Abstract

The aim of the project was to design a Low Vision Aid using commercially available computing and/or entertainment hardware. The aid was designed primarily for the application of reading. A standard PC running Windows is the chosen platform for development. The design of a software application for scrolling enlarged text across a screen is presented. The problems of Line Tearing and slow rendering times from the PC display refresh were investigated. The Win32 GDI, OpenGL and DirectDraw graphics libraries were investigated for possible solutions to these problems. The final application was implemented using DirectDraw.

Declaration of Originality

I declare that this thesis is my original work except where stated.

____

Table of Contents

Abstract i

Declaration of Originality ii

Table of Contents iii

List of Tables vi

List of Figures vii

Glossary viii

Conventions ix

Chapter 1 Introduction 1

1.1 Aims 1

1.2 Motivation 1

1.3 Requirements 3

1.4 Project Planning 3

1.4.1 Research 3

1.4.2 Development 4

1.4.3 Trials 4

1.5 Report overview 5

Chapter 2 Background 6

2.1 Introduction 6

2.1.1 Macular Degeneration (MD) 6

2.1.2 Reading with macular degeneration 7

2.2 Reading Aids 8

2.2.1 Traditional LVAs 8

2.2.2 CCTV 9

2.2.3 Text to speech 10

2.2.4 PCs as Low Vision Aids 11

2.3 A PC based LVA: Motivation 11

2.3.1 LVA for computer use 11

2.3.2 LVA for reading only 12

2.4 Summary **13**

Chapter 3 Text Scroller Design **14**

3.1 Introduction **14**

3.2 Software Specification **14**

3.2.1 A Generic Class Framework 14

3.2.2 Adapting The MFC Framework 15

3.3 Application Design in MFC **17**

3.3.1 GDI 17

3.3.2 Formatting 17

3.3.3 Rendering onto display 18

3.4 Naïve Method for Scrolling **18**

3.4.1 Rendering to memory (double buffering) 19

3.4.2 Protecting the buffer 19

3.5 Improved Methods for Scrolling **20**

3.5.1 Alphabet Buffer Scrolling 20

3.5.2 Scrolling Window 21

3.6 Scrolling Text Implementation **22**

3.6.1 CReaderDoc Initialisation 22

3.6.2 CView Initialisation 22

3.6.3 Calling the scroller 22

3.6.4 Displaying the text 23

3.6.5 Updating the scroll buffer 23

3.6.6 Controlling the application 24

3.7 Analysis **25**

3.7.1 Improvements 26

3.8 Summary **26**

Chapter 4 Smoothing the Scroller **27**

4.1 Introduction **27**

4.2 Problem Description **27**

4.2.1 Line Tearing 27

4.2.2 Rendering Overheads 28

4.3 Generic Solutions **28**

4.3.1 Triple Buffering 29

4.3.2 Refresh Synchronisation 30

4.4 Investigation of Graphics Libraries **30**

4.4.1 GDI 31
4.4.2 OpenGL 31
4.4.3 DirectDraw 34
4.5 Revised Text Scroller 37
4.5.1 The DDGraphics class 37
4.5.2 The DDGraphics back buffer 38
4.5.3 The CreaderView Class 41
4.6 Analysis 41
4.7 Chapter Summary 42

Chapter 5 Feedback 43
5.1 Introduction 43
5.2 A recent development 43
5.3 Feedback on Supernova 43
5.3.1 Interface and control 44
5.3.2 Full screen 44
5.3.3 Text scrolling with speech 44
5.4 Further recommendation 45
5.5 Summary 45

Chapter 6 Summary and Discussion 46
6.1 Project Summary 46
6.2 Still to do 46
6.3 Should have done 47
6.4 Future Work 47
6.4.1 Medical trials 47
6.4.2 Adaptive scrolling 48
6.4.3 Control methods 49
6.4.4 Capture methods 49
6.4.5 Into the future 50

Acknowledgements **51**

References **52**

Appendix A.1 Initial Scroller Code **54**

Appendix A.2 OpenGL Based Code **58**

Appendix A.3 DirectDraw Based Code **60**

Appendix A.4 Mission Statement **70**

List of Tables

Table 3.1 Responsibilities and collaborators for View class *CReaderView* 16

Table 3.2 Responsibilities and collaborators for Document class *CReaderDoc* 16

Table 3.3 Responsibilities and collaborators for Frame class *CMainFrame* 16

Table 4.1 DDGraphics class responsibilities and collaborators 37

List of Figures

Figure 1.1 Map on an eye showing location of the retina and macula 2
Figure 2.1 Fundus image of the retina showing the macula area 6
Figure 2.2 Fundus image of healthy retina (left) compared with macular degeneration case (right)[3]. 7
Figure 2.3 Reading view of someone with severe macular degeneration – the left image showing normal text, the right showing magnified. 7
Figure 2.4 Typical binocular LVA 8
Figure 2.5 CCTV for reading (left) and head mounted for everyday use (right) 10
Figure 2.6 PC with screen magnifier application 12
Figure 3.1 Simple UML Class diagram for reader application 16
Figure 3.2 Typical MFC modal dialogue for changing text font and colour 18
Figure 3.3 Alphabet buffer method for rendering a string of text 20
Figure 3.4 Scrolling window method for rendering and scrolling a string of text 21
Figure 3.5 Maintaining continuous scrolling text before and after a render 24
Figure 3.6 Screenshot of the initial LVA reader application 25
Figure 4.1 Single thread execution of text scroller application 28
Figure 4.2 Single thread execution of scroller with immediate return blit 29
Figure 4.3 Execution of text scroller with rendering performed in a separate thread 29
Figure 4.4 Video architecture for Windows 95 – indicating both GDI and DD interfaces between applications and hardware 34
Figure 4.5 Page flipping with three surfaces in DirectDraw 36
Figure 4.6 Simplified UML class diagram including DDGraphics 37
Figure 4.7 Wide buffer emulation using three surfaces in *DDGraphics* 38
Figure 4.8 Blitting from two surfaces showing split source and destination scrolling windows 39

Glossary

ASCII	American Standard Code for Information Interchange
API	Application Programming Interface
blit	An operation that copies a block of bits
CCTV	Closed Circuit Television
CRT	Cathode Ray Tube –used for computer monitors
DD	DirectDraw
DDI	Device Driver Interface
EOG	Electro-oculography – tracking minute electrical pulses around the eye
Fundus image	Medical image of the back of an eye (retina)
GDI	Graphics Display Interface – usually with reference to Win32 GDI
HEL	Hardware Emulation Layer
html	Hypertext mark-up language
HUD	Head-Up Display
LCD	Liquid Crystal Display – flat-screen displays
LVA	Low Vision Aid
MD	Macular Degeneration
MFC	Microsoft Foundation Classes
OO	Object Oriented (Programming)
PAEP	Princess Alexandra Eye Pavilion, Edinburgh
PC	Personal Computer
PRL	Preferred Retinal Loci
UI	User Interface
UML	Unified Modelling Language
WIMP	Windows, Icons, pull-down Menus and Pointer style interface
Win32	Windows 32-bit – API for Microsoft Windows 95 and NT

Conventions

Coding

Any coding referred to in the main body of text is highlighted in *Italic*. The coding convention adopted in the Appendices is as follows:

- Different font used is `Courier New`
- Keywords and return types in **Bold**
- Comments in *Italic* and preceded by a double slash (//)

The code provided in the Appendices is merely intended as supplementary examples to the main text, as such code for dealing with errors and assertions has been removed to aid clarity. The complete code listing is in the accompanying CD.

Footnotes

Footnotes within the text are indicated using superscript Latin symbols, e.g. ζ, ψ , ξ…

Chapter 1
Introduction

1.1 Aims

The initial aim of this project was to research and develop a novel aid for people with low vision. In particular the low vision aid (LVA) was to be designed specifically for the task of reading. The key elements of the aid were that it should be:

- Effective – it should function *at least* as well as existing aids.
- Cheap – making use of 'off the shelf' technology to be as affordable as possible.
- Useable – convenient and easy to use.

Research into existing aids available for low vision reading was required in order to help build a specification for the new LVA. An analysis of existing technologies, particularly with displays and methods of capturing text for reading was also required. The approach was then to build the text display part of the aid in software on a PC running Windows 95. Following from this it was planned to perform medical trials of the program on low vision patients at the Princess Alexandra Eye Pavillion (PAEP.)

Early on in the software implementation phase however, it became apparent that development of the LVA display would require more work than initially anticipated. Unforeseen problems, relating to the way in which a PC deals with scrolling graphics, posed a new, more software-oriented challenge. The aim of the project shifted away from the more general LVA design towards solving the problems with the display.

1.2 Motivation

Low vision affects nearly one in three of the world's population. It is generally characterised as any loss in visual acuity or field that cannot be corrected by lenses or spectacles. The causes of low vision are varied, but by far the most common is macular degeneration. This is where the macula, or central part of the retina where frontal vision is dealt with, deteriorates over time[1,2,3,4]. See Figure 1.1. The upshot of this is that the sufferer loses all or part of his or her forward vision.

In order to see properly, sufferers must make greater use of their peripheral or side vision. The problem with this is that peripheral vision lacks the acuity of forward vision, and as such cannot make out details such as text or far-off objects. Loss in detail can be partially compensated for by enlarging or magnifying the article to be viewed. This is the job of the low vision aid.

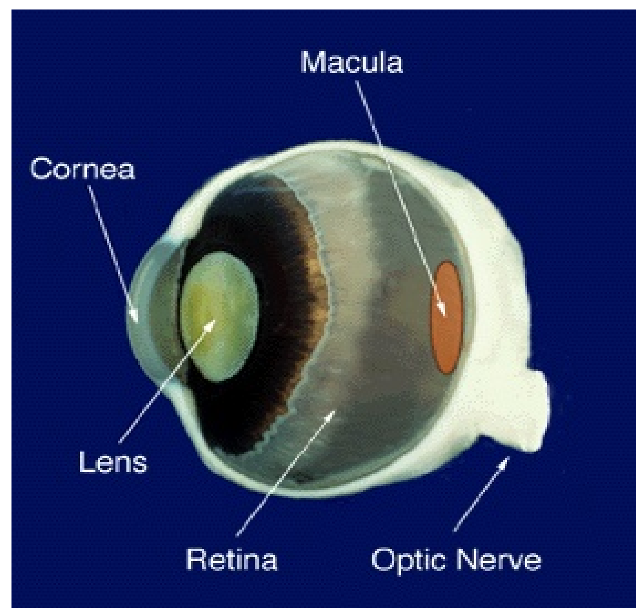

Figure 1.1 Map on an eye showing location of the retina and macula

Traditional aids such as the telescopic monocular, consisting merely of lenses and a tube (much like a small hand-held telescope), can be used to magnify objects for viewing, albeit at the cost of a restricted viewing field. More modern aids based on closed-circuit television, or CCTV technology help address many of the problems associated with lens based aids, but often at the price of being too bulky or expensive to use.

The advent of the home computer, together with continual miniaturisation of technology may hold the key to a better solution. Hardware and software for scanning, processing and displaying text is relatively cheap – particularly when included as part of a 'multimedia package' with most new home PCs. In addition to scanning paper-based text, the Internet provides a vast source of text already in electronic form.

Some software already caters for the needs of low vision users. Screen magnifiers and software text-to-speech readers are all available, but remain largely unused. Part of the problem lies with the added complexity of a computer interface, and on the focus of an aid being directed at general purpose computer-use. Additionally, many sufferers of low vision are elderly, and as such are generally less able or willing to learn how to deal with even the simplest technological devices[5].

1.3 Requirements

The LVA application shall be based around scrolling large text across a screen. Initial requirements of the LVA are outlined below:

- Real-time navigation – should allow dynamic interactions for moving around the text. In particular changing speed and direction of the text flow.
- Text fully customisable – attributes such as size, font, contrast and colour can be altered either dynamically at run-time, or as part of the program set-up procedure.
- Suitable interface - should be designed with a low vision user in mind. Ideally, use of visual controls, such as buttons and pull-down menus should be kept to a strict minimum.

Some possible future additions to the application may include further navigation features, such as letter, word or paragraph jumps and or a find/search facility. Some form of audio feedback should also be considered.

1.4 Project Planning

The project was initially broken into roughly three stages: research, development and trials. In practice, these stages overlapped, with much of the research continuing right up until the end of the project. The outcome of each stage is outlined below.

1.4.1 Research

Much of the initial research time was spent talking to doctors and patients at the Princess Alexandra Eye Pavilion (PAEP) in Edinburgh[6]. This was used to gain a better understanding of the problems associated with low vision, in particular with macular degeneration. Additional material on the subject was obtained from various journals, newsgroups and Internet. Much of the information on existing electronic aids and their usage was provided by the resource centre of Edinburgh District Libraries[7].

1.4.2 Development

Based on the requirements obtained from the preliminary research into low vision aids, an initial specification was drawn up in accordance with the project Mission Statement. The initial software development was carried out using the Java development package. This was chosen due to the inherent cross-platform appeal of Java. This would have allowed the final application to be run on a variety of platforms, ideally leaving scope for a Java applet implementation that could be used as a web based LVA.

Unfortunately, early into the project it became apparent that the application would be fairly demanding both processor and memory wise due to the full screen graphics involved. Java was deemed inappropriate for such intense system usage, and so the lower-level C++ based Win32/MFC platform was adopted instead.

The first LVA application was then built using the Microsoft Foundation Classes (MFC) on a Windows 95 based PC. It was at this stage that complications began to surface with the standard Windows Graphics Device Interface (GDI).

In an effort to find a solution to these complications, the OpenGL and DirectDraw graphics libraries where investigated at length. Two separate implementations of the application were developed using these respective libraries. In the end, DirectDraw was chosen for the final implementation.

1.4.3 Trials

The original project Mission Statement specified evaluation of the LVA through extensive patient trials. Due to the problems mentioned above, these trials were never carried out. An informal investigation of a similar LVA was however carried out at the Edinburgh Library’s resource centre later in the project.

1.5 Report overview

This report is split into six chapters. The contents of which are outlined in brief below:

Chapter 1: Introduction

This chapter introduces the aims and motivations for development of a software LVA. Outlines the project structure and design process, and gives the application requirements.

Chapter 2: Background

This chapter goes into more detail on the problem of macular degeneration and the problems with existing reading aids. Supplementary research relevant to the problem is discussed, together with a brief look at some of the technology available. It also looks at how this may eventually be utilised for developing LVA.

Chapter 3: Text Scroller Design

This documents the main body of work in the development of the initial scrolling text application. The specification is given in detail and from this a basic program structure devised. The implementation of each key element in the design is then summarised, together with support for decisions made and reference to actual code where relevant. The operation of the working LVA application is analysed and problems highlighted.

Chapter 4: Smoothing the Scroller

This chapter deals mainly with solving the problems highlighted in Chapter 3. In particular the problem of line tearing on the display refresh is discussed. Possible solutions to the problem are considered, which includes a comparison of the OpenGL and DirectDraw graphic libraries. The design of a revised text scroller application is given and analysed accordingly.

Chapter 5: Feedback

This is a brief chapter discussing feedback from preliminary trials of a similar program. Some possible improvements to any future development of the application are suggested.

Chapter 6: Summary and Discussion

Summary of the project with a discussion on future work that may be carried out in this area. A discussion on the possibility of medical trials into low vision reading using this technology is also included.

Chapter 2
Background

2.1 Introduction

This chapter attempts to provide some background into the problems associated with reading in low vision sufferers. Existing reading aids are investigated, and the motivations for development of a new software based aid given.

2.1.1 Macular Degeneration (MD)

Macular degeneration (MD) is one of the most common reasons for severe visual impairment. Sufferers range from having 'blurry' vision to almost complete blindness. In most cases, the disorder does not prevent them from going about their daily lives, but if the condition deteriorates, ability to read becomes harder. Although the condition affects many people as they get older, it can develop at any age through a variety of causes, such as diabetes.

The macula is the area of the retina where most of our central vision is dealt with. A healthy retina has a large concentration of light receptive cones in the macula; this corresponds to a high resolution in vision. The highest concentration of these cones is in the centre of the macula at the fovea, this is where vision is most acute. Blood vessels on the retina tend to flow around the edges of the macula. Here cones are sparse and hence resolution is low. This is the part that deals with peripheral (side) vision. Figure 2.1 shows a fundus image from a healthy retina.

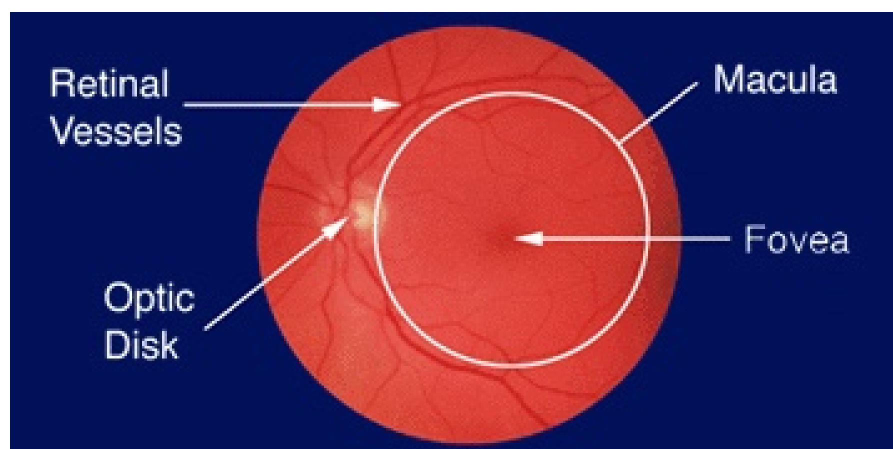

Figure 2.1 Fundus image of the retina showing the macula area

In cases where the macula has become damaged or obscured, the resolution and contrast sensitivity of the macula deteriorates and frontal vision suffers as a result. Figure 2.2 shows a comparison between the fundus image of a healthy retina and that of one with macular degeneration (MD).

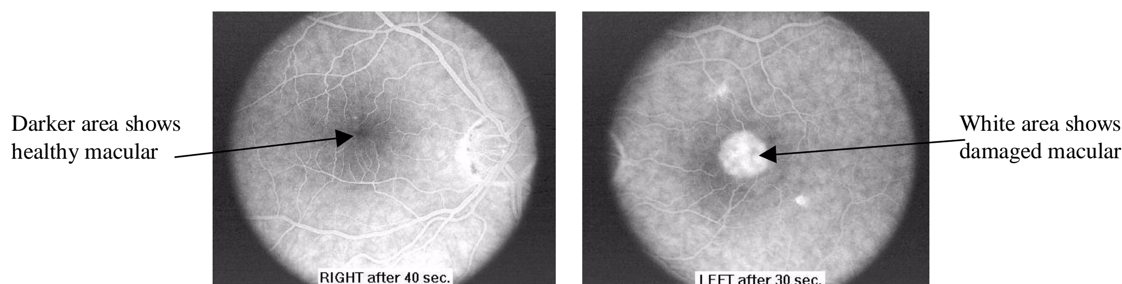

Figure 2.2 Fundus image of healthy retina (left) compared with macular degeneration case (right)[3].

2.1.2 Reading with macular degeneration

It is the brain's responsibility to direct images to the best suited part of the retina—the Preferred Retinal Loci (PRL)[8]. In normal healthy eyes, the PRL is fixed on the central macula and is used for most vision tasks—particularly with reading, where high resolution is desirable. In cases where the macula is obscured or damaged, the PRL is forced to fixate on less acute areas of the retina. This loss in visual acuity can effectively render tasks such as reading almost impossible. Figure 2.3 shows the likely view of someone with macular degeneration reading both normal and magnified text.

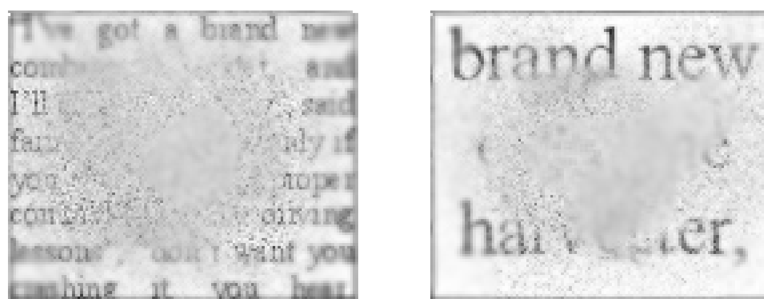

Figure 2.3 Reading view of someone with severe macular degeneration – the left image showing normal text, the right showing magnified.

Enlarging or magnifying text can help to compensate, although not correct, this loss in acuity. Recent studies have also shown that the brain can adopt new reading strategies around the macula defects. This may involve constantly moving the PRL fixation to suit whatever is being read.

2.2 Reading Aids

There is a surprisingly small range of aids available for people with severe vision problems. Although there are a few electronic (e.g. CCTV) aids, the most common involve antiquated optical lens technology such as eyeglasses and telescopes. People with macular disorders use such aids to magnify images onto their lower resolution periphery. The magnification compensates for loss in visual acuity, but this is usually at the expense of a reduced field of view – particularly when using monocular LVAs. Another problem with increased magnification is the corresponding increase in image motion. This can disorientate the brain causing so-called image slips[9], where even the slightest movements can cause visual acuity to suffer - in some case inducing severe motion sickness.

2.2.1 Traditional LVAs

Traditional lens based reading aids are typically available in monocular or binocular form. These are small telescopes that can be handheld or fixed to a sturdy set of spectacle frames, as shown in Figure 2.4. The magnitude of such aids is typically in the range of x2 to x7. These are used by people with a wide range of vision disorders and rarely provide a ‘perfect’ solution. Generally, the fixed focal length of (say) a monocular requires that in order to read, the user must hold the text at a constant distance from the lens. Finding the correct distance is one problem, finding the text itself—especially with a magnification greater than x3 -- can be another. It often takes much effort to get used to such a device—some people never get used to them.

Figure 2.4 Typical binocular LVA

Greater magnification and a hugely reduced field of view speeds up even the slightest movements. The resultant image slips can lead to disorientation and in many cases headache. When reading text it is important to avoid such slips, so as not to loose the reading place. The reader must scan text slowly, word-by-word—or even letter by letter in more severe cases. The problem with this is

that the brain becomes preoccupied with text recognition and hence suffers in understanding. At the end of a line of text, the reader may even have forgotten what was at the beginning.

Further complications arise at the end of a line. In normal reading conditions the eye moves in saccades, progressively fixating at points along a line, before jumping to the start of the next line – the carriage return[10]. The brain deliberately blocks reading during this jump so as not to disrupt the flow of reading. When reading through a monocular, the carriage return must be made more carefully so as not to cause image slip and loose track of the line. This has the knock-on effect of introducing a break in reading flow during which extra text may be read causing confusion.

Invariably the reader may suffer from some form of arthritis, nerve problem or loss of motor skills: both holding the device and scanning the paper (or book) for any length of time becomes a very difficult task. Sadly, for some people whose vision deteriorates, the mental and physical fatigue of reading becomes all too much. What once may have been a leisurely and enjoyable pursuit, can quickly become an unbearable chore. Fortunately, in many more cases, these inadequacies are overcome. People can learn to adapt with the help of even the most primitive LVA devices.

2.2.2 CCTV

There are numerous Closed Circuit television (CCTV) systems available for people with low vision[11]. CCTV is generally the term applied to any electronic LVA. A basic CCTV is comprised of a video camera pointed at the object or text to be viewed, and a television monitor for viewing. See reading CCTV in Figure 2.5. Control circuitry allows the user to move the camera around and magnify or alter the display in some way. This is the advantage of a CCTV over lenses; it can have features such as greater magnification, wide viewing range, adjustable brightness and artificially increased contrast.

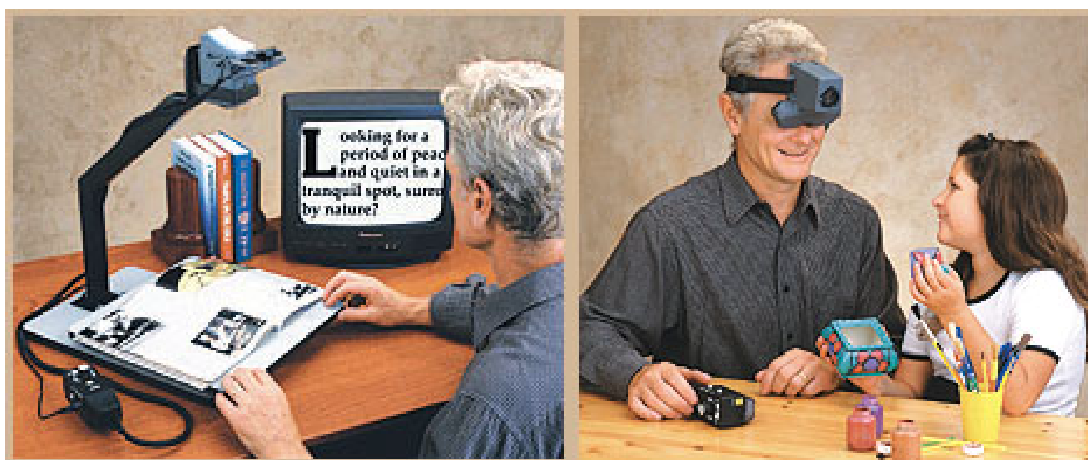

Figure 2.5 CCTV for reading (left) and head mounted for everyday use (right)[12]

Despite these benefits, CCTV is not as widely used as optical devices. Part of the problem with CCTV technology is that it has been, until recently, fairly bulky in size. A large TV screen and camera may be somewhat of an inconvenience for day-to-day use when compared with a small hand-held monocular. There are smaller and lighter head-mounted based CCTVs available, incorporating advanced features such as automatic focus[13], but these tend to use specialised technology and can be very expensive. See head mounted CCTV in Figure 2.5. Advances in technology, driven by the lucrative home entertainment markets, have brought the size and price of such systems down, but there is still some way to go.

Another factor of quality is in ease-of-use. People suffering from low vision tend to be elderly, and as such less willing to learn how to use new ‘gadgetry’. If the use of an LVA does not ‘feel’ natural or convenient, people will inevitably go back to using what they are used to – despite the failings.

2.2.3 Text to speech

Advances in text-to-speech technology has brought many benefits for low vision users. Stand-alone text-to-speech readers are available that can scan any source of paper based text, from newspapers, magazines to books and bills. The text is extracted and read out to the user. Text can be navigated using a simple keypad, with feedback provided vocally by the machine[14].

The initial problem with this technology was the lack of a ‘natural’ sounding voice. This is now becoming less of a problem. Such machines can be quite effective as a reading aid – not just for low vision sufferers, but for the totally blind. As with CCTVs though, the text-to-speech stand alone units can be fairly bulky and expensive.

2.2.4 PCs as Low Vision Aids

A solution to the expense problem of electronic low vision aids can be found in the use of the home PC. Many PCs these days come packaged with facilities for scanning paper based image and text. Once scanned, images can be manipulated in many ways: magnification, colour, contrast enhancement, and even text extraction. There are several software based ‘PC CCTV’ packages available specifically for the low vision computer user. Together with relatively inexpensive text-to-speech software the home PC can be used both as a CCTV and a text-to-speech reader combined. The only real cost involved is with the software[15,16].

PC based LVAs are still in their infancy however, and there are many problems associated with their use. These will have to be overcome before such aids are useable enough for widespread acceptance.

2.3 A PC based LVA: Motivation

Much of the motivation for this project comes from the much neglected potential of merging existing ‘off the shelf’ technology with medical applications. There is already a wealth of cheap video technology driven by the massive potential in the home entertainment market. Miniature video cameras and high resolution displays are just some of the products that with minimal adjustment could be used to build high-tech low vision aids of the future.

The use of an electronic low vision aid can be split into two categories: aid for using a computer, and computer based aid for reading. Although this project shall focus on development of an aid for reading, some of the wider problems associated with aids for computers shall be discussed.

2.3.1 LVA for computer use

Most of the so-called PC CCTV systems available are developed with a general purpose day-to-day computer user in mind. Such aids are essential as it is becoming increasingly necessary for computers to be accessible to all, such is the growing importance of computers in everyday life. The problem is that often these software aids are no more than mere add-on ‘hacks’ that fit uneasily on top of the existing computer interface. These rarely provide an efficient and practical solution to the needs of a low vision user.

As an example of this, consider the case of a typical software LVA on a Windows based platform - the screen magnifier. Screen magnifiers generally work by enlarging part or all of the desktop

area of the screen. The user can move around the screen by dragging the mouse beyond the magnified area's boundary. This can work quite effectively if the user is only working with one part of the screen, but what if an error message appears on another section of screen? It is likely that the user will not see the error message until he/she scrolls the magnified area over it.

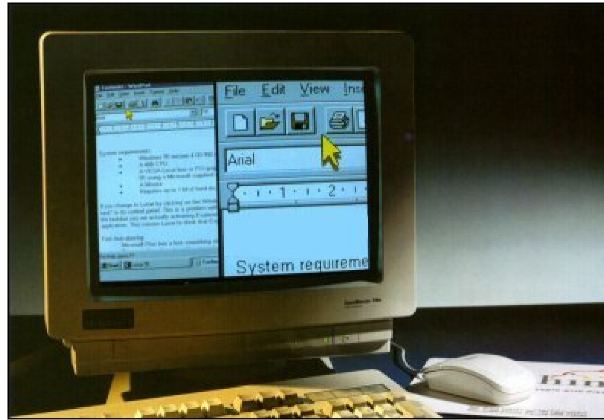

Figure 2.6 PC with screen magnifier application

There are many more such problems with this and other software based LVAs. The problem seems to lie with the fact that existing computer interfaces, such as the Window's desktop metaphor, were simply not designed with low vision users in mind.

2.3.2 LVA for reading only

Falling short of a complete re-design of the standard Window's interface, the approach used in this project concentrates on the specific task of reading. It is hoped that by narrowing the domain down to a reading specific software application, a more effective solution can be built. If fruitful, this may yet lead the way to a more general purpose aid.

Other issues in the general design of a PC based LVA for reading are identified as follows:

- Where does the text to be read come from
- How should the text be displayed for optimal low vision viewing, and
- How can it all be controlled

2.3.2.1 Where does the text come from?

As mentioned previously scanners and digital cameras are widely available for PCs. This allows easy conversion of paper based text sources onto computer for enhancement in low vision use.

It may be the case that today most of our reading tasks are carried out using paper based sources, but with ever increasing use of the Internet, more and more information is becoming available electronically. Most national newspapers are already available 'on-line', as are many books. Increasing use of email is already taking over the traditional postal service as a means of communication. Some telephone and electricity companies have already started billing customers by email. As the text is already available electronically, it may be extracted and displayed using the software LVA with minimal fuss.

2.3.2.2 How should the text be displayed?

There have been many advances in digital displays over the last few years. It is now possible to have a fairly large, full colour, high resolution LCD flat-screen display for a relatively inexpensive price.

For portability and ease of use, there are already head-mounted displays available, in some cases weighing less than a pair of bifocals. Video glasses, such as the Sony Glasstron, developed primarily for the home entertainment market allow a direct link-up to a portable PC[17]. Some PCs, in particular the so-called 'wearables' are so small they can be sewn in to the lining of a jacket.[18]

2.3.2.3 How should it be controlled?

The standard mouse and keyboard is an effective interface for computers. If the application is to be made for wider non-computer use as an LVA, these devices are unnecessarily complicated. Particularly when designed for use by elderly or non-computer literate low vision sufferers.

From the point of view of control, a simpler control may be all that is required, such as a simple button for controlling text flow, or a joystick/control pad as used with video games.

2.4 Summary

The necessary background into the problem of low vision has been given. Some of the problems associated with existing low vision reading aids have been identified, and the motivations for a novel solution using existing technology given.

Chapter 3
Text Scroller Design

3.1 Introduction

This chapter outlines the design of a text scrolling application for use as a low vision reading aid. A specification is given based on some of the requirements and low vision needs discussed previously. A suitable user interface (UI) is also identified. From this, an initial program structure is devised and implemented. Issues arising from this initial implementation are discussed and any major problems identified.

3.2 Software Specification

An Object-oriented (OO) approach was used in the design of the text scroller application. Based on the perceived requirements for an electronic low vision aid, as discussed in the last chapter, a preliminary software specification was put together using UML (Unified Modelling Language[19]) use cases and class diagrams.

3.2.1 A Generic Class Framework

When choosing a set of classes for an OO design, it often helps to base the high-level class structure on an abstraction of some relevant 'real world' situation. Individual classes and their responsibilities can be chosen from the key parts of that abstraction. It seemed natural to base the design abstraction for this application on a traditional monocular LVA. The main components of a text reading scenario where a monocular is being used is the document being read, the lenses and the telescope holder.

Use-cases were drawn up and used to identify the main components used in a typical low vision-reading scenario.

- The *Document* stores page(s) of text. It may be in many different forms, e.g. newspapers, books, bills and letters.

- The lenses are responsible for magnifying and altering the *View* of the text in a way that is useful for the reader. Magnification is changed by adjusting the distance between the lenses, and or replacing the lenses themselves. The lenses may be tinted using filters if necessary, e.g. to assist partially colour blind users.
- The telescope holder (or *Frame*) allows the user to adjust the distance between lenses. It also gives the user a handle to control the positioning and movement of the device.

From an object-oriented programming point of view the *Document*, *View* and *Frame* abstract of the telescopic monocular provides a good basis for any future class structure.

3.2.2 Adapting The MFC Framework

The Microsoft Foundation Classes (MFC) that come packaged with the Visual Studio C++ development suite was used in development of this application.

Built on top of the Win32 API (Application Programming Interface), MFC provides a collection of useful skeleton classes for building Windows based applications. Much of the code for setting up windows and interpreting user commands is already provided. Classes for reading and writing to files is also provided. MFC allows straightforward development of applications without the distraction of having to code commonly used window interface features.

An advantage of using MFC was that it already had an application framework that correlated nicely with the monocular abstraction given in section 3.2.1. The basic *Frame* class (*CMainFrame*) provides a mechanism for handling user interactions and passing them on to the relevant classes for processing. The *Document* class (*CDocument*) provides a basic mechanism for reading and writing to files. The *View* (*CView*) class provides mechanisms for processing and writing the application's display.

The classes *CDocument* and *CView* were subclassed and responsibilities specific to this application built onto them. The responsibilities and collaborations are given below for these subclasses: *CReaderView* in Table 3.1, *CReaderDoc* in Table 3.2. *CMainFrame* is included for completeness in Table 3.3.

Class name: **CReaderView**	Collaborator(s):
Responsibility: • Format text (font, size, colour) • Render text (position, actual screen format and size) • Show text on screen • Scroll text across screen • Retrieve text from Document	*CReaderDoc*

Table 3.1 Responsibilities and collaborators for View class *CReaderView*

Class name: **CReaderDoc**	Collaborator(s):
Responsibility: • Load text from file • Translate to plaintext	[file system]

Table 3.2 Responsibilities and collaborators for Document class *CReaderDoc*

Class name: **CMainFrame**	Collaborator(s):
Responsibility: • Handle user input • Display controls and menus • Forward commands to Doc and View objects	*CReaderView* *CReaderDoc*

Table 3.3 Responsibilities and collaborators for Frame class *CMainFrame*

The corresponding class diagram, showing proposed member functions is shown below using UML notation in Figure 3.1.

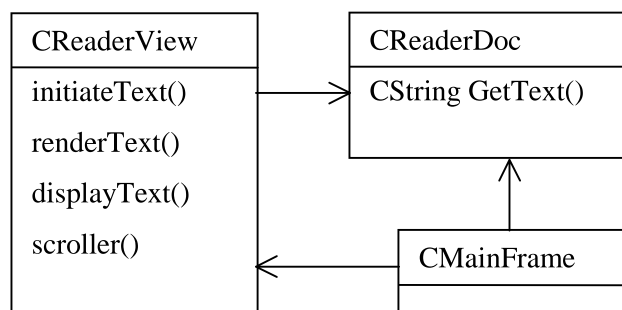

Figure 3.1 Simple UML Class diagram for reader application

3.3 Application Design in MFC

Due to the predominantly graphical nature of the scrolling text application, most of the interactions handled by the *Frame* class are passed onto *CReaderView* for processing. Part of the appeal of using MFC is that the user interface comes almost 'for free'. Indeed most of the setting up of menus, control buttons and shortcut keys are handled automatically using the API's 'Class Wizard'. Additionally, MFC provides standard pop-up or modal dialogue boxes for changing attributes such as colour and font.

Most of the implementation details of dealing with files and text sources are dealt with already using the *CDocument* based class. This can easily be adapted to deal with many different formats of input file. It can be linked with external libraries of code that are written specifically for parsing different formats of text into plain text ASCII. This means that the scroller application itself only ever has to handle plain text.

3.3.1 GDI

The Win32 Graphics Display Interface, or GDI, provides a wide variety of functions for formatting and then rendering to display. It allows programming of applications using low-level display functions without being bogged down by device specifics. Most of the calls to GDI will be carried out in the *CView* class as this is where display work for the application is dealt with.

3.3.2 Formatting

The GDI provides many useful functions for formatting text. The size and font details can all be stored within a *LOGFONT* structure. This structure can be kept active throughout the *View's* runtime by specifying it as a member variable, *m_logfont*. The specific details of the font, such as line spacing and character widths can be taken care of by using the MFC class *CFont*. This sets up a *LOGFONT* structure with suitable default values, while allowing changes if necessary using its member functions. The colour of the text and its background can be set in a similar manner.

Manual setting of the font can be carried out using a pre-defined MFC dialogue box such as that shown in Figure 3.2.

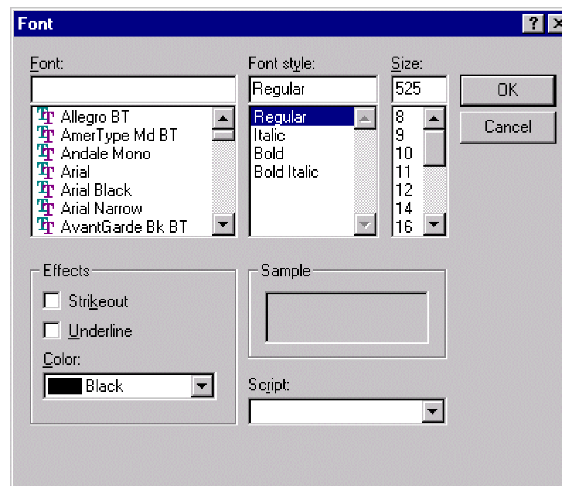

Figure 3.2 Typical MFC modal dialogue for changing text font and colour

3.3.3 Rendering onto display

The GDI provides an abstraction from the device specifics of the display in the form of a device context. A device context is a section of memory that can be mapped to output devices such as printers as well as portions or the entire display screen. The GDI provides functions for easy manipulation of device contexts.

When rendering an image or text to screen, a pointer to the screen's device context is required. Calling the GDI's *GetDC* from within the *CReaderView* class returns a pointer to the client window's device context. This is the portion of screen within the current *View* window. The current font and size can then be selected into the device context using the *SelectObject* function. Similarly, the background and text colours can be set using the functions *SetBackgroundColor* and *SetTextColor*.

A string of text can be rendered into position on screen context using the function *TextOut.* Its position can be specified using *x* and *y* co-ordinates relative to the client screen. The colours and font previously selected are applied automatically to the rendered text. Bitmap and images can be rendered in a similar manner.

3.4 Naïve Method for Scrolling

A naïve approach to scrolling the text would be to call the *TextOut* function within a loop, incrementing the *x* co-ordinate each time. The previous output must also be cleared with each increment so as to prevent the screen becoming cluttered and unreadable. This can be done most

efficiently by creating a rectangle the same size as the text region and colour of the background and selecting it onto the device context before each call to *TextOut*.

In practice, the continual clearing and rendering of text causes a noticeable 'flicker' on screen. Additionally, the rendering process initiated by *TextOut* is fairly processor intensive and can take several tens or hundreds of milliseconds. With this delay imposed by rendering on each refresh, the scrolling becomes noticeably slow and jerky. This is unacceptable for any practical usage.

3.4.1 Rendering to memory (double buffering)

One solution to the redraw 'flicker' is to create an off-screen device context in memory. The memory device context is made compatible with the screen context using the GDI function *GetCompatibleDC*. It acts as a buffer to which text can be rendered to in a similar manner as with directly to screen. This technique is known as double buffering.

The GDI provides a *BitBlt* function for blitting, or bit copying, between display contexts. The off-screen buffer is copied directly onto the screen only when rendering has finished. This can be carried out much faster than rendering directly to screen, and as only a complete image is copied each time, the 'flicker' caused by screen redraw can be reduced.

3.4.2 Protecting the buffer

An issue that arises with multitasking platforms such as Windows is 'who owns what?' In Windows, a device context can be accessed by any of the running programs. As there may be several programs running at any one time, including the window manager, the chance of a newly rendered context being overwritten is high.

Win32 applications are responsible for updating their own windows. This includes their visible portion of a screen as well as any buffers in memory. To allow for other applications also making use of off-screen buffer space, the application should be set up to make a 'backup' copy of its buffer space each time it is rendered to.

One way of making a backup is to copy the buffer onto a bitmap image. The GDI function *SelectObject* can be used to select a bitmap in and out of the device context. When a new bitmap is selected, any calls to write to buffer are also written to bitmap. The bitmap can then be selected out again for safekeeping, freeing the buffer context for use in other applications.

Whenever the buffer is required again, the bitmap is simply reselected, overwriting any unwanted information in the buffer that was left over from other applications.

3.5 Improved Methods for Scrolling

Using *BitBlt* from an off-screen buffer may help remove flicker from screen redraw, but it does not remove the need to render a complete image on each iteration, and so the rendering overheads remain. Two different approaches were taken to this problem; these are best described as the “Alphabet Buffer” and “Scrolling Window” algorithms.

3.5.1 Alphabet Buffer Scrolling

One approach to reducing rendering time would be to render a complete alphabet onto the off-screen buffer. This would effectively require rendering once only. A suitable blit operation can then be used to select individual letters as required and copy them in position to the screen. This is shown in Figure 3.3.

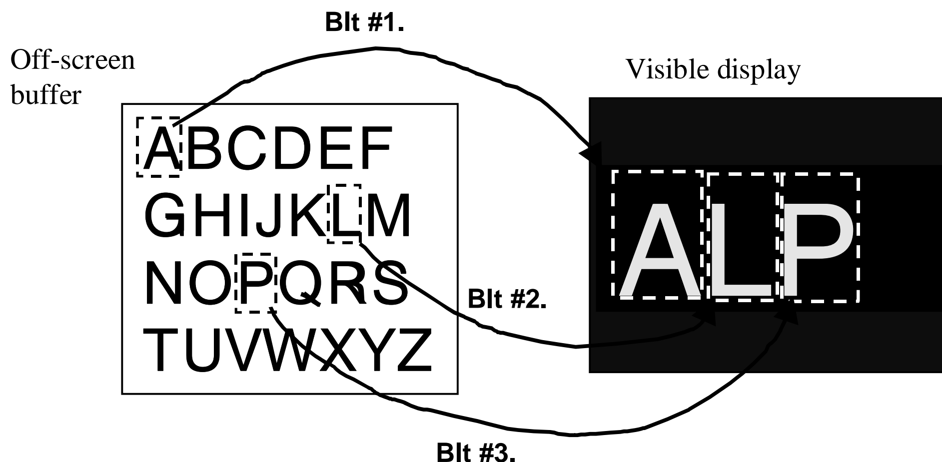

Figure 3.3 Alphabet buffer method for rendering a string of text

This method makes use of a variation to GDI’s *BitBlt* function, *StretchBlt*. *StretchBlt* expands the content of the source rectangle to fill the destination. This magnification results in a loss of text resolution, but this can be minimised if the buffer is sufficiently big and the rendered font being used is large enough.

The major advantage of this method is that the alphabet need only be rendered at set-up or when a change is made to the text font.

Colour changes do not necessitate re-rendering as text and background colours can be selected directly into the screen context. The off-screen buffer can be configured to store the image as a monocolour bitmap. The *StretchBlt* operation can then be configured to merge copy this bitmap with the colours specified for the destination screen. This method of rendering has a further advantage of reducing the memory used by the buffer.

The disadvantages of this method come with the added processor overheads of *StretchBlt* and the need to call a blit for each character being displayed. This leads to a noticeable degradation in scrolling speed. Additionally, the time differences between each call to *StretchBlt* causes the characters to jitter disturbingly.

3.5.2 Scrolling Window

An alternative way of coping with the overheads of rendering is to create an off-screen buffer that is much wider than the screen and using a smaller clipper window to copy to display. GDI's *BitBlt* allows copying between sub-rectangles of the source and destination device contexts. This means that a large section of text can be rendered to the off-screen buffer, with a small sub window being used to copy portions of the buffer to screen. The scrolling can be achieved by incrementing the starting x co-ordinate of this source window and calling *BitBlt*. See Figure 3.4.

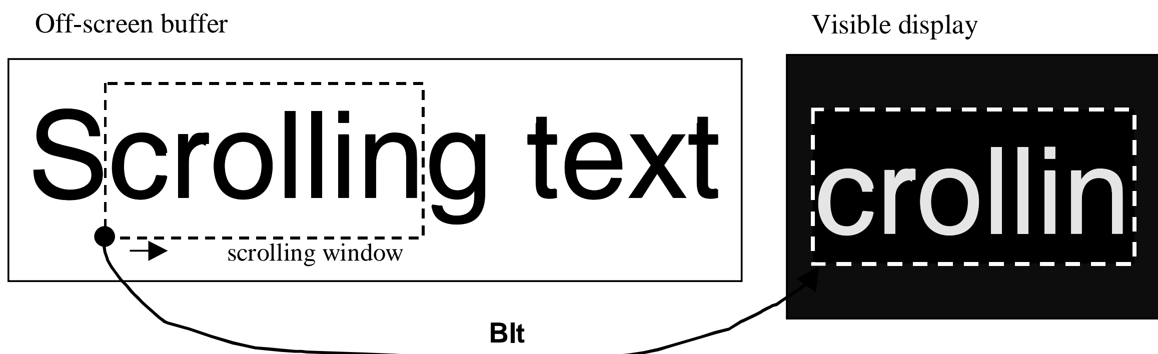

Figure 3.4 Scrolling window method for rendering and scrolling a string of text

The rendering function need only be called at the end of each text section being scrolled. If the buffer is big enough to accommodate an entire sentence, then the slight delay in scrolling can correspond to the reader 'catching a breath' at the end of a sentence. In this way the rendering overhead would not be such an inconvenience. Unfortunately for large text such a buffer would occupy a sizeable portion of memory – even if stored as a monocolour bitmap. A further restriction is that the GDI only allows a certain amount of memory for use as a device context.

3.6 Scrolling Text Implementation

The general algorithm for the initial text scroller application is fairly straightforward. This section gives a short description of the structure and methods used for implementing the scroller using MFC. The application follows the class diagram shown earlier in Figure 3.1. It may help to consult the code samples given in Appendix A.1.

3.6.1 CReaderDoc Initialisation

The *CReaderDoc* loading mechanism extracts strings of up to 256 characters serially into elements of the *CStringArray* member, *m_textStrArray*. The *GetText* member function can then be used to return a string of text from this. In the initial implementation, *m_textStrArray* behaves like a circular stack. Calling *GetText* 'pops' the top string from the stack and 'pushes' it onto the bottom.

3.6.2 CView Initialisation

The *CReaderView::initiateText* function is responsible for retrieving the initial string of text from *CReaderDoc* by calling *GetText*. Using the default font and size attributes, the pixel width of the text string is calculated. This is carried out using *GetCharWidths*, which calculates each individual character width in the current font. These are then used by *GetStringWidth* to determine the pixel length of the entire string, *m_textStrPixelLen*. This is used later in the scrolling process to determine exact pixel locations of characters being displayed.

Setting up the off-screen buffer and rendering text to it is carried out by the *renderText* function. Using the "Scrolling Window" algorithm detailed above, this function can be called in three situations: directly after initialisation; at the end of the each displayed section of text, when new text is loaded; and whenever the screen becomes invalidated. The screen becomes invalidated whenever a window is moved or resized – when the device context of one application interferes with another. Thus the *View's OnSize* and *OnDraw* functions are simply overridden to call *renderText*.

3.6.3 Calling the scroller

The scroller function is called from out with *CReaderView* by the application framework's handle function *OnIdle*. Care must be taken to ensure that the application cannot call this function before *CReaderView* itself has been created and initialised.
The approach of using OnIdle was chosen because initial trials using the system timer ran far too slowly. Ideally, the scrolling mechanism should be regulated in some way to preserve consistency

across different systems. The implementation given in Appendix A.1 attempts to regulate the scrolling loop by using a counter within OnIdle. The counter only allows *scroller* to be called on every *n* counts. This method works well for development purposes, but use of a multimedia timer is recommended for any further work.

3.6.4 Displaying the text

The main thing scroller does is call *displayText*, which handles all of the blitting between off-screen buffer and screen. The *displayText* function blits a window of the current visible screen size (*m_screenSize*) from the larger buffer (*m_bufferSize*) onto the display. See method for “Scrolling Window” in Section 3.5.2. The member *m_pos.x* is used to keep track of the current x-coordinate of the bottom-left point on the source window. Before each call to *displayText*, the scroller increments *m_pos.x* by an integral number of pixels (*m_scrollerIncrement*).

Hence on each call to *displayText* the source window of the buffer is moved along and blitted to screen, thus creating the effect of scrolling text.

3.6.5 Updating the scroll buffer

Once the scroller reaches the end of the current buffer, new text should then be rendered. The *displayText* function checks for when *m_pos.x* reaches the point exactly one visible screen width before the end of a section of text – assuming forward scrolling from left to right. When this is reached, the *m_bRenderText* flag is set, signalling scroller to call *renderText*.

This is where *m_textStrPixelWidth* comes in handy. Function *renderText* uses *m_textStrPixelWidth* to determine at exactly which point in the string the currently visible text begins. It uses this to extract the next section of text beginning *just before* the point identified. This means that the text already on screen is re-rendered at the start of the buffer with new text following directly on from it. Thus allowing a seemingly continuous flow of text. See Figure 3.5.

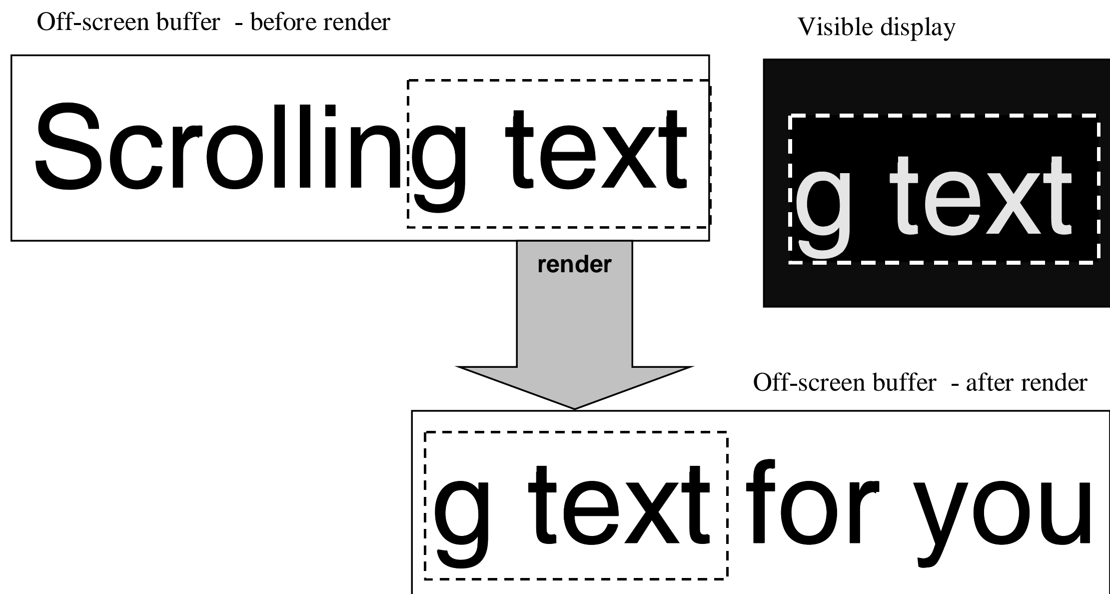

Figure 3.5 Maintaining continuous scrolling text before and after a render

In the initial implementation, no provision has been made for reverse scrolling. This is something that should be rectified for any future release.

3.6.6 Controlling the application

Any changes to the flow of scrolling or to the size and format of the display may be made through both menu and button controls, or via keyboard shortcuts. The handler functions for each user interaction generally make some change to the relevant member variable. For example, pressing the 'up' arrow key increases the scrolling speed by incrementing the variable *m_scrollerIncrement*. Similarly the 'down' key decrements the variable.

Where changes are made to the text format, for example size, font and colour of text, the *renderText* function must be called in addition to changing the relevant variables.

A typical screenshot of the working reader application is shown in Figure 3.6. The executable for the reader is included on the accompanying CD.

Figure 3.6 Screenshot of the initial LVA reader application

3.7 Analysis

For this initial implementation it was decided that the "Scrolling Window" approach gave better results. With a buffer of about seven times the screen width, reasonably smooth scrolling could be obtained. Memory overheads were kept to a minimum by using monocolour bitmaps.

Despite these improvements, the *scroller* continued to halt whenever *renderText* was called. This was noticeable and occurred frequently, particularly with large text. Some further investigation was required to speed up the rendering process and or stop the scroller from being interrupted.

Another problem was the effect of line tearing on the display. The *BitBlt* function runs fairly slowly compared to the refresh rate of Windows itself, and whenever it was called the redraw could be observed 'tearing' on the screen. The effects of this tearing became especially visible when the scrolling speed was increased. This was so visually disturbing that it rendered the application unusable in any practical sense.

3.7.1 Improvements

Other than the operational problems identified above, there is much scope for improvement to this initial implementation. Notably, navigational features such as letter/word/sentence jump and a search facility remain to be implemented. More essentially, backward scrolling should also be dealt with. Additionally the document class should be extended to deal with larger files than just 256 by 256 characters.

From a low vision user's point of view, the use of standard MFC dialogues is not ideal. One future improvement would be to redesign these dialogues to something more appropriate. For example, the text colour selection dialogue could be with fewer but bigger colour selection buttons.

These suggested improvements will not be implemented in the context of this project, but should be noted for any future work.

3.8 Summary

The initial implementation of a scrolling reading aid has been presented. A basic user interface and application framework has been described using a Win32 based platform. Basic graphics techniques such as double buffering and scrolling windows have been investigated. The result of this is an application that fulfils its specification, albeit with two major problems with the particular scrolling implementation. In particular, the effect of line tearing and also the problem of slow rendering times causing the application to halt every few seconds. These shall be investigated in the next chapter.

Chapter 4
Smoothing the Scroller

4.1 Introduction

This chapter attempts to investigate the display problems encountered with the text-scrolling algorithm outlined in Chapter 3. Some possible solutions are discussed and means of implementing them for this application considered. In particular, a specialised graphics class is proposed for interfacing the application to some suitable graphics library. Two popular graphic libraries for the Win32 API are assessed, namely OpenGL and DirectDraw. The implementation of the specialised graphics class using the DirectDraw API is described, as is an improved algorithm for displaying smoothly scrolling text.

4.2 Problem Description

There are two basic problems described in the last chapter. The first is the effect so-called 'line tearing' – the visually disturbing effect observed as the scrolling display is moved across the screen. The second is the effects of rendering overheads – observed as momentary halts in scrolling whenever a new display is being rendered. Both of these problems raise issues with the scrolling text application that are far from trivial. Particularly when implemented using a Win32 GDI based platform.

4.2.1 Line Tearing

The line-tearing problem is essentially one of synchronisation. In particular, the software screens refresh synchronisation with that of the display hardware. Although the Windows operating system itself has its screen refresh in sync with the display hardware, characterised by there being no such tearing observed when a window is moved, the GDI functions do not. This means that whenever a GDI *BitBlt* is called the data is copied to the screen regardless of whether there is a physical screen refresh in progress or not.

Tearing is observed when a new image is blitted to a screen that is mid-way through redrawing the content of the previous image. When displaying a moving image, the succession of partially drawn images creates flickering jagged edges on screen, thus rendering the text uncomfortable, if not impossible to read.

It follows from this description that in order to be able to solve line tearing, the scroller must somehow be synchronised with the physical display refresh. An investigation into possible ways of attacking this problem is detailed later in this chapter. A working solution is also presented.

4.2.2 Rendering Overheads

In any graphical application, before displaying to screen an image must firstly be rendered. The current implementation of the text scroller, using an off-screen buffer with scrolling window, requires an image to be rendered whenever the end of buffer is reached. When implemented in a single program execution sequence (thread) the scrolling display update must temporarily halt while the rendering is carried out. This can cause a noticeable halt in the scrolling, as shown in Figure 4.1.

Figure 4.1 Single thread execution of text scroller application

4.3 Generic Solutions

One approach to this problem is to make the rendering finish as quickly as possible. The faster the rendering can be carried out, the less noticeable the halt in scrolling. Additionally, speeding up the scroller execution would leave more time for rendering. As shown in Figure 4.2 if the blit operation was to return immediately, having registered with the graphics hardware to update the screen, then execution time can be reduced significantly.

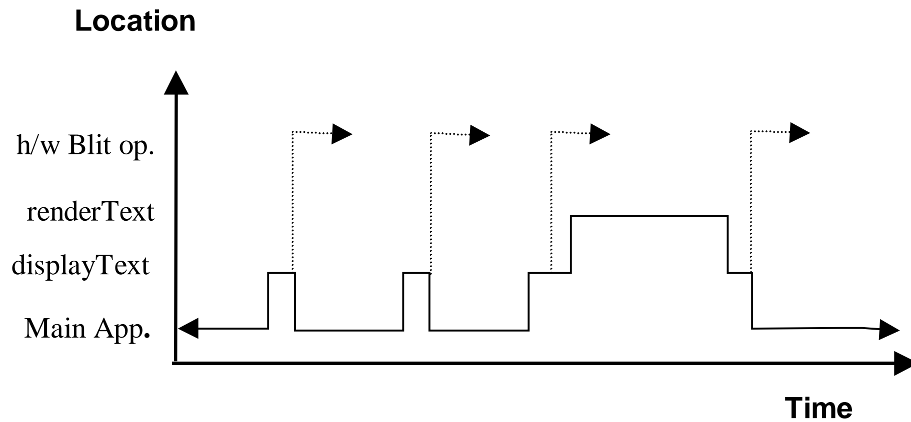

Figure 4.2 Single thread execution of scroller with immediate return blit

A second approach would be to execute the rendering code 'simultaneously' with the scroller. This can be achieved on a single processor PC using a separate thread of execution, as shown in Figure 4.3. The threads effectively enable multiprocessing at a low level, giving the appearance of simultaneous execution. Implementing threads using Java is relatively straightforward[20]. With Win32 however the opposite is true. MFC does provide classes for implementing threads, but using them is non-trivial. A full investigation of threads shall not be carried out in this report due to time restrictions. The implementation given in this chapter does however leave scope for possible future addition of threads.

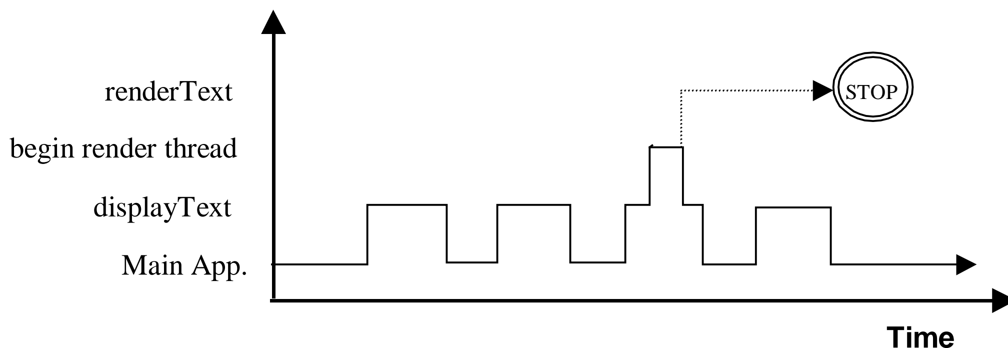

Figure 4.3 Execution of text scroller with rendering performed in a separate thread

4.3.1 Triple Buffering

Although the processor may be freed by diverting blit operations to display hardware, the off-screen buffer would not. Any rendering to this buffer may invalidate the blit in progress. A solution to this is to employ a second off-screen buffer. This can be rendered to while the first is being used by display hardware. When the next blit operation is called, the buffers are rotated –

i.e. one off-screen buffer is handed to display for blitting, while the other is made available for rendering. This technique is known as triple buffering.

4.3.2 Refresh Synchronisation

Moving the blit to hardware does not necessarily imply that the operation will be speeded up. Speeding up the execution of a blit depends both on the display hardware and the number of tasks that blit must perform. Any additional operations, such as magnification or colour translations can significantly slow down the blit.

A one-to-one blit between same size buffers resident in display memory, as opposed to system memory, can increase the speed of operation significantly. This can help to reduce the effects of tearing. Unfortunately it cannot completely remove them. To do this, the screen update initiated by the blit must be fully synchronised with display hardware. Due to its device independent nature, this cannot be achieved using the standard Wind32 GDI. Instead a much closer interface to the hardware is required.

Unfortunately any closer interfaces to hardware comes with the added complexity of having to deal with device specifics. The GDI deals with differences between display devices by interfacing to the hardware through the given device driver interface (DDI) for that device. Often, manufacturers develop their DDI with GDI in mind, so any additional functionality would require bypassing both GDI and the given device interface. This involves dealing with many low level parameters and functions specific to each device. To deal with the multitude of display technologies available, as well as anticipating new ones is a task neigh on impossible. Some mid-way solution is therefore required.

4.4 Investigation of Graphics Libraries

Various graphics libraries are available for providing an improved interface to display hardware. As well as providing lower level blit-style operations, these libraries provide added capabilities for off-screen buffers and can make better usage of advanced graphics hardware. These libraries provide a higher-performance graphics interface, while maintaining at least some of the device independence provided by GDI.

Two desirable characteristics of a suitable graphics library for this application were highlighted as: closer collaboration with hardware for fast screen refresh and blitting; and provision for fast off-screen rendering.

This investigation covers two of the graphics libraries available for Win32. These are OpenGL[21] and DirectDraw[22]. For completeness the Win32 GDI shall also be considered.

4.4.1 GDI

As already shown in the last chapter, the GDI graphics library provides many useful features for 2D graphics manipulation. Although these have proved insufficient for low level display operations, such as fast blitting and display memory buffering, there are many GDI functions that may still be of use to this application.

GDI provides a host of tools for creating and manipulating graphics objects. These objects include brushes, pens, bitmaps, pixels and text. The attributes of these objects, such as fills, thickness, font and colour, can be specified and changed using object member functions.

Off-screen buffering can be provided using the GDI device contexts. Device contexts provide a canvas onto which GDI objects can be assigned and manipulated. As shown in the initial scroller implementation with the *renderText* function (Appendix A.1) these functions provide a convenient way of manipulating text with different system fonts and attributes. Bypassing this capability for lower level code would introduce increased complexity with a huge increase in necessary coding.

4.4.2 OpenGL

OpenGL is an open standard graphics library that may be used on a variety of platforms. Although designed for 3D applications, OpenGL can be tailored for 2D. This platform was chosen for a possible implementation of the scroller’s graphics class due to the wide support available for it. Additionally, its inherent cross-platform appeal meant that any future implementations of the text scroller on another platform would reap the benefits of minimal required alteration.

OpenGL provides support for off-screen buffering using a back buffer. This can be rendered and then copied to screen using the function *SwapBuffers*. *SwapBuffers* itself is not directly an OpenGL function, it is an add-on to the Win32 GDI. It is limited by GDI’s device independence and as such is not directly synchronised with the hardware. It does however provide a much faster transfer to screen than *BitBlt* as it uses the OpenGL back buffer which is optimised for such transfers.

4.4.2.1 Initialising OpenGL

Due to the inherent 3D nature of OpenGL, initialisation for even a basic 2D application is non-trivial. The view is treated as a 3D co-ordinate space and as such requires prior setting up for 2D emulation.

Before rendering can begin, the back buffer must be set up with the correct screen pixel format and other device parameters. Setting up OpenGL to be compatible with the current device must be carried out manually. This involves using *glGet* function to probe the current device for its capabilities and fill them into an appropriate structure for use by the OpenGL engine.

Much of the code for initialising OpenGL for use in this application is irrelevant to the context of this report, but can be found in the accompanying CD.

4.4.2.2 Rendering text with OpenGL

The support for rendering to the back buffer is where OpenGL's strengths lie. Fast manipulation of 3D and 2D polygonal images, texture mapping from bitmaps onto polygons, and fast view translations, rotations and inversions are all possible. The caveat however, is with OpenGL's ability to render text.

The Win32 version of OpenGL available for use in this project did not allow the use of GDI functions directly onto the back buffer. This meant GDI could not be used to format and render text.

To get around this, the OpenGL function *wglUseFontBitmaps* was used. When the application is set up, or whenever the font or text size is changed, this function is called to generate an array of bitmaps. Each bitmap element corresponds to a fully rendered glyph of the desired font.
The process for formatting text in OpenGL is outlined below:

- Prepare GDI font using *CFont* object
- Obtain the main OpenGL device context using *wglGetCurrentDC*[Ψ]
- Select the *CFont* into device context in usual GDI way
- use *wglUseFontBitmaps* specifying the device context to create bitmap array

---

[Ψ] Note the device context returned corresponds to the screen only – it cannot be used as a back buffer

The method employed for rendering is effectively the same as that used in the Alphabet Buffer algorithm described in previous chapter. When a string of text is to be displayed, the desired glyphs from the array are chosen and placed in position onto the back drawing buffer. This is carried out using the function *glCallLists*, specifying the string as a parameter; the function *glTranslate* must also be used to specify the position in 3D co-ordinate space.

Once the drawing has been fully rendered, *SwapBuffers* can be called to transfer the entire drawing buffer onto the screen device context.

4.4.2.3 Scrolling text with OpenGL

In practice, after the desired glyphs are converted into real-size bitmaps and listed as an array of pointers, accessing them and rendering to buffer became quite slow. Swapping the buffer to screen could be carried out quickly, but there is no easy way of translating (horizontally shifting) a selection of the drawing buffer onto screen without completely redrawing the buffer again. The resultant scrolling was very slow and jumpy.

As the *SwapBuffers* function provided no facility to specify a source window, or even to translate the view, the Scrolling Windows approach was not possible for this implementation.

If some OpenGL ‘tricks’ are employed however, the process of translating the display quickly during scrolling can be partially realised. One idea is to alter the OpenGL view co-ordinates on each scroller increment, as opposed to the drawing raster position. Although a simple enough idea, this proved tricky to implement. The bitmaps must still be redrawn on each scroll step, but as the parameters to *glTranslate* and *glCallLists* do not change the redraw should not take much time. In practice however, the speed improvements were slight and the scroller continued to run very slowly.

4.4.2.4 Analysis of OpenGL

The OpenGL implementation failed to solve the problems of either line tearing or rendering speed. Surprisingly, the rendering time for a string of text was even slower than using GDI alone. The likely cause of this is not with OpenGL itself, but with the limitations imposed by the particular Win32 implementation of OpenGL used.

It is likely that a better solution could be found using a later version of the OpenGL library. This is something that may be considered for any future work.

4.4.3 DirectDraw

DirectDraw, from Microsoft's DirectX software development family, provides a more optimistic route. As with OpenGL, DirectDraw can be made cross-platform. However, as its name suggests DirectDraw provides a more direct interface to hardware. In particular it can be configured to make effective use of device memory, and is able to synchronise screen refresh with hardware. Another advantage of DirectDraw API is that it can be used alongside GDI. Figure 4.4 shows the relationship between the DirectDraw hardware emulation layer (HEL) and the device driver interface (DDI) compared with GDI.

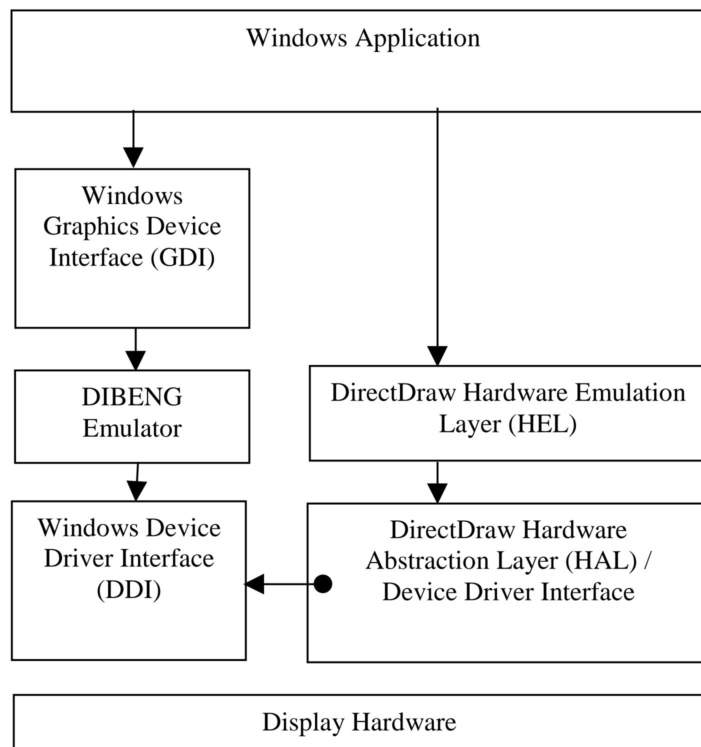

Figure 4.4 Video architecture for Windows 95 – indicating both GDI and DD interfaces between applications and hardware

4.4.3.1 Surfaces and Double Buffering

Access to video memory in DirectDraw is provided through surface objects. As there are many different kinds of video memory, so there also many kinds of surfaces available through DirectDraw.

A primary surface gives access to the main display memory. Anything written to the primary surface is immediately visible on screen. Unlike the GDI device context, a primary surface

includes the entire screen. This means that care must be taken when writing to it so as to avoid overwriting the display space of other applications.

Buffering can be achieved in DirectDraw using one or more off-screen surfaces. Surfaces can be manipulated directly as a contiguous section of memory. Alternatively, a handle to the surface can be obtained using a GDI compatible device context. This allows GDI graphics rendering functions to be used.

4.4.3.2 Fast Blitting

The DirectDraw API provides a *Blt* function for blitting between surfaces. A DDBLT_WAIT flag can be set to specify whether the *Blt* function should wait until the blit operation has been completed before returning. If *Blt* is called without this flag, it will post the blit operation to display hardware and return immediately.

The blit operation itself can be carried out considerably faster than with GDI if all surfaces involved are held in display memory. This is one of the ways in which DirectDraw makes good use of the hardware resources available.

4.4.3.3 Page Flipping

Using *Blt* alone does not ensure synchronisation with hardware. To achieve this DirectDraw must be set up in page flipping mode.

Page flipping uses a number of surfaces, all with the same size and format. One surface is assigned as a front buffer, the rest as back buffers. Whenever a *Flip* command is called, one of the back buffer surfaces becomes the front buffer and the surface acting as front buffer is flipped into a back buffer. Note that it is the pointers to the surfaces that are copied and not the contents. Additionally, the *Flip* operation waits until just before the hardware begins refreshing the screen before flipping the buffers. The effect is an almost instantaneous display refresh, without any tearing.

While waiting for a *Flip* operation, the back buffer surface that is next for flipping to front buffer is locked. This means that no rendering can be performed on it. One way round this is to introduce two or more back buffers. See Figure 4.5. Triple buffering, as described in section 4.3.1 can be used to great effect in this case. The only practical limit to the frame refresh rate is the refresh rate of the monitor.

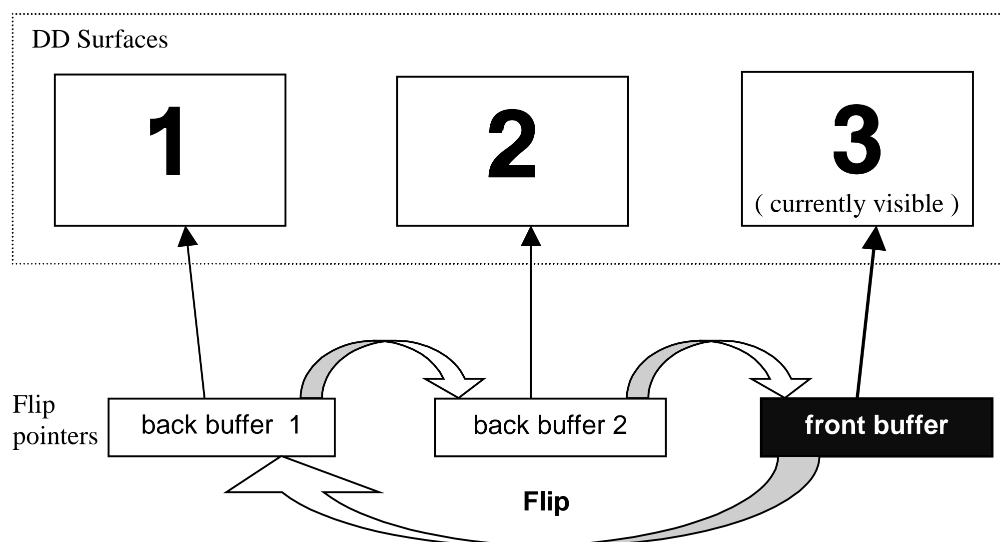

Figure 4.5 Page flipping with three surfaces in DirectDraw

4.4.3.4 Caveat: full-screen mode

Unfortunately page flipping can only be used in full-screen mode. This is because Windows itself does not use Page Flipping. Any application making use of Page Flipping therefore becomes responsible for the entire display. Windows GDI does not recognise the back buffers used in Page Flipping and as such is unable to maintain a useable windows interface. Any user interaction or windowing features must be dealt with by the application itself.

Another problem with page flipping is that only entire surfaces can be flipped. This means that sub windows of a surface cannot be singled out for display. This problem can be overcome, but at the expense of an extra (non-flip) surface and *Blt* operation.

4.4.3.5 Analysis of DirectDraw

DirectDraw provides a far superior API for the scrolling text application than either GDI alone or OpenGL. The speed increase of *Blt* and the provision for multiple off-screen surfaces in display memory is ideal for fast display refresh. The ability to use existing GDI rendering functions is also beneficial from the point of view of development time.

The line-tearing problem can be eliminated in DirectDraw using Page Flipping, albeit at a price of using full-screen mode. With the view of this particular application as a low vision aid, this price may not be so terrible however. It is accepted that the standard Windows interface is not entirely suited to the needs of low vision users. Having other windows and standard menu bars on screen may serve to detract from the low vision users ability to concentrate on the main tasks, i.e. reading

text. With full-screen operation, the display can be optimised for reading text, with a minimal set of interface features geared towards low vision use.

4.5 Revised Text Scroller

A revised scrolling text application was implemented using the DirectDraw graphics API. The scrolling algorithm employed is Scrolling Windows, as described in the previous chapter. Most of the changes to the *CReaderView* class are in relation to switching from using GDI to a new class, *DDGraphics* that implements the main graphic features required for this application.

4.5.1 The DDGraphics class

A new class was defined specifically for providing a basic interface to the DirectDraw library. The responsibilities for this graphics class are given in Table 4.1:

Abstract Class name: **DDGraphics**	Collaborator(s):
Responsibility: • Initialise display interface • Create off-screen buffers • Means of drawing to buffers • Blitting between buffers and screen • Releasing display interface	[classes from graphics library]

Table 4.1 DDGraphics class responsibilities and collaborators

A full listing of the *DDGraphics* class code can be found in Appendix A.3. The revised class diagram for the application is shown in Figure 4.6. The following sections in this chapter deal with the operation and methods of this class.

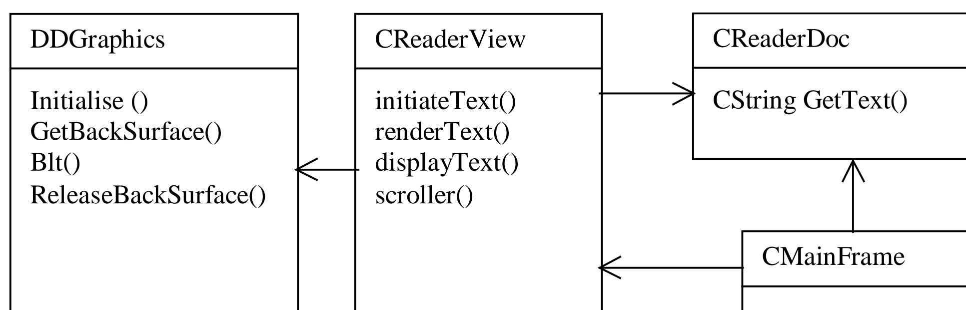

Figure 4.6 Simplified UML class diagram including DDGraphics

4.5.2 The DDGraphics back buffer

The *DDGraphics* class emulates a wide off-screen buffer by using three separate off-screen surfaces. These surfaces are related using a form of triple buffering and are exactly the same size as the screen.

Only one surface can be written to at a time; that is function *GetBackSurfaceDC* only returns the device context of a single surface, *m_lpBackSurf*[Ψ]. The other two off-screen surfaces cannot be written to directly, but can be blitted to the primary surface for display; these are *m_lpBltSurf1* and *m_lpBltSurf2*. See Figure 4.7.

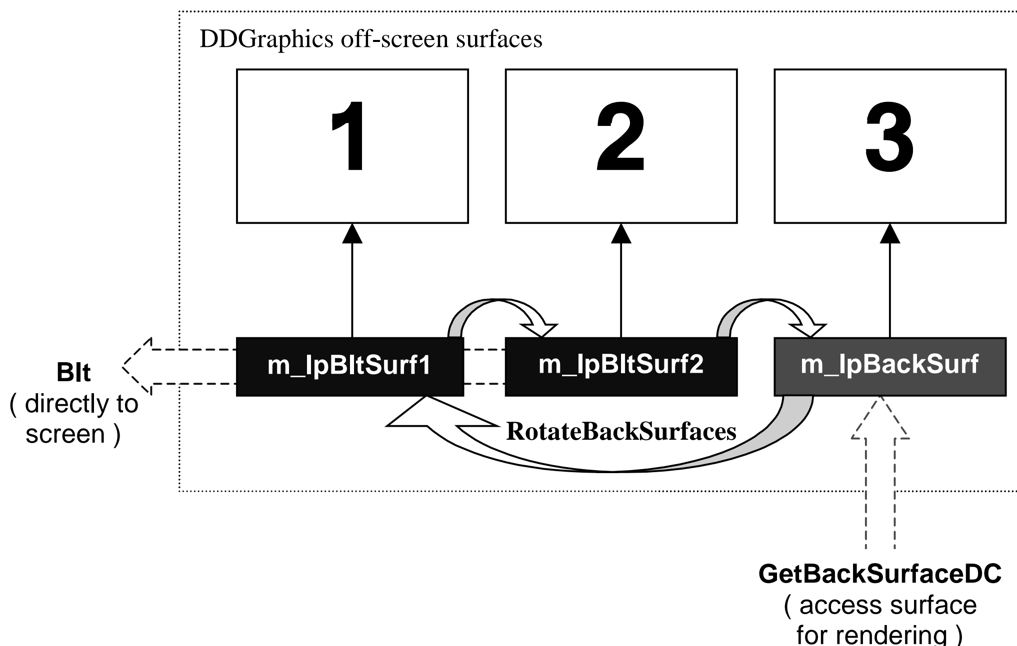

Figure 4.7 Wide buffer emulation using three surfaces in *DDGraphics*

The pointers to these three surfaces, each determining that surfaces particular role, are stored in a circular stack. The function *RotateBackSurfaces* rotates these pointers, thus changing the roles of the surfaces.

The advantage of this method for off-screen buffering is that a large buffer can be used; three (or more) times the width of a single surface, while only needing to render a single surface at a time. Rendering to a single surface takes less time than to the whole buffer. This method also leaves open the possibility for pipelining using separate threads of execution. One surface could be

[Ψ] of type: *LPDIRECTDRAWSURFACE*

rendered to in a separate thread, while the others are used for the scrolling display. This would effectively eliminate the rendering overheads.

4.5.2.1 DDGraphics::Blt

The *Blt* operation takes two parameters: *rDst* the destination rectangle on screen and *rSrc* the source rectangle. To the outside world the *Blt* function treats the two surfaces *m_lpBltSurf1* and *m_lpBltSurf2* as a single contiguous surface. Within the function however, two separate DirectDraw blitters are used, one per surface.

Each blitter has its own variable sized source and destination rectangles: *rSrc1* and *rDst1* for *m_lpBltSurf1*; *rSrc2* and *rDst2* for *m_lpBltSurf2*. The function splits the original source and destination windows between these rectangles. These are then scrolled across the two surfaces in such a way as to give the impression of a single surface source. The diagram in Figure 4.8 may help to illustrate this.

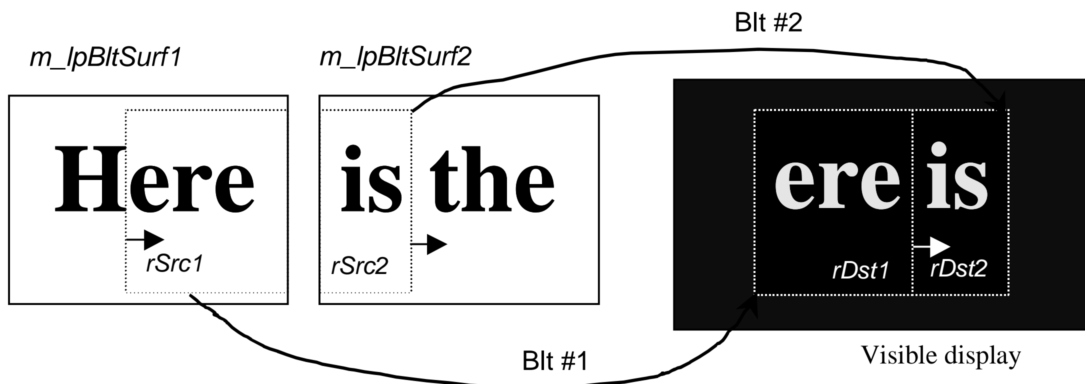

Figure 4.8 Blitting from two surfaces showing split source and destination scrolling windows

4.5.2.2 DDGraphics::Initialise

This method is called when the scroller initialises its *View*. It is responsible for setting up the DirectDraw interface and surfaces. It uses the DirectDraw *GetCaps* function to determine the capabilities of the system; in particular if the device has support for fast hardware blitting. Ideally the application should have redundant code for dealing with systems that do not support hardware blits. In this version however, full support is assumed. This is a reasonable assumption bearing in mind that nearly all new PC’s come equipped with some form of display hardware acceleration.

It then goes on to create a pointer to the primary surface *m_lpPrimSurf*, linking it to the current display and setting it up with the correct system palette. Various attributes of the surface, such as size and memory type[ψ] are set using the surface description structure[ζ], *m_sdPrimary*. The *m_sdPrimary* structure is also set with the flag *DDSCAPS_PRIMARYSURFACE*, to denote the role of this surface.

Note that in order to behave as a good windows application, the primary surface should be clipped to only include the area of screen allocated by the window manager. This is achieved by passing a reference to the *View's* client window to a DirectDraw *Clipper*[ξ] object. This is then attached to primary surface. Whenever the window size is changed, DirectDraw automatically updates the surface.

The three off-screen surfaces are then created: *lpBackSurf*, *m_lpBltSurf1* and *m_lpBltSurf2*. These are created in a similar manner to the primary surface. Instead of the flag *DDSCAPS_PRIMARYSURFACE* however, *DDSCAPS_OFFSCREENPLAIN* is used.

To maintain good compatibility across different display devices, the size of the surfaces should be set to exactly the screen size. This can be determined at run time using the GDI function *GetSysCaps*.

4.5.2.3 DDGraphics::Flip

The DDGraphics code for creating flippable surfaces and for flipping them is included in Appendix A.3. The method of Page Flipping is described in Section 4.4.3.3. However, due to time constraints Page Flipping has not been fully implemented for this application.

[ψ] System or display memory
[ζ] Of type: *DDSURFACEDESC*
[ξ] Of type: *LPDIRECTDRAWCLIPPER*

4.5.3 The CreaderView Class

Much of the *CReaderView* code structure remains essentially the same as before. Most of the changes are minor - merely a switch from GDI function calls to the relevant *DDGraphics* calls. The main code for this class is given in Appendix A.3.

4.5.3.1 CReaderView::renderText

Instead of using GDI *GetDC*, *renderText* uses *DDGraphics::GetBackSurfaceDC* to obtain the device context. This can be rendered to in the same way as before.

4.5.3.2 CReaderView::displayText

There are a few minor changes to the operation of *displayText*. The source and destination 'windows' used for scrolling operate in a similar manner as before. Instead of GDI *BitBlt*, the *DDGraphics* function *Blt* is used.

4.6 Analysis

At first glance, the revised scrolling text algorithm isn't much better than the one described in the previous chapter. For one thing the line-tearing problem remains. Additionally, the rendering overheads causing the scroller to halt also remain. It would seem that the extra effort involved in developing the *DDGraphics* class was wasted.

Despite the initial disheartening appearance, much progress has actually been made. For one thing the time constraints of the project meant that neither Page Flipping nor implementation of a separate rendering thread could be completed. A future incorporation of these would almost certainly produce results that are more impressive.

The implementation as it stands does incorporate some noticeable improvements over the initial GDI approach. The DirectDraw fast *Blt* operation and usage of display memory means that despite a lack of device synchronisation, the display refresh is much quicker than before. On some systems, the effects of line tearing are almost completely negated.

The implementation of the off-screen buffer using surfaces in display memory also means that the rendering time is much quicker than with GDI. The triple buffering technique employed also leaves room for future implementation of a separate rendering thread.

4.7 Chapter Summary

A revised implementation of the scrolling text LVA application has been presented. The problems experienced in previous implementations, in particular line tearing and slow rendering time, still remain albeit to a much lesser extent.

A structure is in place however for using the DirectDraw API to eliminate these problems altogether. An outline of how any future revision of the application could address these problems more effectively has been given. The only reason they have not yet been implemented is purely down to time constraints.

Chapter 5
Feedback

5.1 Introduction

In this short chapter, relevant feedback from the use of a similar LVA program is discussed. Some possible improvements to any future development of the application are suggested.

5.2 A recent development

Since this project's inception, the existence of a similar text scrolling LVA has come to light. The Supernova package by Dolphin software incorporates a host of standard PC CCTV features such as screen magnification, text to speech, as well as facilities for low vision web browsing[18]. More importantly, it incorporates a full screen text scroller application, almost identical to the design described in Chapter 3. Fortunately, from the point of view of this project, the Supernova text scroller also suffers from exactly the same problems as the design in Chapter 3. Notably the problems of line tearing and jerky scrolling. From this it can be gathered that these are common problems, and in the case of Dolphin software, as yet unsolved. This served to enhance motivation for the work described in Chapter 4.

Some features of the Supernova package, in particular the interface for text navigation and the text to speech facility, may be of further use in influencing future design considerations of this project. Also in light of the similarity of the two programs, a preliminary analysis was carried out on the Supernova scroller by talking to a low vision user familiar with the system. The interview was conducted with assistance from the Edinburgh District Library resource centre.

5.3 Feedback on Supernova

In practice the text scrolling facility of Supernova is rarely used. The main reason for this is that it is regarded as nothing more than a gimmick with only short term appeal. This is primarily due to the uneven scrolling problems (as identified with the design in Chapter 3). The text can still be read, but the line tearing and jerky text can cause annoyance after a short time using the program.

Aside from this problem, the text scroller is generally regarded as potentially a good idea. Some points about its operation are noted as follows:

- Easy to use control keys for navigating text
- Run-time adjustable text attributes – size, speed, colour
- Full screen text removes ‘Windows’ distraction
- Combined scroller and text to speech reading.

5.3.1 Interface and control

Most interactions with the Supernova text scroller are performed via keyboard. Navigating the text is performed using the left/right cursor keys, with speed and direction of scrolling using up/down arrows. Various key combinations are specified for changing attributes of the text. In particular, the text size can be changed using the +/- keys. This is a very useful feature allowing the user to ‘zoom’ in and out of the text being read. Likewise different colour combinations can be tried using the colour toggle key-press.

Once learned, the interface is very easy to use and allows fast navigation of text during reading. The problem lies with first-time users trying to learn the interface. Those familiar with computers generally have no problem, but much work is required to design an interface that is ‘friendly’ to non-computer users. Further simplification of control may be required to do this. One suggestion may be to employ a joystick or control pad style interface as found on many games machines.

5.3.2 Full screen

A feature of the program that is particularly appealing to low vision users is the lack of any visual controls such as windows, icons, menus or pointers (WIMP.) Such controls are often no more than a distraction to low vision users who generally cannot see them properly anyway. Clearing the screen of clutter allows more space for the magnified text to be displayed.

5.3.3 Text scrolling with speech

Experience of aids for both day-to-day computer use and for reading shows that a combination of magnified display and speech together produces the best results for low vision users. Text can be read much quicker when using a text-to-speech program while manipulating the interface is achieved best by a combination of visual and audio feedback.

The problem with using the text-to-speech facility however is more a psychological one. Low vision users often have the preconceived notion that text-to-speech readers are for completely blind people. They feel embarrassed having to use an audio reader. Following from this, they try to ‘make the most’ of what sight they have and opt for the magnified display solutions only, even if it takes longer to use.

5.4 Further recommendation

In addition to fixing the scrolling problem, some further recommendations for a future LVA were highlighted, these were: the addition of a search/find facility and screen hooks.

A simple to use word search/find facility was recommended over the use of word/sentence/paragraph jumps as means of quick navigation of text. Screen hooks were also recommended. These are portions of the screen that show some other part of the text that may or may not be chosen as a jump location based on a search result. These would also help the user to keep track of, or bookmark, their position in the text being read.

5.5 Summary

A description of the pros and cons of the Supernova LVA text reader was given. Additionally, some recommendations relevant to future development of a software LVA were identified.

Chapter 6
Summary and Discussion

6.1 Project Summary

The main aim of the project has been partially achieved. A software based low vision aid (LVA) for reading has been developed and a prototype built. Unfortunately due to unforeseen complications in the implementation phase, the final version of the LVA has not yet been finished. The prototype version works and it does conform to the initial specification. However its operation is not of an acceptable standard for release. The problems encountered in this initial version have been extensively researched and a design for a suitable solution presented in this report. A brief summary of what has been achieved is given below:

- Background research into problem of low vision and low vision aids.
- Preliminary specification for a software based LVA
- Initial working version of the LVA designed and built
- Major problems of scrolling smoothness identified and variety of solutions sought
- Similar LVA software discovered with similar problems, useful feedback provided
- Re-design of the LVA incorporating improvements – only partially implemented
- Foundations laid for future trials of the software

6.2 Still to do

Most of the work involved with implementing the final application has already been completed, in coding time the remaining work should take no more than a matter of weeks to finish off. This includes:

- Finish code implementing DirectDraw page flipping
- Implement the rendering functions in a separate threaded class
- Finalise interface for backwards and forward scrolling with possible search facility

A possible future improvement for longer term usage of the program would be to time the scroll rate from a multimedia timer instead of from within the processors idle loop.

6.3 Should have done

One area of the project that could have been improved is in the approach to the initial design. Most of the design for the application was carried out on paper. As such, much of the initial focus was on research into the interface and following the other objectives of the mission statement. It was assumed that development of a text scroller would be trivial – this turned out to be a costly assumption!

The scrolling problems could not have been anticipated given the circumstances and resources available. If implementation of the scroller had been given priority, the problems would almost certainly have been discovered sooner. This may have allowed more time to produce a final working version.

6.4 Future Work

There is much scope for future work leading on from this project. The field of research into low vision aids using modern day technology is very much in its infancy. The basic reading aid developed in this project may be adapted for further research into various aspects of reading and low vision computer use. Additionally, the reading aid could be extended by a more suitable control interface and text capture mechanism. Moving away from the confines of the PC interface may result in a far more widely useable aid. Eventually there may even be scope for an aid that assists low vision sufferers with more than just reading tasks.

6.4.1 Medical trials

To assess the effectiveness of the software LVA extensive medical trials are required involving participation from low vision sufferers.

Measuring the effectiveness of the LVA as a reading aid may best be achieved by performing a time based trial on the software LVA versus another aid familiar to the user such as the monocular. It is important that the user is given time beforehand to familiarise themselves with the reading interface. Additionally the default size, colour and font of the text should be set to whatever preference the user finds easiest to read, and or pleasing to the eye.

There are many factors affecting such trials that may require investigation. Some of the following in particular:

- Screen – the size and type of display are important. LCD may be preferable to CRT. A large screen would also be an advantage.
- Colours used – are they merely aesthetic user preferences, or do they help with reading?
- Good contrast – a known factor in improving acuity, but how well can this be achieved on a CRT or LCD display? Some studies indicate that white text on black background is most effective for CRT screens, whereas the reverse is true for paper and LCD[23].
- Font – which are clearest to read when text is scrolling? Traditionally serif fonts ('tails' on the letters) can aid word recognition and hence reading flow. However, does this apply to magnified text?

Environmental conditions may also play an important part on the usefulness of the aid:

- Room lighting conditions – does the user perform better in a dark room, or a brightly lit one? Is there a difference between the effects of fluorescent or normal lighting?
- Glare on the CRT/LCD – this is related to lighting conditions, generally flat-screen LCD displays do not exhibit much glare compared to CRT.
- Users blink rate – not so much an environmental factor, but something that may well be worth monitoring. One of the effects of reading from a computer screen is often a reduced blink rate. The effects of this on reading speed is something that should be investigated. One experiment would be to note any differences the application of eye drops may have on reading speed using the computer based LVA.

Another factor that would be worth monitoring is the number of times a user must go back and re-read certain words. Similarly the times a user changes the size of the text being read, either to clarify individual letters in a word or to view the word in context of its neighbours. These may provide clues towards a more effective method of displaying text for low vision readers.

6.4.2 Adaptive scrolling

One possible area of future research could be into ways of changing the speed of text scrolling based on reading strategies. A basic trial of this adaptive scrolling could be to slow the scrolling speed whenever some punctuation, such as commas and full stops are reached in a sentence. Further analysis of reading strategies in low vision sufferers may indicate other ways in which the scrolling could be adapted automatically to minimise user effort when reading.

One example of this would be to have a semi-intelligent model of the user based on a combination of previous user interactions together with specific traits of his/her particular eye defect. This can then be used to identify patterns in the user's reading strategy and allow the system to emulate them. This may or may not provide a useful aid to reading: there is a danger here that the system may become a nuisance by attempting to half guess the reader's next action.

6.4.3 Control methods

Although for initial trials using a keyboard to control the LVA would be sufficient, future work into a more suitable input device is required. Consideration should be taken for readers with physical disability. Ideally the controls should not require any prolonged actions, such as holding down a button for any length of time. Research is required into a control that minimises physical effort and strain, while maintaining a good level of control and feedback between text and reader. Additionally, the control should seem 'familiar' to use, with intuitive correlation between physical actions and manipulation of text on screen.

One solution may be to adopt a joystick or game style control pad. The axis of the joystick can be used to control speed, direction and size of the text, while the 'fire' button used for starting and stopping the scrolling. Care must be taken to choose the right balance between level of customisation of the LVA and simplicity of the interface.

Another, albeit more complicated to implement solution may be to use eye-tracking. EOG (Electro-oculographic) sensors, as used in controlling some military head-up displays (HUD), can track the position of the eye and allow the user to control an interface using eye motion[24]. A simple handheld push button could additionally be used for start/stop operations. Far fetched as this may seem, eye-tracking may provide a very effective means of reading control, although a thorough analysis of individual reading methods would be a necessary precursor it this was to be used in practice.

6.4.4 Capture methods

Further work may be carried out into the specific text capture methods mentioned in Chapter 2, such as with paper scanners, digital cameras and the internet. In particular a means of integrating the text scanning and capture devices into the aid itself so that their usage is simplified and convenient. One futuristic example would be a real-time video camera that is 'trained' to track and extract text from its view. The text could then be fed with minimal effort into the LVA display.

Already there are some major book publishers looking towards releasing texts on the internet. This trend is set to continue, and with it the establishment of standards for paying and downloading new titles. One area of future work would be to look at these standards and text formats for conversion into a form suitable for low vision reading.

Similarly there may be demand for integrating a low vision aid into web browsers. One existing means of achieving this is through programs such as the BBC web site's 'Betsie'[25]. Betsie allows extraction of plain text and hypertext links from any format of HTML file, from this the text can be adapted for low vision reading.

6.4.5 Into the future

A culmination of all the possible work identified above may lead to more than just an aid for reading. As mentioned in Chapter 2, digital video camera and display technology is continually shrinking in size. Using existing wearable computer technology – for example high resolution portable video goggles, digital camera and miniature PC embedded into clothing – an aid could be built that would assist a low vision user in day to day life.

The camera(s) could be used as an electronic eye, the images relayed and enhanced by the PC to the video goggles. Features such as automatic focus and zoom would all be available to enhance the view. Additionally, any text that needed to be read could be captured and enlarged (and possibly scrolled) across the display for ease of reading.

Much of the technology for this kind of augmented reality is already available, and low vision aid such as this would be quite possible to build. However the success rate of such a device would be questionable. Although the hardware may be sufficient, as yet there is much work required on both the software and the understanding of human visual system. It is in the area of *how* to present information to a defective eye, whether it be text or video, that more research is needed.

For this to succeed there must be closer collaboration between the medics who research problems of low vision and the engineers who develop the technology.

Acknowledgements

I would like to thank my project supervisor Dr. David Renshaw for his inspiration and support throughout this enjoyable project. I would also like to thank staff and patients at PAEP for providing me with invaluable help and materials on all the medical stuff, in particular Dr. B. Dhillon and Dr. Jeff Mason. Some others I would like to thank are: Ian Stevenson at ST Vision[26], for bailing me out with DirectDraw; Francis Millingen for suggestions on OpenGL[27]; all the people in the lab and newsgroups who helped me get MFC scrolling; and 'Jim' from the library resource centre, for providing me with some insight into reading with low vision. Finally I thank all those who had scrolling text coming out of their ears through patiently listening to my occasional ranting and tales of coding despair.

References

1. FILNER, P.: Head of research, Macular Degeneration Foundation Website, various links on MD, http://www.eyesight.org, 12 October 1999.
2. Macular Degeneration Support Website, various MD articles and links, http://members.aol.com/danrob/MDpeople/index.htm, 16 October 1999.
3. Centre for Macular Degeneration, Iowa University Website, http://www.opth.uiowa.edu/CMD/Default.htm, 16 February 2000.
4. Macular Degeneration Bulletin Board, Updates on the latest in MD research, http://www.westmass.com/macular/bulletin.htm, 16 October 1999.
5. DIX, FINLAY, ABOWD, BEALE: "Human Computer Interaction", Prentice Hall Europe, 2nd Edition, 1998, p48.
6. DHILLON, B.; MASON, J.: Personal Correspondence, Princess Alexandra Eye Pavillion, Edinburgh, 2 November 1999.
7. Edinburgh District Libraries Resource Centre: Personal Correspondence, Central Library, George Fourth Bridge, Edinburgh, 13 March 2000.
8. TRAUZETTEL-KLOSINSKI, S.: "*Eccentric fixation with hemianopic field defects*", Journal of Neuro-opthalmology, Volume 18, No 3, February 1997, pp 117-131.
9. HARPER, CULHAM, DICKINSON: "*Head mounted video magnification devices for low vision rehabilitation: a comparison with existing technology",* British Journal of Opthalmology, Volume 83, 1999, pp497-498.
10. SAFRAN, DURET, ISSENHUTH, MERMOUD: "*Full text reading with a central scotoma: pseudo regressions and pseudo line losses*", British Journal of Opthalmology, Volume 85, No 12, December 1999, pp1341-1342
11. HARPER, CULHAM, DICKINSON: "*Head mounted video magnification devices for low vision rehabilitation: a comparison with existing technology",* British Journal of Opthalmology, Volume 83, 1999, pp495-500.
12. Pictures of "Jordy CCTV" courtesy of Enhanced Vision Website, www.enhancedvision.com, May 2000
13. "*Supernova magnifier for Windows*", Dolphin Software Website, http://www.dolphinusa.com, 9 February 2000.
14. "L&H RealSpeak Text-to-Speech Reader", Lernout & Hauspie Website, http://www.lhsl.com/realspeak/, 3 May 2000.

15. "*Jaws magnifier for Windows*", Henter-Joyce Website, http://www.hj.com, 9 February 2000.

16. "*inLarge magnifier*" and "*outSpoken text to speech reader*", http://www.alva.com, 9 February 2000.

17. "*Sony Glasstron*", Sony Website, http://www.ita.sel.sony.com/products/av/glasstron/",24 October 1999.

18. SCHWATZ, S.J.: "*Wearable Computing: enhance user's movement throughout the real world.*", MIT Website, http://schwatz.www.mit.edu/people/schwatz/index.htm, 10 February 2000.

19. BOOCH, JACOBSON, RUMBAUGH: "*UML Notation Guide*" Version 1.1, OMG ad/97-08-05 1997.

20. OAKS, WONG: "Java Threads", O'Reilly, Nutshell Handbook, 1997, Chapter 1, pp1-10

21. WOO, NEIDER, DAVIS, SHREINER: "*Open GL Programming Guide*", Addison Wesley, 3rd Edition, 1998, pp289-304.

22. COELHO, HAWASH: "*DirectX, RDX, RSX, and MMX Technology*", Addison Wesley Developers Press, 1998, pp 1-54.

23. SILVER, GILL, SHARVILLE, SLATER, MARTIN: "A new f*ont for digital television subtitles*", Tiresius Consortium Website, http://www.eyecue.co.uk/tiresias/design.htm, 1997.

24. KOSKO, B.: "*Gaze direction determined by EOG*", Scientific American, July 1993.

25. MYERS, W.:"Betsie web page to text-only script", BBC Website, http://www.bbc.co.uk/education/betsie/download.html, 14 March 2000.

26. VAN MILLINGEN, F.: Personal Correspondence, Computer Services Dept., The University of Edinburgh, 29 February 2000.

27. STEVENSON, I.: Personal Correspondence, ST Vision, Edinburgh, 1 March 2000.

Appendix A.1

Initial Scroller Code

CReaderView

Note that m_bufferSize is initialised to approximately 7 times the width of m_screenSize.

```
void CReaderView::renderText(CClientDC *dc, CDC *dcMem)
{
	// empty old bitmap for re-use
	if (bmText.m_hObject != NULL)
		bmText.DeleteObject();

	// make bitmap compatible for buffer
	int BitsPixel = GetDeviceCaps( HDC(*dc), BITSPIXEL );
	int Planes = GetDeviceCaps( HDC(*dc), PLANES );
	bmText.CreateBitmap(
			m_bufferSize.cx, m_bufferSize.cy,
			Planes, BitsPixel, NULL
			);

	// load bitmap into buffer
	CBitmap* pOldBitmap = dcMem->SelectObject( &bmText );

	// create a rectangle in the background color
	CRect rect( 0, 0, m_bufferSize.cx, m_bufferSize.cy );
	CBrush brBackground( m_backColour );

	// setup the background & text attributes
	dcMem->SetBkColor( m_backColour );
	dcMem->SetTextColor( m_textColour );
	dcMem->SetTextAlign( TA_LEFT );
	// erase existing buffer content
	dcMem->FillRect( rect, &brBackground );

	// select font into buffer
	Font font;
	font.CreateFontIndirect( &m_logFont );
	dcMem->SelectObject( &font );

	// render text into position on buffer
	dcMem->TextOut( m_bufferPos.x, m_bufferPos.y, m_textStrAll );

	// save new bitmap out of current buffer
	dcMem->SelectObject(&pOldBitmap);
	m_bRenderText = false;
}
```

```
void CReaderView::displayText(CClientDC* dc, CDC* dcMem)
{
	// check if one screen size away from end of buffer
	if ( m_pos.x > m_bufferSize.cx - m_screenSize.cx )
	{
		// re-position rendering position of text in buffer
		m_buffPos.x -= m_pos.x;
		m_pos.x = 0;
		m_bRenderText = true;
	}

	// check if approaching end of text
	if ( m_pos.x + abs(m_buffPos.x) > m_textStrPixelLen )
	{
		// load in new string for rendering
		initiateText();
		m_bRenderText = true;
	}

	// blit a portion of buffer to position on screen
	dc->StretchBlt(
			0,0, m_screenSize.cx, m_screenSize.cy, dcMem, m_pos.x,
			m_pos.y, m_screenSize.cx, m_screenSize.cy, SRCCOPY
			);

	// format of blit parameters:
	// ( dest co-ordinates, dest size, buffer,
	//   source co-ordinates,   source size,
	//   blit code )

}

void CReaderView::scroller()
{
	CClientDC dc(this);

	// create a device context in memory – the buffer
	CDC dcMem;
	dcMem.CreateCompatibleDC(&dc);

	// load saved bitmap into buffer
	CBitmap * pOldBitmap = dcMem.SelectObject(&bmText);

	// increment position
	if (!m_pause) m_pos.x += m_scrollIncrement;

	// display buffer onto screen at new position
	displayText(&dc, &dcMem);

	if ( m_bRenderText )
		renderText(&dc, &dcMem);

	// save bitmap out from buffer
	dcMem.SelectObject( pOldBitmap );
	// free memory
	dcMem.DeleteDC();
}
```

```
void CReaderView::initiateText ()
{
	// reset position to origin on screen and in buffer
	m_pos.x = 0;		m_pos.y = 0;
	m_bufferPos.x = 0;	m_bufferPos.y = 0;
	m_textStrPixelLen = 0;

	// retrieve text string from document
	CReaderDoc* pDoc = GetDocument();
	m_textStrAll = pDoc->GetString();

	// get pixel width of new string
	m_textStrPixelLen = GetStringWidth( m_textStrAll );
}

void CReaderView::OnSetTextFont()
{
	// call MFC's font selection dialogue
	CFontDialog dlgFont(&m_logFont);

	// initiate dialogue as modal pop-up box
	if (dlgFont.DoModal() == IDOK)
	{
		dlgFont.GetCurrentFont(&m_logFont);
		m_bRenderText = true;
	}
}

void CReaderView::OnInitialUpdate()
{
	// retrieve new string from document
	initiateText();
	// initiate render
	m_bRenderText = true;
}

void CReaderView::GetAllCharWidths()
{
	// obtain the device context - similar to  GetDC
	CClientDC dc(this);

	// retrieve the current font in use from LOGFONT
	CFont font;
	font.CreateFontIndirect( &m_logFont );

	// retrieve widths of all font characters
	dc.SelectObject( &font );
	GetCharWidth( HDC(dc), 0, 255, &*m_lpAllCharWidths );
}

int CReaderView::GetStringWidth( CString str )
{
	int strLen = 0;
	GetAllCharWidths();

	// sum widths of characters in string
	for (int i = 0; i < str.GetLength(); i++)
		strLen += m_lpAllCharWidths[ str[i] ];

	return strLen;
}
```

CReaderDoc

```
CString CReaderDoc::GetString()
{
	// get next string from array
	m_textStr = m_textStrArray[m_textStrIndex++];
	if  (m_endText < m_textStrIndex) m_textStrIndex = 0;

	return m_textStr;
}
```

CReaderApp

This is the application class that creates the View, Document and Frame.

```
// regulate the rate at which the View's scroller function is called
// within OnIdle - between 1000 (fastest) and 1 (slowest)
int CReaderApp::setSpeed(unsigned int speed)
{
	long rate = 0;

	if (speed == 0)
	{
		// indicates to stop calling the scroller in OnIdle
		rate = 0;
	}
	else
	{
		// set the rate
		rate = 1000/speed;
	}

	return rate;
}

bool CReaderApp::OnIdle(LONG lCount)
{
	CWinApp::OnIdle(lCount);

	// get a pointer to the View
	CMainFrame *pMain = (CMainFrame*)AfxGetApp()->m_pMainWnd;
	CReaderView *pView = (CReaderView *) pMain->GetActiveView();

	// call the View's scroller function
	if (m_viewSafe)
	{
		// retrieve the desired scroll speed from View
		m_rate = setSpeed( pView->GetSpeed() );

		// counter mechanism for controlling the scroll rate
		if (m_rate <= m_counter)
		{
			if (m_rate != 0) pView->scroller();
			m_counter = 0;
		}
		m_counter++;
	}
	return true;
}
```

Appendix A.2

OpenGL Based Code

CReaderView

```
void CReaderView::renderText()
{
// position raster
      glRasterPos2d(-2,-1);

      // load the string (or part of) into OpenGL's back buffer
      glListBase(1000);
      glCallLists(
                  m_textStrAll.GetLength(),
                  GL_UNSIGNED_BYTE, CString(m_textStrAll)
                  );
}

void CReaderView::displayText()
{
      // translate the view into new position
      float posx = float(m_pos.x)/100;
      glTranslatef(posx, 0, 0);

      // flush any other drawing commands
      glFinish();

      // finally swap the buffers
      SwapBuffers( hDC );
}

void CReaderView::scroller()
{
      if ( m_bRenderText )
      {
            // Clear out the color & depth buffers firstly
            glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
            renderText();
      }

      displayText();

      // increment the text position
      if (!m_pause) m_pos.x -= m_scrollIncrement;
}
```

```
void CReaderView::setFont()
{
	// prepare the fonts & sizes for the text
	CFont font;
	font.CreateFontIndirect(&m_logFont);

	// select font into the current device context
	HDC hDC = wglGetCurrentDC();
	SelectObject( hDC, HFONT(font));

	// create the list of font glyphs for use
	wglUseFontBitmaps(hDC, 0, 256, 1000);

	// set the background (clearing) colour
	glClearColor(0.0F, 0.0F, 0.4F, 1.0F);

	m_bRenderText = true;
}
```

Appendix A.3

DirectDraw Based Code

CReaderView

The member m_DD refers to the instance of DDGraphics. m_bufferSize is set by calling the DDGraphics function GetBufferSize().

```
void CReaderView::displayText()
{
	CRect rSrc = m_rSrcBlt;

	// check for source window moving outwith the buffer
	if (rSrc.right >= m_bufferSize.cx)
	{
		// prepare to render used-up surface
		m_DD.RotateBackSurfaces();

		// adjust rSrc back to just before start of the buffer
		rSrc.right = rSrc.right - m_bufferSize.cx;
		rSrc.left = rSrc.left - m_bufferSize.cx;

		m_rSrcBlt = rSrc;
		m_renderText = true;
	}

	m_DD.Blt(m_rDstBlt, m_rSrcBlt);
}
```

```
void CReaderView::scroller()
{
	if (!m_pause)
	{
		// increment the scrolling source window position
		m_rSrcBlt.left += m_scrollIncrement;
		m_rSrcBlt.right += m_scrollIncrement;
	}

	// display the rendered text to screen
	displayText();

	// perform a render to the offscreen buffer if necessary
	if ( m_renderText == true )
	{
		int charsRendered = 0;

		// do the render, keeping track of characters shown
		charsRendered = renderText( m_textToShow );

		// store an index to the last character displayed
		m_lastChar = m_lastChar + charsRendered;

		// remove the remaining string yet to be rendered
		m_textToShow = m_textStrAll.Mid(m_lastChar);

		// check if a new string needs to be loaded
		if ( m_textToShow.IsEmpty() )
			initiateText();
	}
}
```

```
int CReaderView::renderText(CString drawMe)
{
      // obtain the desired font
      CFont font;
      font.CreateFontIndirect(&m_logFont);

      // obtain the text to be displayed
      CString text = drawMe;

      //// not safe for debugging beyond here - screen locked!

      // retrieve a handle to the GDI-style device context
      // of a DirectDraw offscreen (back) surface
      HDC   hDC;
      m_DD.GetBackSurfaceDC( &hDC);

      // fill the offscreen rectangle in the background color
      CBrush brBackground( m_backColour );
      CRect rscreen = CRect(CPoint(0,0), m_surfaceSize );
      FillRect(hDC, rscreen, brBackground);

      // create a rectangle in the background color
      CRect rect( 0, 0, m_bufferSize.cx, m_bufferSize.cy );
      CBrush brBackground( m_backColour );

      // prepare text attributes
      SetBkColor(hDC, m_backColour);
      SetTextColor(hDC, m_textColour);
      SelectObject( hDC, font );

      // get widths of all characters in selected font
      GetCharWidth( hDC, 0, 255, &*m_lpAllCharWidths );

      // calculate how much of string will fit on this surface
      for (int i = 0, lineLength = 0; i < text.GetLength(); i++)
      {
            int charWidth = m_lpAllCharWidths[ text[ i ] ];
            if ( lineLength + charWidth > m_surfaceSize.cx ) break;
                  lineLength += charWidth;
      }

      // write the first i characters to the offscreen surface
      TextOut( hDC, 0, 0, text, i);

      //// Now release the DC... screen unlocked again!
      m_DD.ReleaseBackSurfaceDC(hDC);

      // now that the first i characters of text have been written
      // remove them from the string
      text = text.Mid(i);

      m_bRenderText = false;

      // return index of the last character displayed in given string
      return (drawMe.GetLength() - text.GetLength());
}
```

DDGraphics

```
void DDGraphics::Initialise( HWND hwndScreen )
{
	// release any previous instances of DD object
	m_lpDD->Release();
	m_lpDD = NULL;

	CreateDD( hwndScreen );
	GetCaps( hwndScreen );

	// create surfaces with size of current screen
	CreatePrimarySurface( hwndScreen );
	CreateBackSurfaces();
}

// create the DirectDraw object
// Note that extra code is required to ensure
// DirectDraw uses its version 2 interface, DirectDraw2
void DDGraphics::CreateDD( HWND hWnd )
{
	LPDIRECTDRAW lpOldDD;
	// create 'old' style DirectDraw object
	DirectDrawCreate( NULL, &lpOldDD, NULL );
	// set DirectDraw for normal windows app
	lpOldDD->SetCooperativeLevel( hWnd, DDSCL_NORMAL );

	// query the interface for DirectDraw2 object
	lpOldDD->QueryInterface( IID_IDirectDraw2, (LPVOID *)&m_lpDD );

	// release old DD object
	lpOldDD->Release();
}
```

```
void DDGraphics::GetCaps( HWND hWnd )
{
      // get device capabilies
      DDCAPS    hwCaps = {0}, helCaps = {0};
      hwCaps.dwSize = sizeof(DDCAPS);
      helCaps.dwSize = sizeof(DDCAPS);

      // look for hardware features of interest to us
      m_lpDD->GetCaps(&hwCaps, &helCaps);

      // is there a h/w blitter to blt between surfaces
      m_bCanBltVidMem = (hwCaps.dwCaps & DDCAPS_BLT) ? true : false;

      // find the maximum device screen width and height
      HDC hdc;
      hdc = GetDC(hWnd);
      m_screenWidth = GetDeviceCaps( hdc, HORZRES );
      m_screenHeight = GetDeviceCaps( hdc, VERTRES );
      ReleaseDC(hWnd, hdc);

      // find out how much video memory there is to play with
      DWORD dwTotal, dwFree;
      DDSCAPS ddsCaps;
      ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
      m_lpDD->GetAvailableVidMem(&ddsCaps, &dwTotal, &dwFree);

      // check there is enough video memory for 4 surfaces
      // i.e. primary and 3 offscreen
      if ( int(dwFree) < 4 * dwWidth * dwHeight )
            m_bVidMemFree = false;
}
```

```
void DDGraphics::CreatePrimarySurface( HWND hWnd )
{
      // create the primary surface and attach a window clipper
      // Note that to be compatible with DirectDraw2, the surface
      // must be set up using a DirectDrawSurface3 interface

      LPDIRECTDRAWSURFACE lpOldSurf;
      LPDIRECTDRAWCLIPPER lpddClipper;

      HDC   hdc;
      int   i;

      // initialise surface descriptor
      ZeroMemory(&m_sdPrimary, sizeof(m_sdPrimary));
      m_sdPrimary.dwSize      = sizeof(m_sdPrimary);
      m_sdPrimary.dwFlags     = DDSD_CAPS;
      m_sdPrimary.ddsCaps.dwCaps    = DDSCAPS_PRIMARYSURFACE;

      // create a primary surface
      m_lpDD->CreateSurface(&m_sdPrimary, &lpOldSurf, NULL);

      // update to a DirectDraw3 compatible surface
      lpOldSurf->QueryInterface(IID_IDirectDrawSurface3, &m_lpPrimSurf);

      // release the old surface pointer
      lpOldSurf->Release();

      // create a clipper: this ensures the primary surface is the
      // same size as the application's window.
      DirectDrawCreateClipper(0UL, &lpddClipper, NULL);

      // set the application window to be clipped
      lpddClipper->SetHWnd(0UL, hWnd);

      // attach the clipper to the primary surface and release pointer
      m_lpPrimSurf->SetClipper(lpddClipper);
      lpddClipper->Release();
}
```

```
void DDGraphics::CreateBackSurfaces( HWND hWnd )
{
      // Note that to be compatible with DirectDraw2, the surfaces
      // must be set up using a DirectDrawSurface3 interface

      LPDIRECTDRAWSURFACE lpOldBackSurf1;
      LPDIRECTDRAWSURFACE lpOldBackSurf2;
      LPDIRECTDRAWSURFACE lpOldBackSurf;

      // set description of the back surfaces
      m_sdBack.dwHeight       = m_screenHeight;
      m_sdBack.dwWidth        = m_screenWidth;
      m_sdBack.dwFlags        = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
      m_sdBack.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;

      // if we can use video memory, then do so.
      if ( m_bCanBltVidMem && m_bVidMemFree )
            m_sdBack.ddsCaps.dwCaps |= DDSCAPS_VIDEOMEMORY;

      // create the first back surface
      m_lpDD->CreateSurface(&m_sdBack, &lpOldBackSurf, NULL);

      // employ the DirectDraw3 imterface for this surface
      lpOldBackSurf->QueryInterface(IID_IDirectDrawSurface3,
                                    (LPVOID *)&m_lpBackSurf);
      // release the old surface pointer
      lpOldBackSurf->Release();


      // do the same for 2nd surfaces
      m_lpDD->CreateSurface(&m_sdBack, &lpOldBackSurf2, NULL);
      lpOldBackSurf2->QueryInterface( IID_IDirectDrawSurface3,
                              (LPVOID *)&m_lpBltSurf2);

      // release the old surface pointer
      lpOldBackSurf2->Release();

      // and finally the 3r surface
      m_lpDD->CreateSurface(&m_sdBack, &lpOldBackSurf1, NULL);
      lpOldBackSurf1->QueryInterface( IID_IDirectDrawSurface3,
                              (LPVOID *)&m_lpBltSurf1);
      lpOldBackSurf1->Release();
}
```

```
void DDGraphics::Blt(CRect rDst, CRect rSrc)
{
      // blit from two separate surfaces, treating them a single buffer
      // assumes source and destination are the same size

      CRect rSrc1 = rSrc,
            rSrc2 = rSrc,
            rDst1 = rDst,
            rDst2 = rDst;

      // destination width
      int dwidth = rDst.right - rDst.left;

      // rSrc1 and rSrc2 should never go outwith 0 and m_surfaceWidth

      // if source has negative left edge, wrap rSrc1
      // back around onto its surface
      if ( rSrc.left <= 0 )
            rSrc1.left = rSrc.left + m_surfaceWidth;

      // if right edge scrolls past buffer boundary,
      // wrap rSrc2 around to beginning of its surface
      if ( rSrc.right >= ( 2 * m_screenWidth ) )
            rSrc2.right = rSrc.right - m_surfaceWidth;

      // set fixed points of two sub rectangles
      rSrc1.right = surfaceWidth;
      rSrc2.left = 0;

      // get new widths of source rectangles
      int width1 = rSrc1.right – rSrc1.left;
      int width2 = rSrc2.right – rSrc2.left;

      // adjust destination rectangles to match these widths
      rDst1.right = rDst1.left + width1;
      rDst2.left = rDst2.right – width2;

      // perform the blitting operations
      // provided the source rectangles are valid
      if (width1 > 0)
            m_lpPrimSurf->Blt( &rDst1, m_lpBltSurf1, &rSrc1 );
      if (width2 > 0)
            m_lpPrimSurf->Blt( &rDst2, m_lpBltSurf2, &rSrc2 );
}
```

```
void DDGraphics::RotateBackSurfaces(BOOL forward)
{
	// this function flips the three back surfaces
	// in a circular fashion
	LPDIRECTDRAWSURFACE3 tmpSurf;

	if (forward)
	{
		tmpSurf = m_lpBackSurf;
		m_lpBackSurf = m_lpBltSurf2;
		m_lpBltSurf2 = m_lpBltSurf1;
		m_lpBltSurf1 = tmpSurf;
	}
	else // backwards
	{
		tmpSurf = m_lpBltSurf1;
		m_lpBltSurf1 = m_lpBltSurf2;
		m_lpBltSurf2 = m_lpBackSurf;
		m_lpBackSurf = tmpSurf;
	}
}
```

DDGraphics Page Flipping

The code for Page flipping is included for completeness. It has not yet been integrated into the main application.

```
void DDGraphics::CreateFlipSurfaces()
{
	// create two back buffer flippable surfaces
	LPDIRECTDRAWSURFACE lpFlipSurf1;
	LPDIRECTDRAWSURFACE lpFlipSurf2;

	// flippable back surfaces should be almost identical to primary
	// so describe primary surface and copy onto flip descriptionS
	m_lpPrimSurf->GetSurfaceDesc(&m_sdPrimary);

	memcpy(&m_sdFlip, &m_sdPrimary, sizeof(DDSURFACEDESC));
	m_sdFlip.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
	m_sdFlip.dwFlags = DDSD_WIDTH | DDSD_HEIGHT
	                  | DDSD_PIXELFORMAT | DDSD_CAPS;

	// if we can use video memory, then do so.
	if (m_bCanBltVidMem)
		m_sdBack.ddsCaps.dwCaps |= DDSCAPS_VIDEOMEMORY;

	// now create flip surface 1 in similar manner as before
	m_lpDD->CreateSurface(&m_sdFlip, &lpFlipSurf1, NULL);
	lpFlipSurf1->QueryInterface( IID_IDirectDrawSurface3,
	                                   (LPVOID *)&m_lpFlipSurf1);
	// now create flip surface 2
	m_lpDD->CreateSurface(&m_sdFlip, &lpFlipSurf2, NULL);
	lpFlipSurf1->QueryInterface( IID_IDirectDrawSurface3,
	                                   (LPVOID *)&m_lpFlipSurf2);

	// attach flip surfaces to the primary
	m_lpPrimSurf->AddAttachedSurface(m_lpFlipSurf1);
	m_lpPrimSurf->AddAttachedSurface(m_lpFlipSurf2);
}


void DDGraphics::Flip()
{
	// flip with wait until finished flag
	return m_lpPrimSurf->Flip(NULL, (DWORD)DDFLIP_WAIT);
}
```

Appendix A.4

Mission Statement

[1] Macular degeneration literature ***

[2]

[3]

[4]

[5] Elderly people and HCI …

[6] PAEP

[7] Edinburgh libraries

[8] eccentric fixation

[9] Goodrich et al {hmdp497} image slips

[10] saccadic movement BJO vol85 No12, Safran, Duret, Issenhuth, Mermoud (Geneva) Dec 1999

[11] cctv

[12] pictures courtesy of www.enhancedvision.com

[13] CCTVs head mounted and older ones..

[14] text-to-speech

[15] PC CCTV

[16] …

[17] Sony Glastron

[18] wearables

[19] UML

[20] java threads

[21] OpenGL references

[22] DirectDraw references

[23] contrast

[24] EOG eye tracking

[25] Betsie

[26] st vision

[27] OpenGL