had been added. A structure titled `drawingOperation` was written to accommodate this new method of drawing. It keeps track of both the drawing mode at the time of adding a shape, and an index variable that references which line, rectangle, ellipse, or curve should be drawn. With this change in place, a stack of these `drawingOperation`s are added to every time a shape is added. If the user chooses to draw a line and it happens to be the fifth line drawn of the picture, a `drawingOperation` is created with 4 as the index and 1 as the drawing mode. (1 indicates that the drawing

mode is a line.) Now in `cTools::draw`, it loops through all `drawingOperation`s. If the mode is 1, it draws the line indexed by 4, which is the fifth line drawn so far. If the mode was 2, it would draw the fifth rectangle, and so on. If a user says undo, the most recent drawing operation is simply popped off the stack, and that object is not drawn. Note that performing an undo does not delete or remove the pointer to that shape from the vector of shapes, it stays allocated until the application quits and all pointers are deleted. It would not have been difficult to also delete the pointer to the shape itself when popping the most recent drawing operation from the stack, but since vectors are capable of keeping track of thousands of items, this was not deemed necessary.

This worked fine for adding shapes in a way that facilitated easy removal, but it posed a problem with flood-fill. Due to the time taken to copy the data from the texture into a buffer, alter it and copy it back to be drawn on screen, it was not possible to have a drawing operation for flood-fill that would be repeated every draw-call. It was decided to take note of the coordinates that a flood-fill had been performed at, along with the original colour. If the user needed to undo a flood-fill operation, another flood-fill would be performed at that position with the original colour. When testing undo with flood-fill, another fundamental problem was found with integrating these two operations. By rendering the scene to a texture for a flood-fill, any shapes that were on the screen at that time, would now be part of the texture being modified for a flood-fill. At this point shapes were being drawn both on top of the texture individually, and as part of the texture which was being rendered to the screen first. Saying 'undo' would remove the `drawingOperation` required to draw the shape in front, but once they were part of the texture it was not possible to remove them. This could be fixed by only drawing shapes to the screen, and not the render target, but then they were not taken into account when a flood-fill was being performed. When it was realised that a solution could not be found with the current system of performing undo, it was decided to implement this piece of functionality in a radically different manner.

The system of having a stack of `drawingOperation`s which were cycled through to draw was kept. However after each shape was drawn, the screen was rendered to a new texture and the stack of `drawingOperation`s was emptied. So if the user draws 3 shapes, three textures would be kept in memory. The first would show only the first shape, the second would show the first 2, and the third would show all three shapes.

The application always displays the most recent texture to the screen, and undo can be performed by simply removing the most recent texture added. The amount of textures to keep track of was limited to 20 in order to limit the amount of memory being taken up by textures. This was thought of as more than enough, considering Microsoft Paint only allows 10 undo operations. It was a radical change to the application, and resulted in shapes only being drawn by their own vertex buffers very briefly, before being rendered to a texture. However this change was necessary to provide a robust solution to the task of integrating undo with flood-fill.

4.4.7 Helper Functions

As described in Section 3.4, it was intended to implement some helper functions to make drawing with gaze easier. The primary method of controlling paint programs is by using a mouse which provides accuracy to the pixel level. By adopting the method for smoothing discussed in Section 4.3.2, much of the inherent jitter was removed, and the cursor was made very stable. However, the cursor would still move somewhat, and there were two situations where gaze alone would not fulfill the accuracy required. The first is if a user wished to start or stop drawing a shape at exactly the corner of a shape, or end of a line. Also if they wished to draw lines that were exactly vertical or horizontal, it was difficult to do with gaze. In fact, these are somewhat difficult to do perfectly with a mouse in Microsoft Paint. By adding the facility to snap to nearest vertex, and also fix lines to be either vertical or horizontal, it was intended that the quality of pictures made possible with this application would increase substantially.

Snap was implemented first and allows the user to say 'snap' instead of 'start' to snap the current gaze point to the nearest vertex when starting a shape. If the distance to the closest point is greater than a certain threshold, then nothing happens and the shape starts as normal. It works by maintaining a vector of points, which represent ends of lines or corners. When a line is started, its first point is added to this vector, and when the line is stopped, so is the second point. This also happens for each line of a poly-line. When a rectangle is added, its four corners are added to this vector, and when a curve is added, so are its first and last point. `cTools::snapToPoint` was written to facilitate snapping to the nearest vertex. It takes a `D3DXVECTOR3` representing the current gaze point and returns a new point when finished. This new point is initialized

to be equal to the current gaze point. It first tests to see whether this is close enough to any vertex to snap to. It does this by looping through the vector of snap points, calculating the distance in pixels from the current gaze point to the current point in the list and testing whether this is less than a threshold. If it is, the function then tests whether the distance is less than the shortest distance found so far and if so, sets the shortest distance found to the current distance, and the point to return is set to the current point in the vector of snap points. After looping through all snap points, the new point will have been either assigned the value of one of the snap points, or if it was not close enough to any of them (or if no shapes with corners or ends of lines have been drawn so far) it is not overwritten. In this case it holds the value of the current gaze point which was supplied to the function.

When a user says 'undo', not only does the most recent shape need to be removed from the picture, but also its snap points from the list. This is done by checking the most recent `drawingOperation` for information on what drawing mode was selected when the shape was added. Then it is just a case of removing the correct number of snap points from the end of the vector. For example, if the most recent shape drawn was a line, then two snap points are removed. Once these snap points have been removed, the drawing operation itself is removed, before removing the most recent texture to display on screen and the undo is complete. This function is also called before stopping a shape if the user says 'leave' and wishes to stop drawing a shape at the nearest vertex.

`cTools::getfixPoint` was written to fix lines to be either vertical or horizontal and if the user turns this feature on by saying 'fix', the function will be called while a line is being drawn and its second point being modified. It starts by evaluating the angle between the current line being drawn and the X axis. If this angle is either less than 45 degrees, or it is in the range between 135 and 225 degrees, the line is more horizontal than vertical. In this case the Y value of the point returned is set to the same Y value of the first point, rendering the line horizontal. The reverse applies to lines closer to being vertical. Saying 'unfix' turns this feature off, and it is also turned off automatically when a line is finished. This meant that if the user wished to draw a series of lines in this manner, they would have to say 'fix' after saying 'start' each time to turn it on. This might seem a little awkward, but it was found to be more frustrating if it was left on and forgotten about.

4.4.8 Menu System

As mentioned in Section 3.5.4, the Menu System is made up of 3 classes, `Button`, `MenuScreen`, and `MenuSystem`. The purpose of the **Button** class is to first and foremost link to other menus and/or perform some function, and can take the address in memory of another `MenuScreen` or a global function on creation. If a button does not need to link to either, it is passed NULL for this parameter. Its other purpose is to display graphically on screen the purpose of that button. To facilitate this, the **Button** constructor is passed information pertaining to the area of an image file it should sample and display on screen. The constructor is also passed coordinates describing where it should appear on screen. A `MenuScreen` refers to a collection of buttons which make up a menu and serve a similar purpose. Figure 4.7 (top Left) shows the tools menu, where a user can change the current drawing tool by hovering the cursor in the area of a button and saying 'select'. To create the menu screen in Figure 4.7, six buttons were created which were passed the global function `modeChange` and NULL for `MenuScreen` address (since clicking on these buttons only required changing the current drawing tool, and not navigation to another menu screen). These buttons were then passed to the constructor of an instance of `MenuScreen`, along with the filename of an image which contained pictures of each drawing tool. Each button had been passed details of where in this image file to sample for image data. The `MenuSystem` class is responsible for interfacing with the different menu screens and drawing the menu screen which is currently active. If a user says 'select', the menu system checks to see if a menu screen is active and being displayed (only one will ever be active at a time) and then calls the `::getRect` function of this screen with the position of the cursor. This function loops through all its buttons and tests each one to see if the coordinates of the cursor lie within the area of any of its buttons. If so, it then tests whether it contains the address of another menu screen. If it does, it deactivates the current menu screen and activates the menu screen that it holds the address to. It also tests whether this button contains the address of a function. It calls the function and deactivates the current menu screen if it does. So in Figure 4.7, Top Left, when a user says 'select' while hovering the cursor above the rectangle button, `modeChange` is called with the index of the current button which is 2. This number indicates that the mode should be changed to rectangle. Examples of other functions include `colourChange`,`thicknessChange` and `savePicture`.

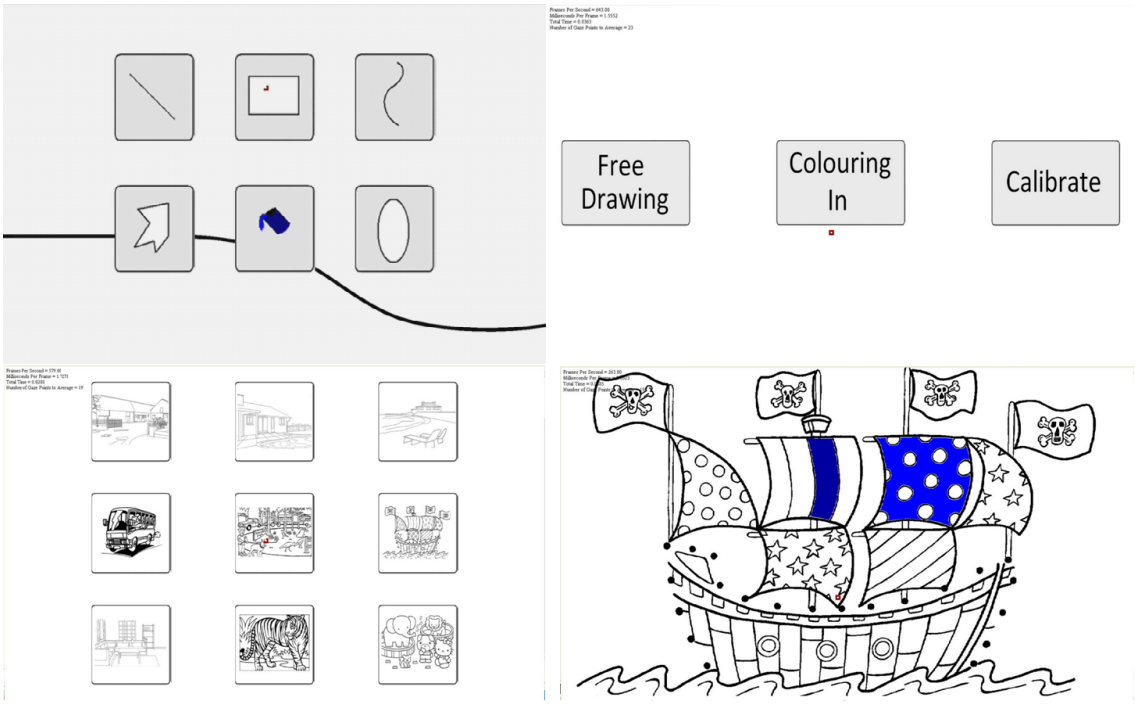


Figure 4.7: Top Left, Tools Menu. Top Right, Start-Up Menu. Bottom Left, Pictures Menu. Bottom Right, Colouring In Mode.

With these classes implemented, a whole system of interconnected menus could be set up with some screens linking to others. The `DrawingTools::ExecuteCommand` function is where voice commands are interpreted, and it is here that the menu system is interacted with when a user wishes to activate a certain menu (saying 'Open Tools' opens the drawing tools menu), select a menu button, or close a menu (by saying 'back').

4.5 Colouring In Mode

As described in Section 3.4, it was also intended to have a mode for colouring in line drawings by using gaze and voice with the intention that this might be more fun for younger users. In this mode, the only drawing tool that is available to users is the flood-fill, and the only menu that is available is the colours menu and a menu for saving and quitting. When the application starts, users have a choice to either choose free drawing mode, colouring in mode, or to calibrate. (See Figure 4.7, Top Right) If they choose colouring in, they are taken to a menu which contains pictures to colour in. (Figure

4.7, Bottom Left) Following this, a texture which corresponds to the same image as the button chosen is set as the default texture to show on screen, and users can use the default drawing tool in this mode, flood-fill, to colour it in. (Figure 4.7, Bottom Right) When they are finished, users can say 'Open File' which opens a menu screen with 2 buttons, save and quit. Choosing save outputs the texture being displayed on screen to a bitmap file, and choosing quit shuts down the application. This mode was one of the last features to be added to the application, and by this time everything was already in place to allow straightforward integration. The program had already been modified by this point to constantly be displaying a texture on screen (as discussed in Sections 4.4.5 and 4.4.6) so it was trivial to set this texture to one that contained a picture. Since flood-fill worked by rendering the screen to a render-target, it was just a case of taking a snap-shot of everything that was on-screen at one time and performing a flood-fill. A flexible menu system had been implemented also, so it was straightforward to create and link up the extra menu screens (Figure 4.7, Top Right, Bottom Left). Due to this framework being in place prior to this, implementing the colouring in mode did not take long.

4.6 Data Output

It was decided that the application should output data to a textfile which would provide information on which actions a participant took, and when they took them. This would be useful in analyzing their behavior and for gathering data on issues with the program. For instance if they found one drawing tool particularly difficult to use they might need to give the command several times, each time followed by an 'undo' operation in order to complete it successfully. This could not be interpreted from the final picture. The feature works by outputting a line of text every 100 milliseconds. This line of text contains the time in seconds since the application started, the position of the gaze point in screen coordinates, and the current menu that is activated. If no menu is currently activated, then the number 0 would indicate this. When a user performs any action this information is logged as can be seen with the example of a rectangle:

```
Time: 94.8506 gazepoint: {521, 242} menu state: 0
Time: 94.9513 gazepoint: {521, 241} menu state: 0
```

```
Time: 95.052 gazepoint: {521, 240} menu state: 0
Time: 95.1367 user ADDED rectangle at {521, 240} with
     colour {0,0,0} & thickness 0.04
Time: 95.1532 gazepoint: {521, 240} menu state: 0
Time: 95.254 gazepoint: {524, 240} menu state: 0
Time: 95.3547 gazepoint: {531, 246} menu state: 0
Time: 95.4556 gazepoint: {535, 250} menu state: 0
....
Time: 135.407 gazepoint: {915, 446} menu state: 0
Time: 135.507 gazepoint: {869, 409} menu state: 0
Time: 135.527 user STOPPED rectangle at {851, 394}
Time: 135.607 gazepoint: {828, 374} menu state: 0
Time: 135.707 gazepoint: {804, 357} menu state: 0
```

Chapter 5

Evaluation

This chapter discusses the user evaluations which were necessary to evaluate the application. Section 5.1 explains the design of the user evaluation while Section 5.2 presents the results of the trials. Finally Section 5.3 gives an appraisal of the results, followed by the printed pictures that were drawn by the participants.

5.1 Design of the User Evaluations

Once the application had been finished it was necessary to evaluate it and assess its usability. It was decided to recruit two different groups of participants to test the application. The first group would be made up of experienced programmers, and the second group would be recruited from outside the computer science department of Trinity College, and have no experience with computer programming. It was expected that the first group would have substantially more experience with paint programs such as Adobe Photoshop etc. and this would be ascertained during the evaluation process. The main aim of these participant trials was to:

1. assess the difficulty in using gaze and voice as input for a paint program when compared to mouse and keyboard.

2. assess whether either group found it easier.

3. assess whether people thought it could be of benefit to disabled users or not.

It was decided not to have a clear-cut compare and contrast situation between gaze/voice and mouse/keyboard. This was because firstly it was expected that most people have used some sort of paint program in their lives and also it was desirable to keep the time taken for each trial to about 20 minutes, so time needed to be spent on acclimatizing oneself to the application before use. Since comparing the two input methods would be done on the basis of participant's history of experience with gaze and voice, this would need to be done by means of a questionnaire. It was intended that this would answer the questions above, and also provide other helpful data. This was a subjective form of analysis, but was the best option since comparison with keyboard and mouse was being done on the basis of participants prior experience. With this decided upon, the process of the trial was loosely set out. This took the form of asking the participants to experiment with the application and try out each drawing tool, followed by completing a drawing task within a certain time limit. It was decided not to have participants test out the colouring in mode. This was partly due to the fact that it would have made the overall trial time too long. Also, since this mode is based on just the flood-fill tool being used to colour in line-drawings, participants experience with this tool in the free-drawing mode could give an impression of how well the colouring in mode might work.

5.1.1 Participants

Participants were selected from volunteers, both from the Interactive Entertainment Technology (IET) MSc. course at Trinity College and from outside the college. There were eleven people recruited for each group, which allowed for one person per group not being able to complete the study. For the first group of experienced programmers, one individual ran into considerable difficulty with the voice recognition due to his foreign accent. The program had been trained with just the authors voice which worked in most cases, but it was decided to exclude this person due to the fact that it could not recognize the word 'start' at all, which meant that he could not use the application in a satisfactory manner. For the other group, there was one participant who was significantly older than the rest of the group, and had great difficulty maintaining the calibration, with the result that this participant also could not use the application satisfactorily. Due to the fact that this participant was of a sufficiently different demo-

graphic than the others and that it was thought that this had affected the results of the eye-tracker, this individual was also excluded. In the end, results for ten participants from each group were collected.

5.1.2 Questionnaires

It was decided to keep the questionnaires relatively short and to have corresponding questions for both using gaze and voice and mouse and keyboard. By collecting participants opinions on similar aspects of these two input methods, simple statistical tests could be run on the results in order for conclusions to be drawn. As can be seen in Appendix A, the bulk of the questionnaire was taken up with questions rating both input methods under the following headings; Ease of navigating the menus, how much control participants felt they had, how fast they drew, precision of controls, enjoyment, and how natural the controls were. Each question asked participants to rank an aspect of either input method on a scale from 1 to 7, with 1 being most negative and 7 being most positive. Participants were also asked to rate the ease of giving voice commands, though this could not directly be compared to mouse and keyboard.

The author wears glasses with quite a strong prescription and this had proved to offset the result of the eye tracker considerably during implementation, usually by about a couple of inches away from where the actual point of regard was thought to be. Because of this, participants were recruited on the basis of having normal vision in order to avoid running into similar issues with calibration. Participants were first given an information sheet which gave some details on the experiment and how it would be carried out. They were also given a consent form to sign. After signing the consent form, they filled out the first page of the questionnaire as shown in Appendix A which collected data on their age, gender, and experience with paint programs. This page also asked if participants had any history of epilepsy. If a participant answered yes, they were to be excluded from the experiment immediately.

5.1.3 Setup and Procedure

It was decided to have the experiment take place in the computer lab that was used for implementing the project. The experiment would be more controlled if it took place in a sound proof room, but this does not give a realistic environment for everyday use.

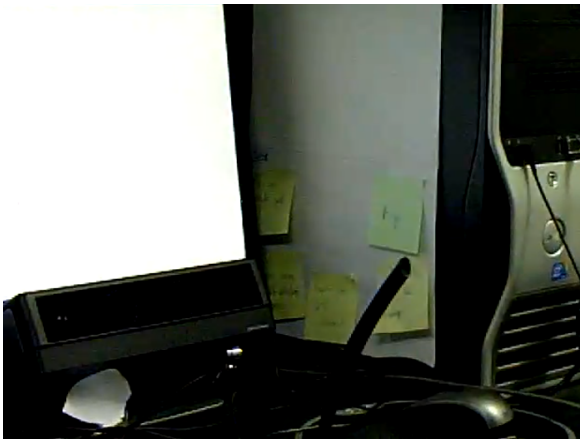


Figure 5.1: Eye-tracker position.

By using a room with some background noise, a feel for how well the voice recognition performed in everyday conditions could be achieved. This computer lab was also in use by other people so during the course of the experiments, they were asked to stay as quiet as possible. This meant that any background noise was as a result of regular noises from outside (for example, there was some noise from trains going by quite regularly) and that each participant had a similar amount of background noise.

The Tobii X-120 portable eye-tracker was positioned below a widescreen monitor (see Figure 5.1), along with a USB microphone which was placed in front of the keyboard. A headset microphone may have provided better response for the voice recognition but this was deemed to be somewhat intrusive to wear and was decided against. Once participants had signed the consent form and filled out the first page of the questionnaire it was safe to proceed, none of the participants answered yes to having epilepsy or a history of epilepsy in their family. In order to keep each experiment as similar as possible, it was decided to hand each participant an instruction leaflet to read at this point (See Appendix B) which explained how to use the drawing tools and helper functions. They were than asked to sit comfortable so that their eyes were reflected back at themselves in the front panel of the eye-tracker (which ensured that they were not sitting too high or too low) and were told they could adjust the seat height if necessary. The distance from their eyes to the eye tracker was measured using a measuring tape to ensure that this was in the range of 60cm to 70cm. They were then asked to calibrate the eye tracker prior to starting the experiment. This

was done after participants had read the instructions since it was desirable to conduct calibration immediately before starting drawing. This was so that there would be as little chance as possible of participants moving position following calibration. Looking down to read instructions would also have increased the chance of them moving position. Calibrating the eye-tracker only took a couple of moments so it was thought that the instructions would still be fresh in peoples minds.

Once calibration was completed participants were asked to start the free drawing mode and to test out each drawing tool at least once. They were told that they could ask questions at any time if there was something they did not understand. Once they felt they were ready, the application was reset to a blank canvas, and participants were given a picture of a house to draw. They were told it did not have to be exactly the same, but to draw it as best they could and that they had a time limit of ten minutes. Once they were ready to start, a key was pressed on the keyboard which started a timer in the application. The length of time in seconds from this moment was kept track of in the main update loop of the application. If it exceed ten minutes, the application saved the picture to an image file and automatically closed down. The whole experiment took about 20 minutes per participant. Once the application had terminated, participants were handed back the questionnaire to complete.

5.2 Results

This section discusses the results obtained from the questionnaires completed by participants. These results can be split into three broad categories; rankings obtained, participant comments, and favourite/most difficult drawing tools. The results alone will be presented before moving onto discussing what can be learned from them. One participant from each group failed to complete the section of the questionnaire pertaining to mouse and keyboard. This was participant 10 from group one and participant 6 from group two. These participants were not taken into account when performing statistical tests and have been marked with an asterisk on any bar charts in the next section, where the value of zero for mouse and keyboard indicates that they did not complete this section. Data on individual participants is given in Appendix C. This also contains information on which paint programs participants had experience with. Since the mean amount of paint programs that participants in group one had expe-

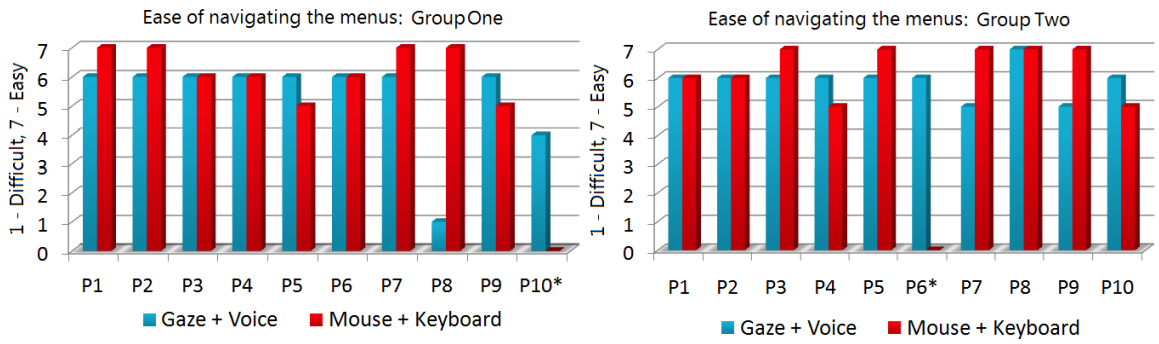


Figure 5.2: Ease of Navigating the Menus.

	Gaze/Voice	Mouse/Keyboard	P Range	Sig. Difference?
Group 1 Mean	5.44	6.22	$p > 0.2$	No
Group 2 Mean	5.89	6.33	$p > 0.2$	No

Table 5.1: Ease of navigating menus.

rience with is 3.5 and the mean for group two is 1.4, group one was deemed to have more experience with paint programs overall.

5.2.1 Rankings Obtained

Each question on the questionnaire was analyzed by a two tailed Wilcoxon Matched-Pairs Signed-Ranks test [24] to ascertain whether there was a significant difference between both methods of input. The first question 'Ease of Navigating the Menus' was found to have no significant difference between Gaze/Voice and Mouse/Keyboard. This is shown in Table 5.1 and Figure 5.2 where both input methods are ranked overall relatively highly by both groups. The P value must be below 0.05 to indicate a significant difference between the two input methods.

The second question 'How much control was there?' was found to have a significant difference with both groups, favouring mouse and keyboard. This is shown in Table 5.2 and Figure 5.3.

The third question 'How fast did you draw?' was also found to have a significant difference between groups favouring keyboard and mouse.

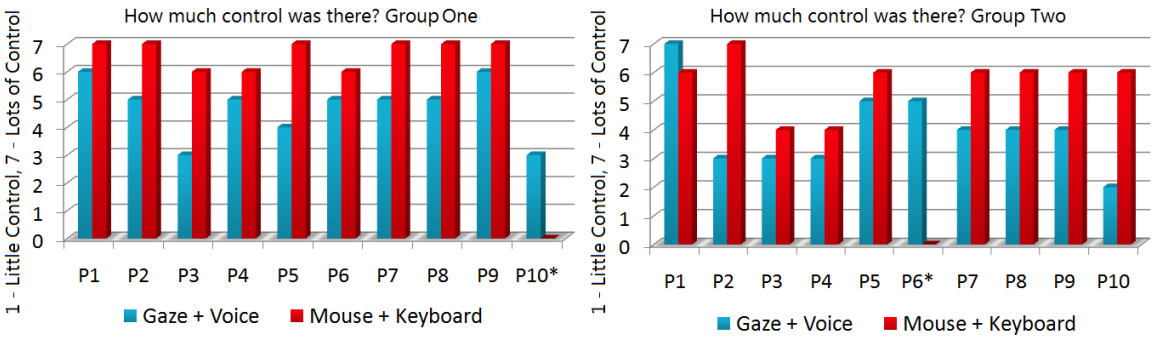


Figure 5.3: How much control was there?

	Gaze/Voice	Mouse/Keyboard	P Range	Sig. Difference?
Group 1 Mean	4.88	6.67	$0.01 < p < 0.02$	Yes
Group 2 Mean	3.89	5.67	$p < 0.001$	Yes

Table 5.2: How much control was there?

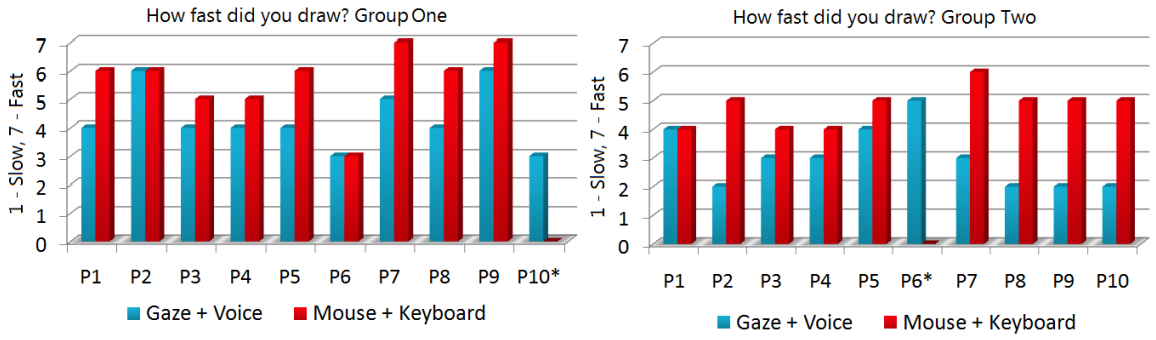


Figure 5.4: How fast did you draw?

	Gaze/Voice	Mouse/Keyboard	P Range	Sig. Difference?
Group 1 Mean	4.45	5.67	$p < 0.001$	Yes
Group 2 Mean	2.78	4.78	$p < 0.001$	Yes

Table 5.3: How fast did you draw?

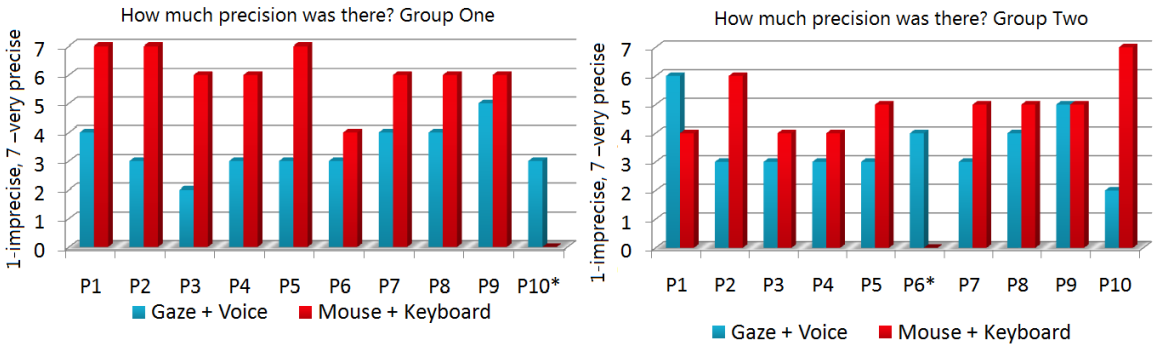


Figure 5.5: How much precision of the controls was there?

	Gaze/Voice	Mouse/Keyboard	P Range	Sig. Difference?
Group 1 Mean	3.44	6.11	$p < 0.001$	Yes
Group 2 Mean	3.56	5	$0.05 < p < 0.1$	No

Table 5.4: How much precision of the controls was there?

The results for the fourth question ('How much precision of the controls was there?') were interesting. The first group scored a significant difference favouring mouse and keyboard, while results for the second group returned no significant difference. (See Figure 5.5 and Table 5.4.)

The fifth question 'How enjoyable is it?' resulted in a significant difference in favour of using Gaze and Voice as input.

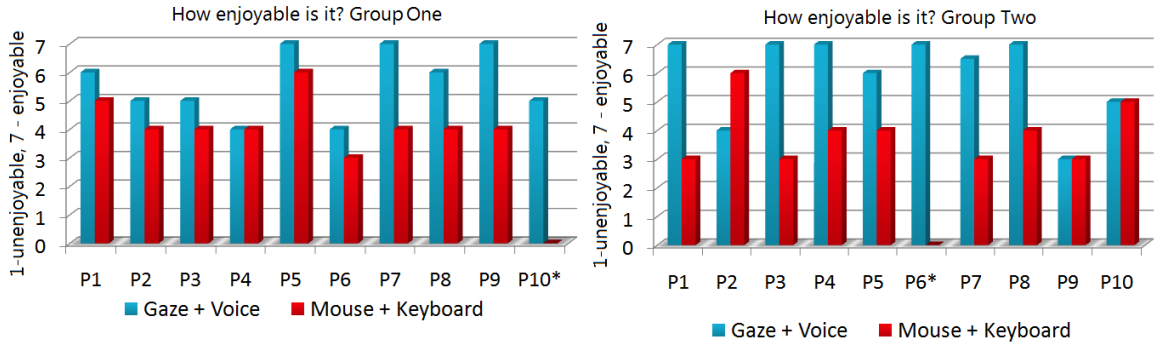


Figure 5.6: How enjoyable is it?

	Gaze/Voice	Mouse/Keyboard	P Range	Sig. Difference?
Group 1 Mean	5.67	4.22	$p < 0.001$	Yes
Group 2 Mean	5.83	3.89	$0.02 < p < 0.05$	Yes

Table 5.5: How enjoyable is it?

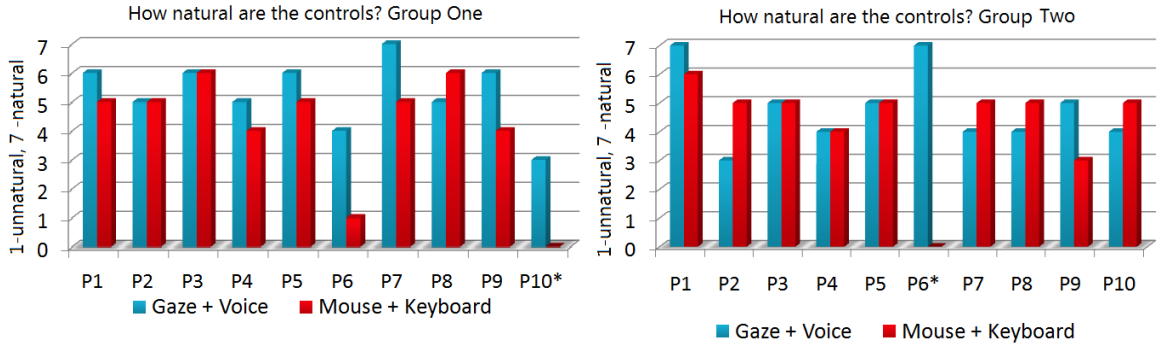


Figure 5.7: How natural are the controls?

The sixth question 'How natural are the controls?' did not return a significant difference, though in Figure 5.7 it can be seen that with the first group, 6 out of the 9 people being compared rated naturalness of the controls as higher for Gaze and Voice.

The questionnaire also asked participants to rate the ease of giving commands with voice on a scale of 1 to 7. Since this question was specific to using gaze as input and did not apply to mouse and keyboard, statistics were not run on these results. They can be seen in Figure 5.8 and result in a mean of 6.1 for group one and a mean of 6 for group two.

5.2.2 Participant Comments

In general the comments from participants were quite promising. Everybody replied that this application could benefit users who cannot use traditional forms of input.

	Gaze/Voice	Mouse/Keyboard	P Range	Sig. Difference?
Group 1 Mean	5.56	4.56	$0.05 < p < 0.1$	No
Group 2 Mean	4.56	4.78	$p > 0.2$	No

Table 5.6: How natural are the controls?

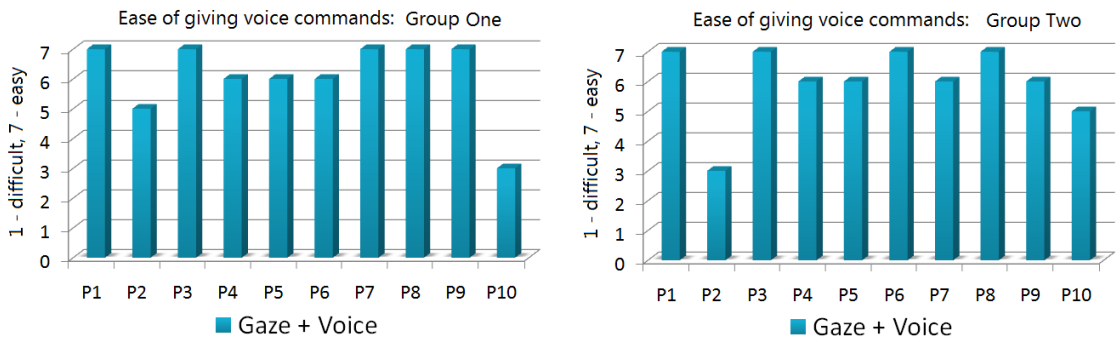


Figure 5.8: Ease of giving voice commands.

Some of the comments relating to this are:

- "The menus were easy to navigate with large icons making tool selection simple and while not as precise as typical tools it is certainly a viable alternative if the user is unable to utilize traditional tools"
- "Yes, because I can't think of an application with such intuitive alternative input devices"
- "I think with a lot of practice, it could be really beneficial to anyone who cannot use a mouse or keyboard, (and it's really fun)"
- "The combination of voice and eye control after getting used to it is very similar to mouse use. So for people not able to use a mouse it would be quite useful"
- "It could provide a much needed outlet for people with limited mobility, I'm surprised someone didn't think of it sooner!"
- "Very enjoyable and very interesting way to use computers for people with physical disabilities"

Also in the **any additional comments** section, 30% of respondents remarked that with some practice this would be easier to use with one participant remarking "I think that with some training performance using the application can improve greatly, even though higher position in the tracked eye position is necessary for fine control tasks (like drawing without 'snap' or 'fix')" and another, "Yes because with practice this

type of input could be as user friendly as a keyboard and mouse".

Several respondents felt frustrated with the precision offered with gaze, "The eye tracking was difficult to use to pick precise points on the screen, but was intuitive and immediate for menus", "Commands were straightforward to use and remember, but lack of precision in tracking eyes became somewhat frustrating", "As a tool though it is not precise enough to replace other peripherals like the mouse or tablet".

Some participants had suggestions for features that would make drawing with gaze easier; "Could be an idea to make cursor change colour to confirm menu option has been activated as I was not sure it had registered until my shape darted across the screen!" while another participant suggested "An aid for focusing, like a grid because it's difficult to focus on white space".

There were several other promising comments from group one; "It is easy and intuitive to use overall and is a satisfactory experience", "It's an alternative pointing device with an improved command input, as speech commands are faster and more natural than using mouse to open menus", "Navigation of menus are very natural and easy to understand. The use of "snap" and "leave" commands allow compensation for any lack of precision. Does not take long to adapt to the controls. Fun new interface".

5.2.3 Drawing Tools

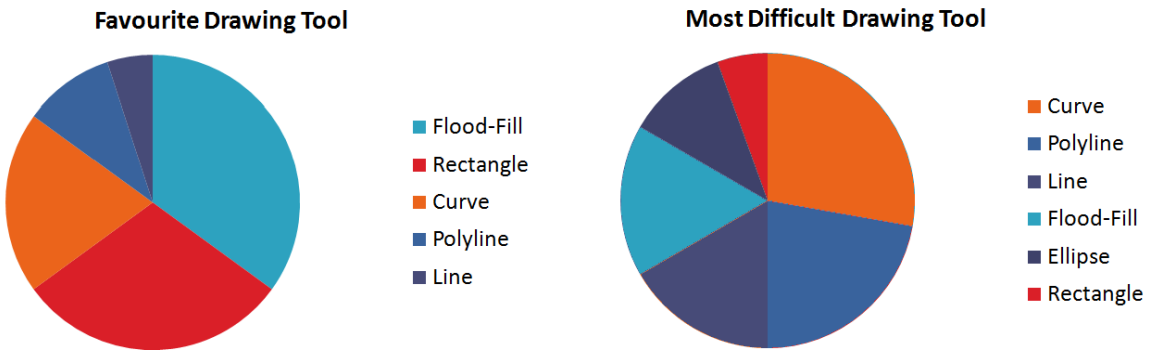


Figure 5.9: Favourite and Most Difficult Drawing Tools.

There were also questions relating to which drawing tools participants found to be their favourite and most difficult to use. These are shown in Figure 5.9 and have been collated for both groups together. The tool that most people found to be their favourite

was flood-fill, and this was expected since it only needed to be activated in one area, and by one voice command. It also offered great visual feedback by colouring in large portions of the picture with such little effort. The fact that this tool scored so highly is promising for how much participants might enjoy the colouring-in mode. Rectangle was the second favourite drawing tool which had also been expected to score highly. This is because it offered the ability to draw a shape that looked good quickly as it always had vertical and horizontal lines. Curve came third as a favourite tool which is strange considering it came first in the most difficult drawing tool category. This may be because some participants just "got it". Figures 5.10 and 5.11 show that some of the pictures have nice curves drawn, whereas others show some difficulty with this tool. Line and Poly-line had been expected to score higher than they did, occupying the last two places in the favourite tools category, and coming second and third in the most difficult tool. This may be because there were a lot of voice commands to use, and perhaps participants needed more practice in order to use the helper functions ('snap', 'leave', 'fix', 'unfix') effectively. Only one participant commented that these helped drawing. Ellipse did not feature in the favourite tool category but did in the most difficult category, though it did come fifth in this. Participants may have found this tool difficult to use since the shape itself did not lie on either of the two gaze points required to describe its position. (Recall from Section 4.4.2 an ellipse is described by the area of a rectangle which is defined by two gaze points.) As a result, participants may have been put off by this shape being further away from their POR than rectangle was for example.

5.3 Appraisal of Results

The rankings obtained for aspects of each input method were quite promising. The question relating to ease of use of the menus returned no significant difference between input methods. This is promising as it shows that participants felt that using the menu system in this application was close to being as easy as with a mouse or keyboard. It had been intended to have the menus as accessible as possible with large enough buttons for choosing with gaze. Perhaps it was felt to be more intuitive to look at a large icon with a picture on it than to use a mouse to select words on a menu, as is found in most programs. The next two questions, 'How much control was there?'

and 'How fast did you draw?' both returned a significant difference favouring mouse and keyboard, which indicates that participants felt that traditional programs using mouse and keyboard offer more control and faster drawing. This result was expected though, since gaze simply cannot compete with the sub-pixel accuracy of the mouse. The fourth question only returned a significant difference from group one, and favoured keyboard and mouse. It had been expected that this would have also been the result for the other group. It is thought that this is because group two had less experience with paint programs overall than group one, and therefore found less of a difference in precision between the two modes of input.

Both groups felt that using gaze and voice as methods of input was significantly more enjoyable than keyboard and mouse, which was a good result. There was no significant difference in how natural each group found each input method. This was also a good result as it indicated that this application is on par with using keyboard and mouse even though this was the first time that each participant had used gaze to control a cursor. Overall the participant comments were very positive. 100% of participants felt that it would be of benefit to disabled users, and many people replied that it was a fun experience, which backed up the statistical results for this aspect. Since 30% of participants remarked that with practice this would become much easier without being asked their opinion on this, it seems that there is definitely scope for this to be used as a viable input method for users who cannot use traditional input methods. The voice recognition worked very well also, though several female participants had difficulty with the application recognizing their voices. One participant commented: "Found it hard to Stop and undo, but if it recognized my voice better, than it would be brilliant! thanks." The overall participant response was very promising for the question of "Ease of giving voice commands" where there was a mean of 6.1 and 6 for groups one and two respectively. This is a high score and shows that there were not significant difficulties with ease of giving voice commands. It can be seen that using gaze and voice as input methods is significantly slower and offers less control than keyboard and mouse (and also less precision with group one). This is an expected result and as can be seen in Figure 5.10 and Figure 5.11, most participants were able to complete the drawing task satisfactorily. Considering each participant's experience with using gaze and voice consisted of spending roughly ten minutes testing out each drawing tool before starting, this is also quite promising. When considering the pictures returned

by both groups, and also the statistical results for each question, both groups are seen to have had a relatively similar level of difficulty with the program. This fits in with the idea that controlling a cursor on screen with gaze is a new skill which needs to be practiced if used regularly.

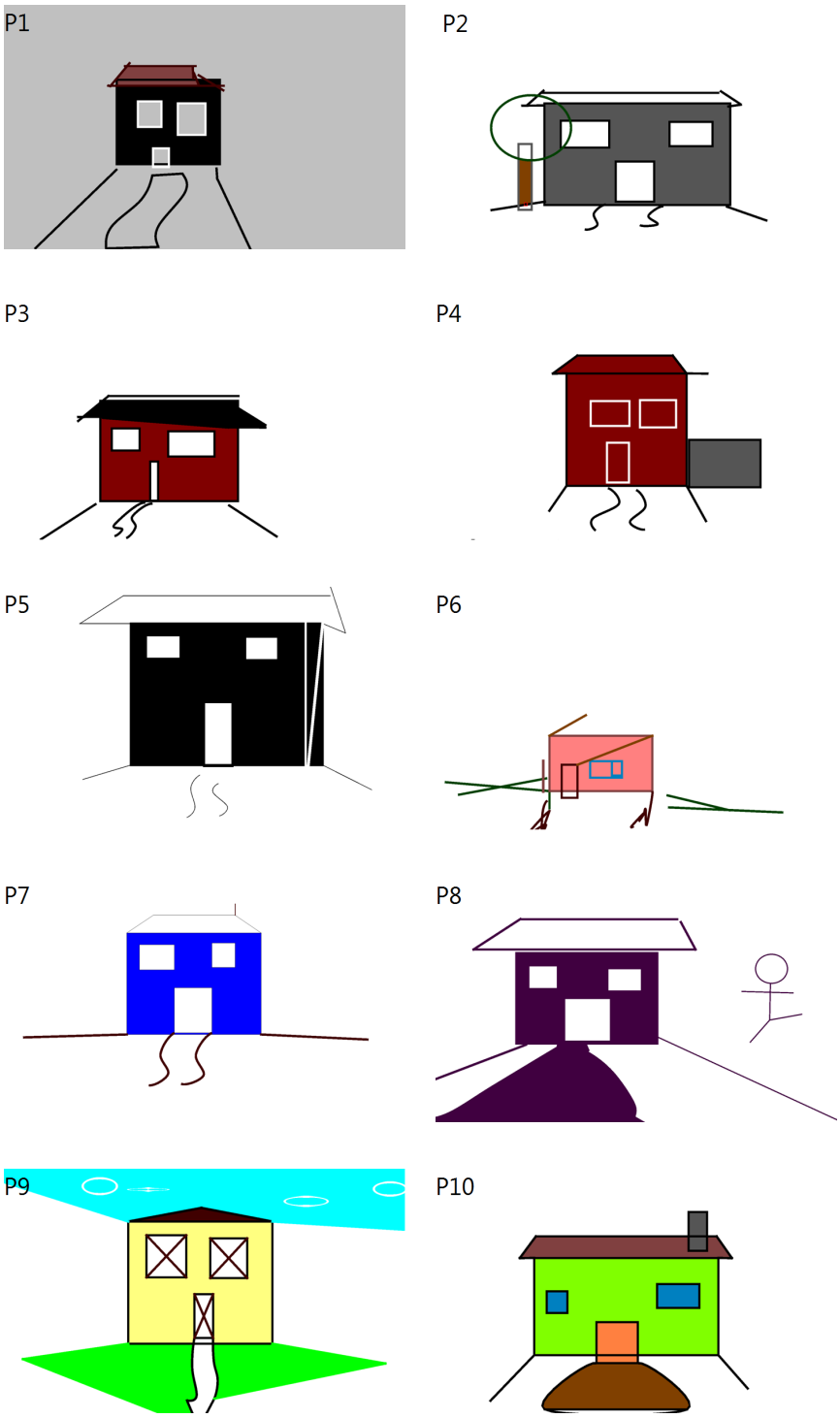


Figure 5.10: Group One Pictures.

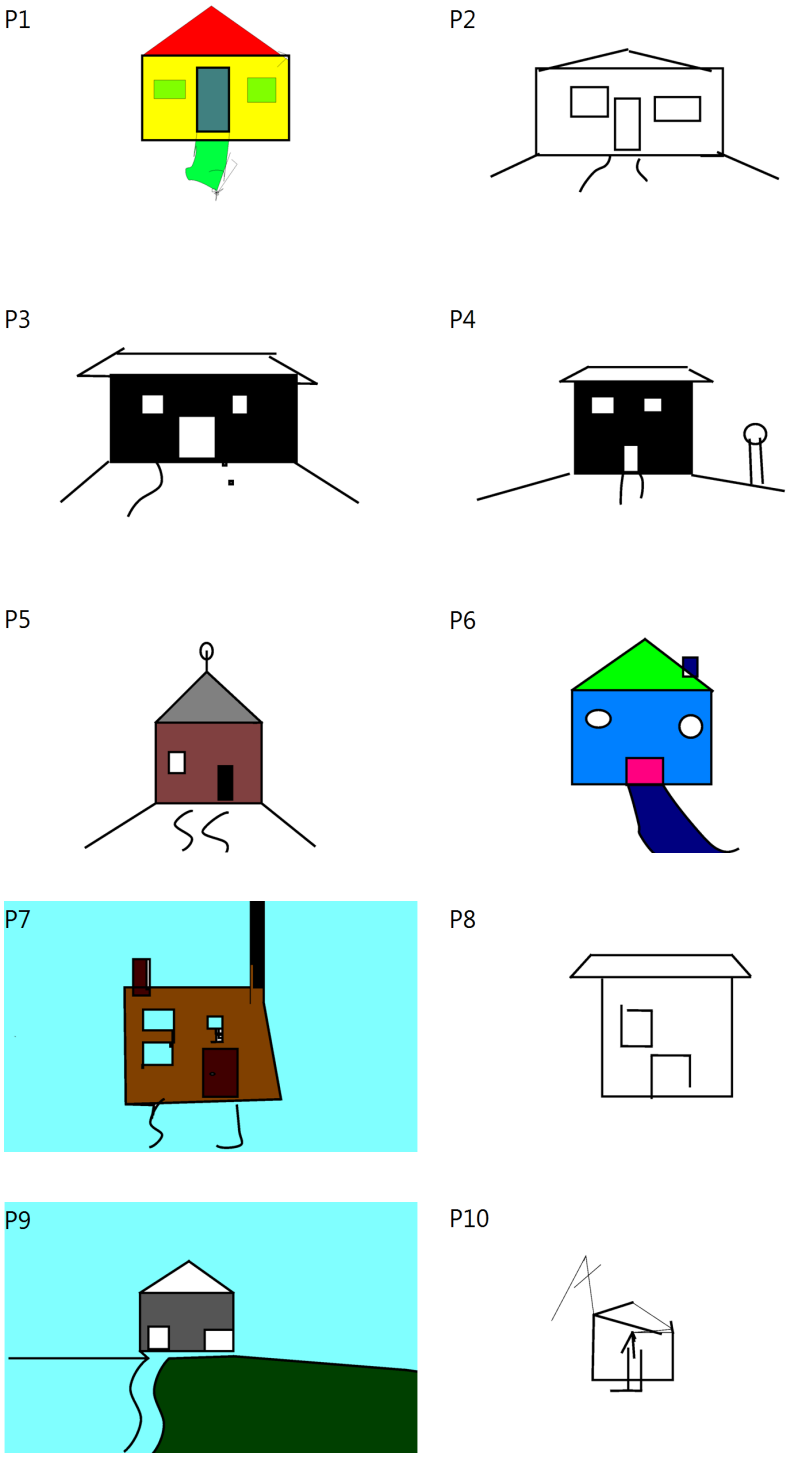


Figure 5.11: Group Two Pictures.

Chapter 6

Conclusions and Future Work

This chapter begins by discussing conclusions drawn from the project, before ending with details on possible future work with the application.

6.1 Conclusions

The main aim of this dissertation was to create a paint program which is controlled by gaze and voice. User trials took place in order to evaluate how successful this was. It was implemented in such a way that it can be changed around very easily. Since the menu system was made to be quite flexible, new arrangements of menu screens can be set up with minimal effort.

Implementation of the drawing system was not as straightforward as was hoped at the beginning, with the issues related to the flood-fill and undo operations taking up a large amount of development time. Perhaps if another graphics API had been used instead of DirectX, this would have been much more straightforward, but the fact that it was implemented with a 3D graphics API opens up many possibilities for future work. While a paint program primarily needs a 2D API, impressive 3D effects could be implemented with the power of DirectX such as having the picture on screen twist in space like a piece of paper flying away in the wind when the user shuts down. Also, there is scope for extending the program to modelling 3D objects. How effective this would be with gaze and voice is hard to say, but the possibility is there.

The project intended to improve on previous work in this area by implementing

a novel approach of using voice recognition along with gaze. The voice recognition helped in several ways. By using it to activate drawing, users of this application do not have to wait for a fixation to be picked up. This avoids the delay involved in using dwell time for activating drawing. Also the problem of accidentally activating drawing by fixating gaze at a point is removed. Using voice recognition also made it possible to have menus that were completely invisible when not in use and can be accessed without gaze. This removed the problem of having distracting icons along the side of the screen that were limited in size. These improvements were seen to be successful according to participants responses given to the "Ease of giving voice commands" and "Ease of use of menus" discussed in Section 5.2.1. The voice recognition worked well. There were some issues with female users and one user who had a very different accent to most others. This is not seen as a major problem since it is possible for end users to train the voice recognition engine themselves in the Control Panel of Windows. It had been decided not to do this for each participant due to the extra delay it would introduce for each trial. Drawing was made easier with gaze by implementing both a smoothing algorithm (see Section 4.3.2) and helper functions (see Section 4.4.7). The helper functions were not used by all participants, but it is thought that with more time and practice, participants would learn how to use them to their advantage to increase the quality of pictures produced with a gaze application. Every user who tried the application thought that it would would benefit disabled users which was promising.

6.2 Future Work

Since this application used a novel approach and is the first of its kind to use both gaze and voice in a paint program, development unearthed many possibilities for future work and improvements. Visual feedback is always very important, and the image of the cursor could change according to what stage of drawing the program was in. For instance it could be a cross while looking around the screen, change to a filled circle as soon as 'start' or 'snap' is uttered, and then change back to a cross once a 'stop' command had been processed. These three different states are shown in Figure 6.1. This would take away ambiguity of the exact time that a command had been processed, and avoid users looking away from the desired second point of a shape as soon as they said 'stop'. This feature was suggested by a user from group two (see Section 5.2.2)

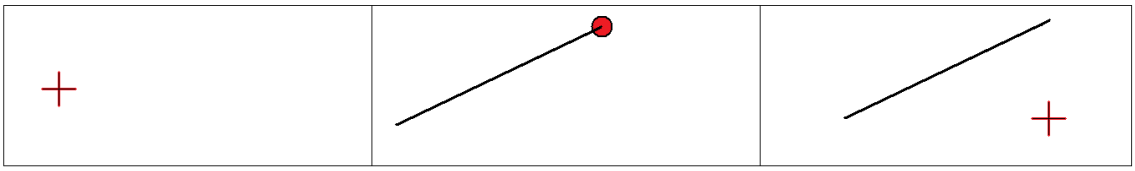


Figure 6.1: Left, not drawing. Middle, drawing a line. Right, finished drawing.

and would be extremely trivial to implement. Some users had trouble concentrating on pure white space and suggested a series of grid points which would help with positioning shapes. This could be implemented with a repeated pattern of different colours as seen in Figure 6.2 and might help in concentrating gaze at a particular point. Naturally there should be an option to turn this grid on or off easily as needed.

Another potential addition to the program would be to have the colouring-in mode integrated into the main application, as it is accessed by a separate mode at the moment. It would be nice to have the ability to add line drawings to a picture that was being drawn in free-drawing mode, in order to mix the two. It could be implemented by opening up a menu of line-drawings and choosing one, specifying the size (small, medium, or large) and then placing the drawing on the virtual canvas. This would not be difficult to implement since the flexible menu system allows new menus to be created with a minimum of code. To display the picture on screen, a texture which samples the line drawing could be placed with its centre point specified by the current gaze point which would allow the user to move the drawing around the canvas with gaze. They could then give a voice command to stop updating the position of the texture. This would then be rendered to the main texture which takes up the whole screen, and become part of the overall picture. Another improvement to this system would be to have a folder titled "Line Drawings" which would be inside the main program folder. This could be read by the application on startup and any image files contained inside could be included by the application for this line drawing-mode. This would allow users to add extra line drawings from any source once the amount that are included in the application had been exhausted.

A settings menu could also be included to change certain parameters in the application. At the moment the eye tracker address is hardcoded, since only one was to be used in the project. It would be important to allow users to supply the address of their

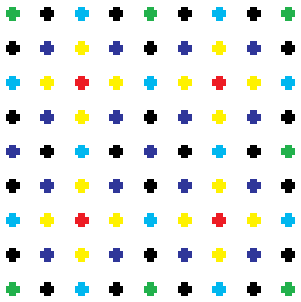


Figure 6.2: Grid points to aid drawing

own eye tracker however. This menu could also be responsible for changing how many points would be used for calibration since sometimes five can be enough for calibrating satisfactorily. Other features related to the drawing system could be set here such as the distance threshold for the snap function and what format the image file should be output as, at the moment the only possibility is .bmp. Due to the need to type in the eye-tracker address, and perhaps a sliding toolbar for the threshold, this menu would not be controlled by gaze. Able bodied users such as a subjects parent could make changes here by using a mouse.

Feedback on the project was received by both Acuity ETS and Tobii, who both felt that the free drawing mode would be too difficult for the majority of disabled users in its current format. The number of voice commands was felt to be too many and this amount could be reduced in several ways. 'Snap' could be changed to a feature which is turned on or off by means of a menu button. Activating and deactivating 'fix' could also be moved to a menu button. These changes would add two new buttons but remove four voice commands ('snap','leave','fix','unfix'). Also all four menus could be accessible from one central menu that would be opened by one voice command, resulting in lowering the amount of commands to remember by a further three. Not all users may wish to use the helper functions so access to these could be set in a settings menu. In order to suit the majority of disabled users it would be beneficial to have a mode that only recognized a noise being spoken to activate drawing, since some might have speech impediments which would prevent them from using all the voice commands. In this case, drawing could be started and stopped by making any noise, and a menu with perhaps less available drawing tools could be accessed by directing gaze to the top left corner of the screen. Due to the flexibility of the menu system

and the ability to change the voice commands quickly via an XML file, these changes would not be difficult to implement. They consist of re-arranging the functionality in a way that would benefit most users, and in some cases just cutting down on features that might be superfluous and distracting to most people.

There was also a lot of data collected during the user trials which was output during drawing. This was not considered in this project but could possibly be studied to provide further information on participants' drawing behavior, and any tools that they found difficult.

Appendix A

Questionnaire

Questionnaire:

Personal Questions:

Please tick the appropriate box/es:

Gender: Female: [ ] Male: [ ]

Age:_____

Have you ever used any of the following drawing programs?

Microsoft Paint [ ]
Adobe Photoshop [ ]
Paint.NET [ ]
Other (Please Specify) [ ] _____

Do you or any of your family have a history of epilepsy?

Yes [ ] No [ ]

Evaluation:

Please rate drawing using eye-tracking and voice commands under the following headings:

-How easy was it to navigate the menus? (1 - difficult, 7 - easy):
1 [ ] 2 [ ] 3 [ ] 4 [ ] 5 [ ] 6 [ ] 7 [ ]

-How much control did you have? (1 - little control, 7 - lots of control):
1 [ ] 2 [ ] 3 [ ] 4 [ ] 5 [ ] 6 [ ] 7 [ ]

-How fast did you draw? (1 - slow, 7 - fast):
1 [ ] 2 [ ] 3 [ ] 4 [ ] 5 [ ] 6 [ ] 7 [ ]

-How much precision of the gaze control did you have? (1 - imprecise, 7 - very precise):
1 [ ] 2 [ ] 3 [ ] 4 [ ] 5 [ ] 6 [ ] 7 [ ]

-How did you enjoy drawing with gaze and voice? (1 - unenjoyable, 7 - enjoyable):
1 [ ] 2 [ ] 3 [ ] 4 [ ] 5 [ ] 6 [ ] 7 [ ]

-How natural were the controls? (1 - unnatural, 7 - natural):
1 [ ] 2 [ ] 3 [ ] 4 [ ] 5 [ ] 6 [ ] 7 [ ]

-How easy was it to give commands with voice? (1 - difficult, 7 - easy):
1 [ ] 2 [ ] 3 [ ] 4 [ ] 5 [ ] 6 [ ] 7 [ ]

Please rate drawing using traditional keyboard and mouse under the following headings:

-How easy it is to navigate the menus? (1 - difficult, 7 - easy):
1 [ ] 2 [ ] 3 [ ] 4 [ ] 5 [ ] 6 [ ] 7 [ ]

-How much control you have? (1 - little control, 7 - lots of control):
1 [ ] 2 [ ] 3 [ ] 4 [ ] 5 [ ] 6 [ ] 7 [ ]

-How fast you draw? (1 - slow, 7 - fast):
1 [ ] 2 [ ] 3 [ ] 4 [ ] 5 [ ] 6 [ ] 7 [ ]

-How much precision of control you have? (1 - imprecise, 7 - very precise):
1 [ ] 2 [ ] 3 [ ] 4 [ ] 5 [ ] 6 [ ] 7 [ ]

-How much you enjoy drawing with mouse and keyboard? (1 - unenjoyable, 7 - enjoyable):
1 [ ] 2 [ ] 3 [ ] 4 [ ] 5 [ ] 6 [ ] 7 [ ]

-How natural are the controls? (1 - unnatural, 7 - natural):
1 [ ] 2 [ ] 3 [ ] 4 [ ] 5 [ ] 6 [ ] 7 [ ]

-Which drawing tool did you prefer?
[ ] Line [ ] Rectangle
[ ] Ellipse [ ] Polyline
[ ] Curve [ ] Bucket-Fill

-Which drawing tool did find most difficult to use?

[ ] Line [ ] Rectangle
[ ] Ellipse [ ] Polyline
[ ] Curve [ ] Bucket-Fill

-Do you think this application could benefit users for whom mouse and keyboard cannot be used as input?

Yes [ ] No [ ]

-Please explain why:

-Please write any other comments you might have here:

Appendix B

Instruction Sheet

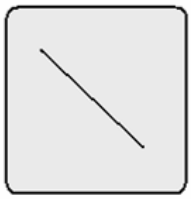
Instruction Sheet:

The application being tested is a paint program which uses an eye tracker to obtain information of where you are gazing at on the computer screen to control the cursor. Drawing commands are given using voice recognition. Here are a list of commands the application can recognize:

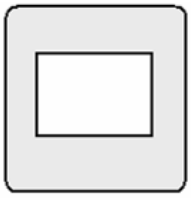
START starts drawing a shape
STOP stops drawing the current shape
SNAP starts drawing a shape at the closest corner or end of line/curve
LEAVE stops drawing a shape at the closest corner or end of line/curve
SELECT chooses a button on a menu screen
BACK closes a menu screen
UNDO removes the last shape drawn. Only possible when finished drawing a shape
FIX forces a line being drawn to be either vertical or horizontal. Only possible while in the middle of drawing a line
UNFIX allows a line being drawn to be at any angle. Only possible while in the middle of drawing a line

MENUS: (say 'select' to choose buttons)
OPEN COLOURS Opens the colours menu
OPEN TOOLS Opens the tools menu
OPEN THICKNESS Opens a menu to choose line thickness
OPEN FILE Opens the file menu

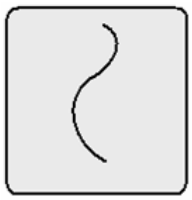
Here are the drawing operations that can be performed. By default, Line is the current drawing tool. These can be accessed by saying Open Tools



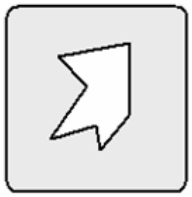
1. LINE: say **'start'** to start drawing a line at the current gaze point, a line will then be drawn from this point and wherever you are then gazing at the screen. Say **'stop'** when you are happy with where it is positioned. If you say **'fix'** it will be forced to be either horizontal or vertical. Saying **'unfix** will go back to normal. A line can also be started by saying **'snap,'** which will start it at the closest corner/end of line/curve, and stopped by saying **'leave'** which will stop drawing the line at the closest corner/end of line/curve, if there is one nearby.



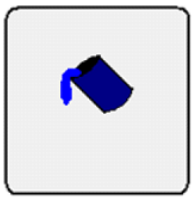
2. RECTANGLE: similar to line, say **'start'** to start drawing, and **'stop'** to finish. Can also be used with **'snap'** and **'leave'**



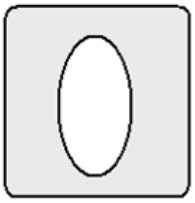
3. CURVE: This tool works by placing 4 points on the screen, a curve will then be drawn between them. Only **'start'** is used, say it each time you want to place a point. You can also say **'snap'** to place a curve point at the closest corner/end of line/curve.



4. POLYLINE: start using this tool by saying **'start'** or **'snap.'** A line will then be drawn. If you say **'start'** again, this line will be stopped, and another line started from the end of this, so many lines can continue in sequence. **'fix'** and **'unfix** can also be used while drawing these lines.



5. BUCKET-FILL: This tool fills enclosed regions with a colour. Say **'start'** to fill a region with the current colour.



6. ELLIPSE: Similar to rectangle, this tool draws an ellipse in the rectangular area given by 2 points. Say **'start'** to begin drawing, and **'stop'** when finished.

Finally, these are new technologies, and it helps to have patience while using them! If you find the cursor is slightly off from where you are looking at on screen, try not to follow it around but concentrate on aiming it where you want it to go. Also voice commands are not as immediate as mouse clicks so it helps to stay looking at a point until a command has been processed, for example, when you say select to choose a menu button, dont look away too quickly.

TASKS:

1. When application starts, choose FREE DRAWING to begin. (say select)
2. try using each drawing tool at least once to get used to them.
3. when you are ready, you will be given a picture to try and draw using gaze and voice, you will have ten minutes to do this. If you are finished early, try and add more detail to the picture.

Appendix C

Participant Data

Participant	Gender	Age	Paint Program Experience
1	male	21	Microsoft Paint Adobe Photoshop Paint.NET Fireworks
2	male	27	Microsoft Paint Adobe Photoshop Paint.NET
3	male	22	Microsoft Paint Adobe Photoshop Paint.NET
4	male	23	Microsoft Paint Adobe Photoshop Paint.NET Gimp
5	male	27	Microsoft Paint Adobe Photoshop Paint.NET
6	male	26	Microsoft Paint Adobe Photoshop Paint.NET Corel Painter
7	male	22	Microsoft Paint Adobe Photoshop Paint.NET
8	male	40	Microsoft Paint Paint.NET Gimp
9	male	23	Microsoft Paint Paint.NET Gimp
10	male	30	Microsoft Paint Adobe Photoshop Paint.NET Corel Draw Paint Shop Pro

Table C.1: Group One Participants.

Participant	Gender	Age	Paint Program Experience
1	male	20	Microsoft Paint
2	female	27	Microsoft Paint
3	female	25	Microsoft Paint Adobe Photoshop
4	male	27	Microsoft Paint
5	male	26	Microsoft Paint
6	male	30	Microsoft Paint Adobe Photoshop
7	female	23	Microsoft Paint
8	male	26	Microsoft Paint Adobe Photoshop Adobe Illustrator
9	female	25	Microsoft Paint Adobe Photoshop
10	female	22	none

Table C.2: Group Two Participants.

Bibliography

[1] T. N. Cornsweet. *Visual Perception.* New York: Academic Press, first ed. edition, 1970.

[2] A. T. Duchowski. *Eye Tracking Methodology, Theory and Practice.* Springer, second ed. edition, 2007.

[3] EyeArt. Gaze-controlled drawing program. `http://www.cogain.org/wiki/EyeArt`, Accessed 18 January 2010.

[4] EyeDraw. Eyedraw version 2.0. `http://www.cs.uoregon.edu/research/cm-hci/EyeDraw/evolution.html`, Accessed 13 September 2010.

[5] J. Faubert and A. M. Herbert. The peripheral drift illusion: A motion illusion in the visual periphery. In *Perception*, 1999.

[6] R. Grootjans. *XNA 3.0 Game Programming Recipes, A Problem solution Approach.* Apress, 2009.

[7] A. J. Hornof and A. Cavender. Eyedraw: enabling children with severe motor impairments to draw with their eyes. In *CHI '05: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 161–170, New York, NY, USA, 2005. ACM.

[8] H. Istance, R. Bates, A. Hyrskykari, and S. Vickers. Snap clutch, a moded approach to solving the midas touch problem. In *ETRA '08: Proceedings of the 2008 symposium on Eye tracking research; applications*, pages 221–228, New York, NY, USA, 2008. ACM.

[9] M. Kumar, J. Klingner, R. Puranik, T. Winograd, and A. Paepcke. Improving the accuracy of gaze input for interaction. In *ETRA '08: Proceedings of the 2008 symposium on Eye tracking research; applications*, pages 65–68, New York, NY, USA, 2008. ACM.

[10] libSDL. General FAQ. `http://wiki.libsdl.org/moin.cgi/FAQGeneral`, Accessed 4 September 2010.

[11] F. D Luna. *Introduction to 3D Game Programming with DirectX 9.0cA Shader Approach.* Wordware Publishing, 2006.

[12] MetroVision. `http://www.metrovision.fr`, Accessed 25 August 2010.

[13] A. Meyer and M Dittmar. Conception and development of an accessible application for producing images by gaze interaction EyeArt (EyeArt Documentation). Unit of Engineering Psychology and Cognitive Ergonomics Dresden University of Technology.

[14] MSDN. Microsoft Speech SDK 5.1 Help.

[15] MSDN. IDirect3DDevice9::GetFrontBufferData Method. `http://msdn.microsoft.com/en-us/library/bb174388/(v=VS.85).aspx`, Accessed 12 September 2010.

[16] MSDN. Implementing the Flood Fill Algorithm. `http://www.codecodex.com/wiki/Implementing_the_flood_fill_algorithm#C`, Accessed 11 September 2010.

[17] MSDN. OpenGL. `http://msdn.microsoft.com/en-us/library/dd374278(VS.85).aspx`, Accessed 4 September 2010.

[18] MSDN. "Speech API Overview.". `http://msdn.microsoft.com/en-us/library/ms720151(VS.85).aspx.`, Accessed 4 September 2010.

[19] MSDN. The Component Object Model. `http://msdn.microsoft.com/en-us/library/ms694363(v=VS.85).aspx`, Accessed 31 August 2010.

[20] MSDN. What is COM? http://www.microsoft.com/com/default.mspx, Accessed 31 August 2010.

[21] MSDN. Windows GDI. http://msdn.microsoft.com/en-us/library/dd374278(VS.85).aspx, Accessed 4 September 2010.

[22] J. O'Donovan, J. Ward, S. Hodgins, and V. Sundstedt. Rabbit run: Gaze and voice based game interaction. In *EGIrl '09 - The 9th Irish Eurographics Workshop*, Trinity College Dublin, Dublin, Ireland, 2009. EGIrl.

[23] OpenGL. GLSL - OpenGL Shading Language. http://www.opengl.org/documentation/glsl/, Accessed 4 September 2010.

[24] F. Sani and J. Todman. *Experimantal Design and Statistics for Psychology, A first Course*. Blackwell Publishing, 2006.

[25] Live Science. The eye. http://www.livescience.com/health/051128_eye_works.html, Accessed 25 August 2010.

[26] Tobii. Tobii SDK - Developers Guide.

[27] Tobii. Tobii T/X series Eye Trackers - Product Description.

[28] T. Vilis. The physiology of the senses transformations for perception and action. In *Course Notes - Lecture 11 - Eye Movements*. University of Western Ontario, 2006.