Virtual Reality and Program Comprehension

Application Using Spreadsheet Visualisation

Robert Duncan Gregson

M.Sc. Computer Science
University of Durham

1994

Abstract

Program comprehension is an important function undertaken in the process of software maintenance. Compared to other research subjects, program comprehension has received little attention even though it is one of the biggest influences on a programmer's output. Research into aiding program comprehension has led to software visualisations, but these are mainly two dimensional views and overload the viewer with information. With the advent of more powerful computers, virtual reality can be used to create three dimensional visualisations, in which the viewer is able to navigate freely.

Spreadsheets were studied in this work on visualisation because programming languages are extremely complex and a model employing spreadsheets was developed. Spreadsheets offer many similarities to programming languages, for example, cell referencing and formulas in spreadsheets are similar to procedure calls, variable referencing and data manipulation in conventional programming languages. Common mistakes made in spreadsheets have been shown to be very difficult to locate, mainly because the spreadsheet user has a reduced ability to make hypotheses about the computational domain of a spreadsheet. Therefore, in order to address this shortcoming a visualisation model was developed to allow a spreadsheet user to be able to view both the problem domain (the *what*) and the computational domain (the *how*) simultaneously. A spreadsheet, a spreadsheet description language and a virtual reality system were the objects in the model, and a generator and translator were the links between those objects. Implementing the model indicated that spreadsheets could be visualised in virtual reality, and this technique was shown to improve the process of spreadsheet comprehension.

Declaration

No part of the material offered has previously been submitted by the author for a degree in the University of Durham or in any other University. All of the work presented here is the sole work of the author and no-one else.

Table of Contents

List of Illustrations

1 Introduction

1.1 Software Engineering

1.1.1 Overview

The term 'software engineering' was first introduced in the late 1960's, to address what was then called the software crisis. Macro and Buxton [1] stated that software engineering was, historically, the term to denote programming, and that in 1968 it took on a more widespread meaning to include a whole set of activities including programming. Today, software engineering encompasses a large number of activities.

There are many interpretations of software engineering, but the underlying principles are quite similar. Sommerville [2] states that engineered software has four key attributes (assuming that the software provides the required functionality); maintainability, reliability, efficiency and an appropriate user interface. Macro and Buxton [1] proposed that software engineering is the set of activities to produce high quality systems with known limitations of resources. Sommerville [2] and Pressman [3] both agree on this point. Pressman believes that software engineering encompasses three main elements, those of methods (the technical 'how to'), tools ([semi]automated support for the methods, i.e. Computer Aided Software Engineering) and procedures (the overall management system).

All of the authors believe that software engineering is some sort of structured process by which software can be developed from an idea detailed by a client right up to a fully maintained system. The ANSI/IEEE [4] standard definition of software engineering also reflects this attitude:

> ***Software Engineering:*** *The systematic approach to the development, operation, maintenance, and retirement of software.*

> ***Software:*** *Computer programs, procedures, rules, and possibly associated documentation and data pertaining to the operation of a computer system.*

A valuable point illustrated here (also by Macro and Buxton [1] and Sommerville [2]), is that software engineering also includes the documentation and data. Sommerville [2] points out some key attributes of software engineering stating that it:

- is concerned with software systems built by teams, not individuals
- uses engineering principles in the development of these systems
- includes both technical and non-technical aspects

In summation, Sommerville [2] puts it best by stating that software engineering involves producing products in a cost-effective way and that the challenge for software engineers is *"to produce high quality software with a finite amount of resources and to a predicted schedule"*.

1.1.2 Waterfall model

The software engineering process has been defined in many different ways, but the key elements of the process remain the same. One such simple general process is the Waterfall model, called so because of the cascading feature from stage to stage.

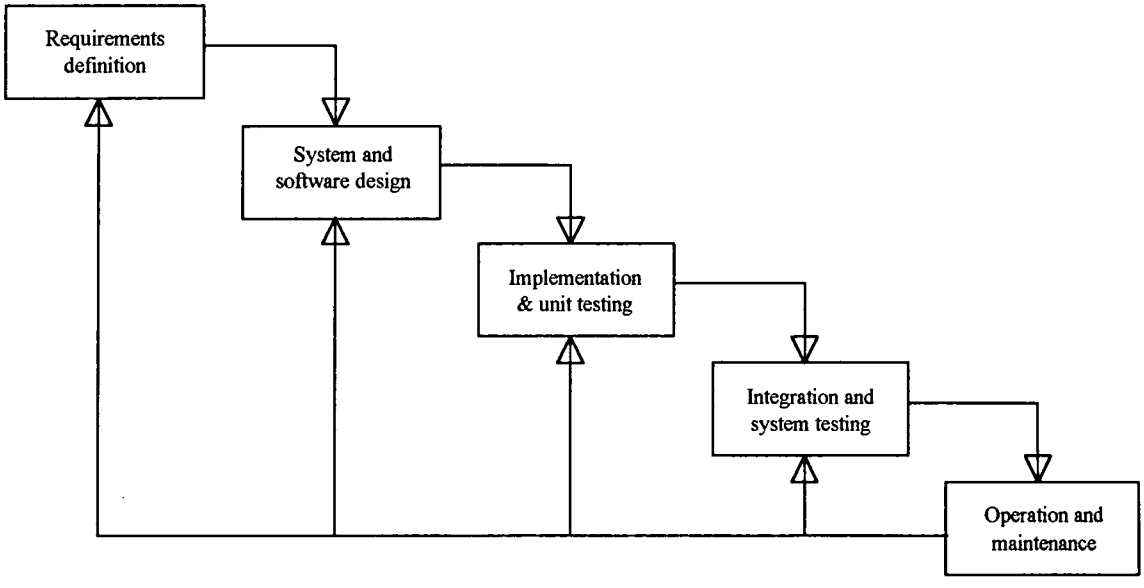


Figure 1.1 The Waterfall life cycle. For an explanation, please read the text.

The Waterfall model is one such process (or life cycle) that demonstrates the stages of software development and on-going maintenance. In fact, the ANSI/IEEE [4] standard declares that software development and maintenance are the two components that define the software life cycle, which in turn includes all the stages shown above in figure 1.1. To put each stage into context, a brief description of each of the stages follows below, starting with the ANSI/IEEE [4] definition for that stage.

1.1.2.1 Requirements definition

The process of studying user needs to arrive at a definition of system or software requirements. Requirements (and/or specifications) are established by consultation with clients, in view to form a blue-print from which the software system can be designed. The requirements are often defined in a manner that is understandable by both the client and the software engineers.

1.1.2.2 System and software design

The period of time in the software life cycle during which the designs for architecture, software components, interfaces, and data are created, documented and verified to satisfy requirements. The design phase is a type of translation from requirements (usually in a natural language) to some sort of structured representation of the software from which an executable program may be derived. This also allows for the system to be assessed before the implementation begins.

1.1.2.3 Implementation and unit testing

The period of time in the software life cycle during which a software product is created from design documentation including the process of locating, analysing and correcting suspected faults. At this stage the design is translated into a set of programs or program units, sometimes referred to as the 'coding' stage. Unit testing is set of activities that is undertaken by the software engineer to make sure that a program unit functions as it is required to do by the specifications.

1.1.2.4 Integration and system testing

An orderly progression of testing in which software elements, hardware elements, or both are combined and tested until the entire system has been integrated. [Also] the process of testing the integrated hardware and software system to verify that the system meets its specified requirements. Once the program units have been tested by themselves, they are brought together to contribute towards the finished product. Integration testing is the testing done as the program units are brought

together as opposed to system testing which is carried out on the finished product to make sure that it functions in accordance with its requirements.

1.1.2.5 Operation and maintenance

The period of time in the software life cycle during which a software product is employed in its operational environment, monitored for satisfactory performance, and modified as necessary to correct problems or to respond to changing requirements. Finally, once the completed product has been fully tested, it is installed and put into use. From this point onwards any tasks carried out on the software system are classed as maintenance.

1.2 Software Maintenance

Defined by ANSI/IEEE [4] as *"Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment"*, software maintenance is the last stage of the life cycle, but arguably the longest. Maintenance covers the time period from when a software product has been delivered right until it is no longer in use which is, on average, 10 to 15 years according to Osborne and Chikofsky [15].

Foster *et al* [5] point out that, as software maintenance resides at the end of the software life cycle, it is treated in three different ways by published descriptions of the software life cycle:

- as simply another activity at the end
- (in diagrams) as a series of feedback lines, implying repetition of some or all of the preceding activities
- by ignoring it

Software maintenance is receiving a growing audience as it is now being realised that maintenance is a major role in the software life cycle. Cornelius *et al* [7] state that *"Software maintenance is recognised as the most expensive phase of the life cycle"*. Different publications have differing ideas as to exactly how much of resources is put into maintenance. A study by Lientz and Swanson [17] showed that about 50% of total programming effort was devoted to maintenance, another study by McKee [80] revealed that the percentage was about 65 to 75. Pressman [3] summarised that maintenance could account for up to 70% of total effort, while Foster *et al* [5] indicated that *"40% to 70% of all software expenditure goes into maintenance"*. As it can be seen

that there is quite a bit of diversity over what the real figure is, but it would seem likely that the cost will vary from product to product.

Obviously, those publications that choose to ignore software maintenance, are simply leaving out a large proportion of the whole picture. As Foster *et al* state so well, *"the third option* [ignoring software maintenance] *merely stores up problems for the future"*.

1.3 Program Comprehension

Program comprehension or program understanding as it is also known has been estimated at consuming 50% to 90% of the maintenance stage [(Robson *et al*) 6, (Standish) 18]. By this statement alone, program comprehension can be seen to be one of the biggest factors in any software product's life. Assuming that the statistics presented so far are a fair representation of real life developments of software, program comprehension could take up anything from 20% (50% of 40%) to 63% (90% of 70%) of the whole software life cycle. Why then, is program comprehension not the most researched area?

Robson *et al* [6] indicates that *"Very little research effort has been put directly into this phase of the life cycle. Research effort has been almost exclusively devoted to the development stage with the aim of reducing the cost of maintenance by using new development techniques"*. This is a fair point, such that researchers are attempting to tackle the source of the problem, namely bad development techniques. However, Robson goes on to point out that this *"ignores the problem of the maintenance of existing software"*.

So, perhaps, program comprehension cannot expect to gain the attention it deserves until its 'parent' (software maintenance) is focused upon.

1.4 A wider context

Program comprehension is only one of a number of things that make up software engineering. As a summary of the above sections, figure 1.2 shows how program comprehension fits in with software maintenance and the life cycle.

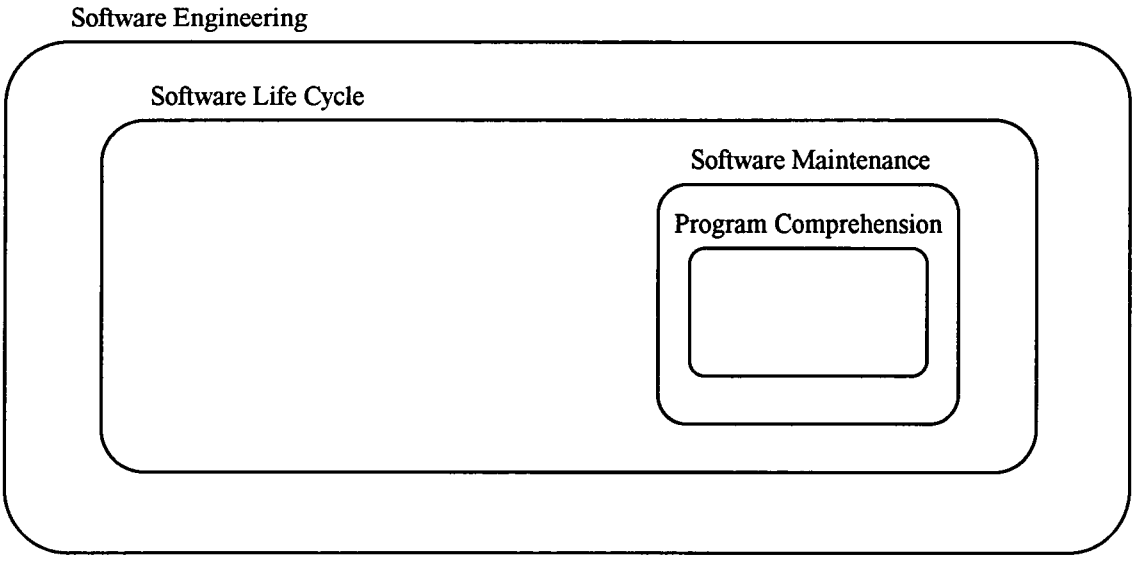


Figure 1.2 How program comprehension fits into the wider context of software engineering. Labels are outside their respective boxes.

The aim of this section is to place program comprehension and the specifics of this work into a wider context, to describe the purpose of this research.

In May 1986, IEEE Software [8] devoted a whole issue to software maintenance. The guest editor's introduction gave a brief summary of software maintenance. About half of this was given over to understanding programs (or program comprehension). This sub-section of the introduction is mainly what prompted the research into this area. What follows are the two quotations from this article that summarise the reasons behind this work:

> *"All too often maintainers are faced with hard-to-understand software for which documentation is missing or out of date. Re-creating the documentation is typically out of the question in deadline-driven maintenance shops because of the time required and the difficulty in understanding software previously maintained without programming style standards".*

> *"Understanding and modifying programs are fundamental to software maintenance and are influenced by*
>
> - *the information about programs that is available to the maintenance programmer,*
> - *the format in which this information is presented,*
> - *the media used to present the information, and*

- *the operations available for viewing and analysing the information."*

Addressing the first quotation, perhaps the whole key to software maintenance is for the maintainer to be able to understand what the original developers were thinking of at the time of conception (and possibly implementation). How often has the phrase "why has he/she done that?" been heard in a maintenance environment? The key element to all of this is to transfer the knowledge from the developer to the maintainer. This is backed up by the first point of the second quotation, namely that the more [relevant] information that is available to the maintainer, the better chance that he/she has of understanding the software in question.

1.5 Research area and criteria for success

Being able to understand large programs is a very complex task and by the state of the art of current research can be seen to be a few years away yet. This research is aimed at understanding a simpler, smaller, less complex system and language. To this end, spreadsheets have been selected as an ideal tool for visualisations as they contain many similarities with their programming language counterpart. Instead of visualising programs, the aim is to represent spreadsheets, such that a maintainer can obtain a better understanding of how a particular spreadsheet functions.

Being able to visualise spreadsheets is a problem in itself. Hendry and Green [9] tackled this issue, their aim being to *"make it easier to describe a spreadsheet to a co-worker"*. Essentially this is the aim of this research, to provide a tool that will aid a maintainer in his or her understanding of the workings of a spreadsheet, whether it was written by themselves or not.

In order for this research to be considered a success, the following points will have to be addressed:

- examining program comprehension theories and how they may be put to use in comprehending spreadsheets
- that spreadsheets can be represented in a virtual reality world
- and that it is possible to implement this technique (of visualising spreadsheets)

Only once these points have been addressed can this research be considered a success.

1.6 Thesis overview

1.6.1 Layout

This thesis is broadly divided into two halves. The first half is comprised of chapters one, two and three. This forms the introduction and literature survey of the research. Chapters two and three detail the two main subject areas behind this research; program comprehension and virtual reality.

The second half of this thesis is made up from chapters four, five, six and seven. It details the ideas obtained from applying virtual reality (chapter three) to program comprehension (chapter two). These ideas are then explored, implemented and evaluated. Finally, the work is concluded in chapter seven.

What follows is a chapter by chapter guide of this thesis, including this chapter (in most cases, the text below can be found at the start of the relevant chapter):

Chapter One: Introduction

The aim of this chapter is to give a general background to the subject area by indicating how program comprehension fits into the wider context of software engineering and software maintenance. In addition this also provides an introduction to the structure of the thesis work, documentation and the criteria for success.

Chapter Two: Program Comprehension

The aim of this chapter is to present a brief overview of software maintenance, putting program comprehension into perspective. This will then be followed by a detailed account of program comprehension and its theories. Hence the reader is introduced to the problems associated with program comprehension.

Chapter Three: Virtual Reality

The aim of this chapter is to present an introduction to virtual reality, by exploring what is needed for it, what the problems are with it and to what types of applications it is being used for. It is also the intent of this chapter to dispel some of the myths that have arisen from all the 'hype' that virtual reality has received.

Chapter Four: Visualising Spreadsheets

The aim of this chapter is to bring together the notions presented in the previous two chapters (chapters two and three) and to indicate how the technologies involved in virtual reality can be applied to program comprehension. However, as complete program understanding is beyond the scope of this work, spreadsheets are introduced as a substitute for programming languages. Spreadsheets offer the same characteristics and share many of the same problems as a small programming language would have. To this end, this chapter explores how virtual reality technology can be applied to the problems associated with spreadsheet maintenance and, as a side-effect, spreadsheet construction and debugging.

To end this chapter, the theories presented will be probed to see how they can be extended to program understanding.

Chapter Five: Implementation

The aim of this section is to describe how the ideas in chapter four were turned into reality with the use of current technology and applications. Incorporated into this is a description of *trans* (a C language [75] program) and SDL (Spreadsheet Description Language) both constructed specially for this research. Details of REND386, VR386, Minicalc and Microsoft Excel are also included as the 'already available' applications used to facilitate the research.

Chapter Six: Evaluation

The aim of this chapter is to evaluate the work presented so far by exploring examples of how common errors in spreadsheets can be recognised and corrected using the model presented in chapter five.

Chapter Seven: Conclusion

The aim of this chapter is to conclude the work undertaken in the previous chapters by briefly detailing and critically appraising it. The criteria for success will be examined and discussed. This will then be followed by a section detailing possible future work in this subject area.

2 Program Comprehension

The aim of this chapter is to present a brief overview of software maintenance, putting program comprehension into perspective. This will then be followed by a detailed account of program comprehension and its theories.

2.1 Software Maintenance

2.1.1 A definition

Although authors may propose different definitions for software maintenance, those differences are only minor and each arrives at nearly the same conclusion, namely that software maintenance is any action performed in respect to a piece of software, whether it be technical, non-technical or managerial, after that software has been delivered and installed.

It is worth noting that many authors agree that it is impossible to produce software that will not need to be maintained, in fact Lehman and Belady [81] suggest that maintenance is inherent of software.

2.1.2 An overview

Software maintenance is gaining much (rightly deserved) attention from academia and industry alike. Read almost any paper on software maintenance and the reasons will become apparent (for example [2, 3, 5, 6, 10, 35, 36, 38, 40]). The computer industry is waking up to the fact that maintenance consumes the largest portion of the software life cycle [2, 3, 6, 38], so large in fact, that Canning characterised it as an 'iceberg' [16]. Estimates of exactly how much of resources maintenance utilises range from a conservative 40% to a more likely 75%. The figures are so high because it is often economically sound to extend the life of a software product for as long as possible [(Bennett *et al*) 10] and this is reflected in the fact that the average life span of software is currently 10 to 15 years [(Pressman) 3]. Now assume that it took 3 years to develop a piece of software, it can be derived that (on the average) this piece of software would spend between 20% and 33% of its life in

development and between 67% and 80% being maintained. This is why 75% is a much better estimate of exactly how much is consumed by maintenance.

Once this fact has been established it would be reasonable to suppose that maintenance is a high profile activity. Not so, generally because it is regarded as *second class* [10], *of lesser importance* [(Lientz and Swanson) 35], and so on. Not only has maintenance achieved this reputation, it is often lucky enough just to have one as it is sometimes neglected altogether with the view being that a software engineer's role finishes once the software fulfils the requirements [(Genuchten *et al*) 40]. However, any software engineer would be happy to announce that if there is one thing that is inherent to change, it is requirements, thus resulting in maintenance.

Times are changing, more and more papers are being published about software maintenance, spreading the news that you had better look out for the 'iceberg' before you sail your ship right into it. Recent studies show that maintenance is regarded by companies as at least as important as development [35]. For example, one set of results depicted that 48% of personnel were allocated to maintenance and 46.1% to development, certainly indicating that maintenance had an equal priority. A second set of results clearly demonstrated the fact that maintenance was of more importance as can be seen in figure 2.1.

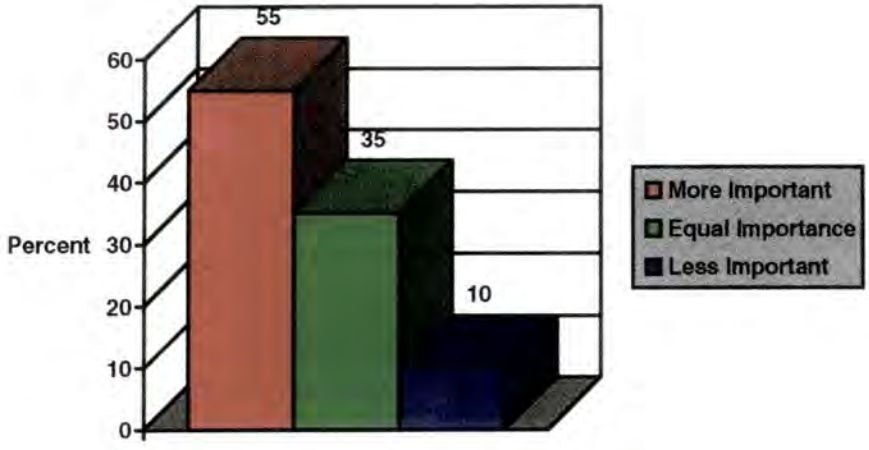


Figure 2.1 Importance of maintenance versus development. From a survey of Dutch companies this graph depicts how they regarded the importance of maintenance against the importance of development. For example, 55% stated that maintenance was of more importance to them than was development [(Lientz and Swanson) 35].

In the light of such evidence it is no wonder then that a new standard for judging software is being accepted, that of *how maintainable is it?* [(Schneidewind) 36]. Generally speaking, for a piece of software to be maintainable it had to have been designed with maintenance in mind. This is one current area of research, led by the belief that a better design process will result in more maintainable software, although, Bennett *et al* points out that this is rarely substantiated [10], perhaps because researchers are still not properly constructing design methodologies with maintenance in mind. However, this is no good for the existing software that has to be maintained.

Economics is undoubtedly the most powerful propellant of any change, so it is interesting to note why companies have not realised the true potential of software maintenance. For is it wise to limit one's attention to an activity that only consumes about 20% of the whole activity? Attention must be drawn to software maintenance and how important it is to design for change in mind.

2.1.3 Types of modification

Maintenance is the process of change in software. It would be reasonable for anyone to say *"...but I don't spend 70% of my time fixing bugs"*, and they would be quite right as there are four classes into which modifications may fall. What follows is a brief description of each type of maintenance task with current distribution figures from the Centre for Software Maintenance [11].

2.1.3.1 Perfective (60%)

The activity of improving or enhancing the software without changing its functionality. For example, re-writing a calculation routine because it is too slow and needs to be speeded up. These types of modification usually occur due to changes in the requirements.

2.1.3.2 Adaptive (18%)

Modifying the software to keep up with a changing environment. Essentially this is referring to the fact that new hardware is updated or released much faster in comparison to the average life span of software. This means that the hardware, for which the software was written, will change (if not become obsolete) over the period of the software's use. Thus changes must be made to support the new hardware.

2.1.3.3 Corrective (17%)

This is the *bug fixing* category. That is to say that modifications of this nature occur because there is a difference between what the software does and what it is intended to do (by either the developers or the client). Maintenance of this type is usually due to faults missed at the testing stage of development.

2.1.3.4 Preventive (5%)

Modification of the software to improve future maintainability, or to provide a better stand-point from which future enhancements can be made. An example of this type may be the restructuring of source code so that it meets in-house rules and conventions. This type of maintenance is usually caused by bad or deficient design.

2.2 Program Comprehension

Examining the categorisations above, it would seem sensible to believe that change is unavoidable, and that would be correct [(Lehman and Belady) 81]. However, change cannot be implemented without understanding. A maintainer must understand a program before he/she can change it, in fact, the better the understanding the more likely the change will be correct [(Letovsky and Soloway) 12].

Program comprehension, however, is not confined to software maintenance. Basili [27] states that program comprehension is carried out on a daily basis and this would be reasonable to believe when it is considered that program comprehension is used in many phases of the life cycle. For example: testing phase (for debugging), implementation phase (for code reviews), software re-use (the function of the code must be established before it can be re-used) or any other activity that requires a programmer to gain an understanding of the program (perhaps documenting a program after it has been written). In light of this evidence it cannot be denied that program comprehension is the major factor in software development [(Standish) 18]. This in turn would point to the fact that it probably is the single most influencing factor of a programmer's ability to produce output. Industry is always focusing on how it can improve the efficiency of its work force, so yet again here is another propellant for the attention needed to be diverted to program comprehension.

Maintainers spend a major proportion of their time studying the intent and style of the original programmer [(Littman *et al*) 29]. Even if documentation is present, research has shown that code can be studied in excess of 3½ times that spent studying the documentation [(Robson *et al*) 6]. What is needed is some way of speeding up this process, or at least making it easier. Much attention has been focused on constructing computer programs that 'can understand code' (for example the PROUST system [31]) resulting in the maintainer having to do less work to concrete his theories about what the software's function is. The other main research area is tackling what could be considered the root of the problem, namely design. Constructing a design process that considers maintainability a major factor is certainly a step in the right direction and would produce more maintainable software. However, as stated before, this does nothing for the legacy of code that is already in existence.

2.2.1 Theories of program comprehension

Once it has been established that there is a problem (understanding programs), what can be done to aid the situation? Unfortunately program comprehension has only really been studied from the late seventies onwards. Little is known about it and little research has been done in respect to it. Even though, there are a few dominant theories or strategies for comprehending programs. Most of this work has been carried out by observing programmers carrying out maintenance tasks on small programs [(Younger and Bennett) 13]. This process has often been referred to as *thinking out loud* experiments [(Letovsky) 28].

What follows is an overview of the main types of comprehension strategies that are being researched today, and are then discussed in a subsequent section.

2.2.1.1 Systematic / As-needed strategy

Littman *et al* [29] carried out one such *thinking out loud* experiment where they analysed video tapes of software engineers maintaining a database program. The authors believed that there was a small plan to making a modification in software:

- **Approach:** the method used to gain knowledge of the software.
- **Knowledge:** a mental model of the software giving a general overview of the program's functions and interactions. The integrity of the model would be reflected in the type of approach used.
- **Modification:** the actual change being implemented. The success of the modification would depend on the correctness of the mental model.

Hence the main factor being the approach as all subsequent stages rely upon the effectiveness of this information gathering process. Littman *et al* conjectured that there were two types of approach to understanding a program, the *systematic approach* and the *as-needed approach*.

Systematic approach

This is the process of understanding the whole program before any changes are to be made. It would usually involve executing the program symbolically, identifying data flow and control flow between subroutines.

As-needed approach

Program reading time is minimised by partitioning the software such that all parts that will not be affected by the change are ignored. The remaining portions of the software are then studied; *"attention to the program was guided by the need to gather information required to answer specific questions about local program behaviour"*. Once the maintainer believed that he/she had gathered enough information, the modification is implemented.

Littman *et al* stated that *"to understand a program a maintainer must know about the objects the program manipulates and the actions the program performs"*. They go on to add that the maintainer must also acquire two forms of knowledge about the software; static knowledge (knowledge of the code that performs the task) and casual knowledge (knowledge about the connections and interactions between tasks in the software). Using this knowledge the maintainer would then be able to construct the aforementioned 'mental model' of the software. Obviously it can be seen that the mental model depends solely on the correctness of the knowledge obtained about the software.

The results of the experiment indicated to the authors that the as-needed approach only allowed a weak mental model to be constructed. Although such a technique is faster in constructing the enhancement, later repercussions occur when testing and debugging force the maintainer to examine the code more thoroughly and remove the effects of their previously undetected mistakes.

As a final note the authors pointed out that it is clearly impossible to use the systematic strategy on large systems as the process would take far too long. Therefore some form of as-needed strategy is required.

2.2.1.2 Syntactic / Semantic knowledge

Schneiderman and Mayer [20] proposed that the program comprehension process is the *formation of internal semantics for a given program.* Such a formation would consist of a multi-levelled representation with levels ranging from *high* to *low*. At the higher levels there would be an overview of what the program does, leading down to lower levels where actual fragments of code familiar to the maintainer would reside.

This lead Schneiderman and Mayer to conjecture that there are two types of knowledge:

- **Semantic** general programming concepts that are independent of specific programming languages. For example, possessing knowledge about a binary search tree, its properties and how it functions.

- **Syntactic** details of the actual statements needed to implement a programming concept. For example, knowing what C statements [75] (and their syntax) are required to implement a binary search tree.

Evidence to substantiate these claims was in the form of two observations. Firstly, to corroborate the two types of knowledge theory, the authors stated that learning a second programming language is easier (provided that is has the same semantic structure) than a first because the first requires familiarisation with both semantic and syntactic whereas the second only requires acquiring syntactic. Further to this, they also pointed out that previous semantic knowledge can, in fact, conflict with the acquisition of a second language if the first language has radically differing semantic structures. This could be observed when trying to learn a functional language after learning, say, a procedural one.

Secondly, Schneiderman and Mayer conducted an experiment where programmers of differing skill levels were asked to memorise two 20 line FORTRAN programs. This required the programmers to become familiar with the program in a timed environment. Expert programmers remembered more of the programs than did novice programmers. The authors interpretation of these results was that expert programmers concentrated more on building a semantic representation of the program, whereas novice programmers relied more upon the retention of specific code. To concrete this claim, Schneiderman and Mayer also noted that the recall errors amongst the expert programmers occurred mainly due to syntax mistakes, however the overall meaning of the program was retained.

In conclusion, the authors believed that the ability to memorise and recall a program is a strong correlate of program comprehension, and that this comprehension process does not take place on a line by line basis but more on a hierarchical semantic model of the program.

2.2.1.3 Program Slicing

The theory of program slicing was first introduced by Weiser [21, 23] and is defined as a *technique for restricting the behaviour of a program to some specified subset of interest...* [such that] *slices are complete programs which compute a restriction of the specification* [(Gallagher and Lyle) 22]. This means that a program slice is a program in itself, i.e. it is executable. Following on from this Weiser added, therefore program slicing is *"the process of stripping a program of statements which do not influence a given variable at a given statement"*. To illustrate this idea figures 2.2 and 2.3 demonstrate a small timer program and one slice performed on it, with respect to the variable *seconds*.

Weiser [21] stated that program slicing is based on control flow and data flow analysis, which is depicted quite clearly in figure 2.3. The program slice (figure 2.3) demonstrates how all control flow and data flow relating to unwanted variables is eliminated. However, it should be

pointed out that this technique does not account for statements such as **delay(990)** which are crucial to the correct operation of the program but are left out in the slice, and combined output statements (i.e. the **printf** statement), which need to be split up manually before any slicing can take place (see figures 2.2 and 2.3).

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

void main(void)
{
  register int seconds = 0;
  register int minutes = 0;
  register int hours   = 0;

  while (1)
  {
    delay(990);
    seconds++;

    if (seconds >= 60)
    {
      minutes++;
      seconds = 0;
    }

    if (minutes >= 60)
    {
      hours++;
      minutes = 0;
    }

    if ( kbhit() )
    {
      getch();
      break;
    }
  }

  printf("Time elapsed =
%02d:%02d'%02d\n", hours,
minutes, seconds);
}
```

Figure 2.2 TIMER.C, a small program to count the number of hours, minutes and seconds since the program was invoked. Pressing any key on the keyboard stops the program. The comments have been removed to use less space in the document.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

void main(void)
{
  register int seconds = 0;

  while (1)
  {
    seconds++;

    if (seconds >= 60)
    {
      seconds = 0;
    }

    if ( kbhit() )
    {
      getch();
      break;
    }
  }

  printf("Time elapsed =
%02d:%02d'%02d\n", hours,
minutes, seconds);
}
```

Figure 2.3 A slice on TIMER.C, depicting the seconds. Gallagher and Lyle [22] believed that this is how a programmer would see the program (shown left) if he/she was only concentrating on the variable *seconds*.

Weiser's claim was that programmers used slicing as a debugging technique [23], such that the maintainer would locate statement(s) where an error would first manifest itself and then trace

backwards in an attempt to locate its place of origin. Weiser termed this technique *working backwards.*

He substantiated his theories by conducting an experiment where programmers were given different types of slices (ranging from *relevant slices* to *jumble*) for three programs. Weiser conjectured that the results did show that slicing was used as a debugging aid. His main premise for program slicing being used as a maintenance aid for understanding programs was the notion that slices are useful because the maintainer would only have to comprehend a small subset of the original program. Therefore, the maintainer would not have to waste time learning about portions of the program that are of no relevance to the modification task at hand.

Gallagher and Lyle [22] applied Weiser's theories of program slicing to software maintenance. They extended the idea of a slice to a *decomposition slice*. A decomposition slice is independent of a line number (Weiser's required a line number) and catches all computation in a program that affects the variable under scrutiny. Similarly a decomposition slice executes a subset of the program specification, but something termed the *complement* executes the remaining specification. Statements belonging to the complement have no influence on the variable which is the subject of the decomposition slice. Although it should be noted that in order for both the decomposition slice and the complement to be executable there are some statements within a program that are required by both. Such statements are called *crucial statements*, an example of which is the **main()** statement in C [75].

In order to construct a decomposition slice, Gallagher and Lyle use a technique employed by dead-code elimination. All *critical statements* (in this case the statement(s) under consideration by the maintainer, perhaps because this is where an error first manifests itself) are marked and then *use-definition* chains are traced back through the program to indicate which statements have an affect on the critical statements. All statements that do not affect the critical statements and those statements that are not crucial statements are discarded and the decomposition slice is left. The complement is then considered to be those statements that were discarded plus the crucial statements.

From this analysis, Gallagher and Lyle believed this helped maintainers achieve their modification as they only had to concentrate on understanding the decomposition slice. It also had the advantage that if the maintainer changed any part of the complement during the process of modification then a mistake had obviously been made. The essential factor being that the decomposition slice should be the only portion of the program that is allowed to be changed, the complement must always remain untouched.

2.2.1.4 Beacons

The originator of this theory is Brooks, in his paper *"Towards a theory of the comprehension of computer programs"* [19]. He believed that program comprehension was all about reconstructing mappings between the *problem domain* and the *programming domain*. Where the problem domain is defined to be the problem/task that the software is solving; and the programming domain is the set of constructs/statements used to implement the problem as a piece of software.

It was his belief that the process behind program comprehension involved starting with a global hypothesis which is generated as soon as the software's name or a small description of the software is read. Sub-hypotheses are then formulated in a hierarchical manner such that those lower in the hierarchy are more specific in detail and add concreteness to those hypotheses directly above it. This process is iterative (in a top down fashion) and continues until the knowledge needed to perform the task is acquired [(Brooks) 25].

Brooks did point out that *"knowledge of the problem domain can be critical in making hypotheses about the program"*. That is to say that the ability to generate an appropriate hypothesis is a function of the overall skill level of the maintainer and his/her experience in that particular problem domain. It would then follow that a lack of these skills would lead to a bad hypothesis which results in one of two things (depending on whether the maintainer realises their misunderstanding), firstly an incorrect modification to the software or secondly time wasted investigating an incorrect hypothesis.

Brooks believed that hypotheses were formed from *beacons* in the code. He defined a beacon to be a *"set of features that typically indicate the occurrence of certain structures or operations within the code"*. For example, one beacon often cited by research in this area is that of a simple type of sort algorithm [(Wiedenbeck) 34]:

```
temp := array[j];
array[j] := array[k];
array[k] := temp;
```

Just by examining the above code it is obviously a swap process, swapping two items within an array. Weidenbeck conjectured from her results that this is a strong beacon for suggesting that some kind of sort is being performed in the region of these lines of code.

Once a beacon had been located it would lead to a verification of current hypotheses and result in further refinement of the hierarchical structure. Although, should the beacon be inconsistent with current hypotheses, Brooks conjectured that this would lead the maintainer to generate another

hypothesis that is consistent with the beacon encountered as well as conforming with the hypotheses directly above it (in the hierarchical structure).

Weidenbeck [24] has carried out further studies from Brooks' original theories and claims that (from experimental evidence) beacons give a general high level understanding of the software, but are not sufficient for program comprehension. It should be understood that beacons in themselves do not present a picture, they are merely an aid to the comprehension process, the preceding statement made by Weidenbeck should probably have read *beacons* ***reduce the amount of time needed to construct*** *a general high level understanding of the software.*

In a more recent study, Weidenbeck [34] reports that program comprehension *is a gradual process of assimilation through study resulting in the generation of a 'program overview'.* Weidenbeck also noted that *"programmers do seem to have a tendency to go to the code itself to develop an initial grasp of the program".* It was her suggestion that this initial stage allowed the maintainer to create a simple mental model of the software under consideration. Although the model was far from complete, it formed the start point from which hypotheses could be made, and contained some basic information about elementary goals and operations of the software.

Weidenbeck takes the stance that beacons are not the only key to program comprehension, but certainly are good directives towards understanding. She also claimed that the beacons are related to the concept of programming plans (detailed in the following section), this issue is dealt with in the subsequent section titled *'Discussion'.* Further experiments were carried out to confirm the existence of beacons. In order to achieve this, Weidenbeck presented programmers with several small programs, each with a different mutation performed on the beacon. For example the beacon in one program could be unaffected, whilst in another be hidden by adding complex sequences of code, or possibly removed altogether. The results are summarised below:

- beacon containing programs were understood more often than non-beacon containing programs, even if the programmer had never seen that kind of program before

- the disguise or absence of beacons delayed (and could disrupt) the process of program comprehension

From these findings, Weidenbeck conjectured that beacons were powerful indicators of function, such that the presence of a false beacon had a strong tendency to mislead or misinform programmers about the function of a program. Further to this, she added that beacons may have relative strengths, i.e. strong beacons and weak beacons. Differing beacons would obviously lead to dissimilar conclusions

(or hypotheses) made by the maintainer, for instance Weidenbeck cited that strong beacons to a program's function would result in the formation of definite conclusions.

There is, however, one important discovery made by Weidenbeck (aside from beacons) worth noting, that of *initial study*. Weidenbeck observed that the programmers would often study the software source code first, in an attempt to construct a mental model. Programmers simply looked for a familiar entity in the software and based judgements (hypotheses) on this. Familiar entities appeared to be stereotypical code (beacons), which had a considerable power in moulding the initial comprehension.

2.2.1.5 Programming plans

Most research into program comprehension has come under the heading of programming plans [12, 14, 28, 30-33]. The main architects of this technique include Letovsky, Soloway, Johnson and Détienne, as well as other periphery authors. Again, all of the research in this area has been based on empirical data gained from experiments involving real life programmers undertaking maintenance (or understanding) tasks.

Soloway and Ehrlich [32] first described programming plans, in the light that there were two types of knowledge: rules of programming discourse and programming plans. Examining rules of discourse first, Soloway and Ehrlich believed, just as there are discourse rules for conversation, there are such rules for writing software. These rules included statements such as *variable names should reflect their function*, and it would seem that most of the rules are purely common sense. From this stand-point, Soloway and Ehrlich conjectured that *"If programmers do use these rules and expect other programmers to also use these rules, then we would predict that programs that violate these rules should be harder to understand than programs that do not"*. That is to say, that it would seem likely that a programmer already versed in the style techniques used to construct a program would require less effort to comprehend such a program. Soloway and Ehrlich believed that these rules had an important role to play, and when such rules are broken, comprehension time increases dramatically as the maintainer must employ more of their abilities to reason about the source code.

Secondly, programming plans were defined as *fragments that represent stereotypic action sequences in programming, e.g. a RUNNING TOTAL LOOP PLAN.* So plans represented high level concepts, or semantic knowledge, such that plans were not restricted to one particular language. Although to implement such a plan in a specific language would require syntactic knowledge of that language.

To substantiate their claim, Soloway and Ehrlich carried out an experiment, hypothesising that unplan-like programs (unplan-like defined as breaking one or more discourse rules) would be less

easily recalled than plan-like programs. After allowing programmers time to study a program and then recall it (repeating this three times for each program), Soloway and Ehrlich found this to be true in the fact that unplan-like programs were mistakenly recalled as plan-like programs but then altered in due course to reflect their unplan-like structure. He concluded that *"advanced programmers have strong expectations about what programs should look like, and when the expectations are violated - in seemingly innocuous ways - their performance drops drastically"*. Again, more evidence to support the theory that programmers versed in the style techniques in which the program was written, can comprehend that program quicker than someone who is not.

However, it was not to be for another two years until further papers were to be published and a greater understanding of plans was to be introduced. In 1986, Letovsky and Soloway [12] published claiming that the aim of program comprehension is to discover the intentions behind the code. Letovsky and Soloway detailed two concepts, that of:

- **goals** intentions of what the program's function should be
- **plans** the techniques employed to realise programming goals

He believed that the programmer would form a mental model of the program, such that at the top level would be the specification and at the bottom level would be the implementation. The specification would be the goal of the program, and that these goals would then be broken up into sub-goals (in a top-down refinement fashion) until such sub-goals could be represented by actual statements from a programming language.

Plans would then relate directly to this mental model, such that plans were defined to be a *bag of tricks* [(Letovsky) 28] or solutions known to the programmer, i.e. some kind of knowledge base. Letovsky added that some plans are universally known and others depend on the programmers history (i.e. domain knowledge).

Plans are different from algorithms in the fact that plans are composed in complex ways (by interleaving, nesting, merging or mixing) and that algorithms are the result of this process. For example, Soloway [30] cites that *"programmers have a template like structure for reading in a stream of integers, summing them, and stopping when a sentinel value (99999) is input"*. Such a template-like structure is the plan. A plan can be viewed in figure 2.4.

```
initialise a running total
obtain value from the user
if the value is not the sentinel value then
        add the value to the running total
        loop back to obtaining a value from the user
```

Figure 2.4 A programming plan. This is how Soloway believes that programs are stored by programmers. That is to say that actual lines of code are not stored, but concepts or *plans* are. This is an example of a *"SENTINEL-CONTROLLED RUNNING-TOTAL LOOP PLAN"* [30].

Therefore a plan is a way of storing information, a base from which more detailed information can be calculated without actually having to store that more detailed information as well. A good example to demonstrate this would be of a compression tool (e.g. LHA, ZIP or ZOO). For instance, a file of, say, 600K is compressed to 150K using one of the archivers. The decompression process is similar in concept to plans. The compressed file is equivalent to the plan; from a small information base (150K file) the 600K file is created using mathematical formulas performed on the information base. Hence 450K of the information did not have to be stored at all. Just like a plan can be decoded into source code statements, i.e. those statements did not have to be stored. Plans have backing from psychology, the concept of *chunking* information is well understood and documented [12, 30].

Therefore, in this model, program comprehension is the process by which plans and goals in the source code are recognised. These plans and goals then represent an explanation of what the program does and how it is done.

Once the concept of plans and goals is understood, the next obvious questions are how are they formed and what happens when they go wrong? Addressing these points one at a time, Letovsky [28], Détienne [33] and Soloway [30] carried out experiments using the *thinking out loud* protocol described earlier. Their findings indicated that maintainers started out with a high level mental model of the program, often as soon as the program's name (or a short description) was encountered. From this point, successive levels of the mental model would be created, refining the detail using clues from the program source code. Once plans can be applied to all code and all code belonged to a plan, then and only then, would the process be complete and the maintainer have a complete model and therefore a complete understanding of the program's functions and interactions. Letovsky believed that this was achieved because the maintainer *is opportunistic in that he/she uses top down and bottom up techniques as the cues became available.* For example, if the maintainer encountered a familiar piece of code and could substantiate its function, he/she would fill in a piece of the mental model at a lower level. Conversely, if a maintainer had an idea how the program might be structured (i.e. function of subroutines used), then pieces near the top level would be constructed. This process would repeat until the whole model meshed together somewhere in the middle.

Secondly, it is needless to say that things can go wrong. In the process of forming sub-goals (or sub-hypotheses), there are influencing factors that can lead to false hypotheses. These are:

- **Ambiguity** Maintainers often form hypotheses from the name of a variable. If the variable name is misinterpreted, then a false hypothesis is generated. For example, the subroutine GETDB could either mean get a database from somewhere, or get something from the database.
- **No plan** It is possible that a maintainer cannot construct a coherent plan to match the source code that is yet to belong to a plan. In this case, a mistake has been made further up the model and must be corrected by back tracking and re-hypothesising. Soloway [30] termed this floundering, i.e. searching the problem domain.

- **Delocalized plans** A term cited by Letovsky [12], plans may not always be contained in a small region of the program. Such spread out (or delocalized) plans can also lead to misunderstanding. Letovsky found that maintainers tended to make hypotheses based on local code and also noted that documentation accompanying the software did not detail non-local interactions.

In summary, Letovsky believed that the main problem lay in the fact that programmers tended to make plausible but incorrect assumptions about the program code. Only later on, when no plan could explain a section of code, was the programmer forced to back track and re-hypothesise.

The latest research in this area, carried out by Détienne [33] picked up on another important discovery. She observed that, in all cases, one strategy was employed to construct the first mental model, that of *symbolic evaluation*. This is the process of dry-running the program as though the maintainer was the computer executing the program. The maintainer would use this strategy for as long as he/she did not experience an understanding failure. As soon as this occurred, additional different strategies would then be employed (for example, reasoning according to rules of discourse and principles of the task domain, concrete simulation or reasoning with plan constraints; all detailed in [33]).

If, on the other hand, symbolic evaluation succeeded, the maintainers were able to define the different sub-goals (sub-hypotheses) of the program.

Finally, all were in agreement that programmers built up a 'catalogue' of such plans and would use this as a knowledge base from which to solve other problems.

Although this subject is the most widely accepted theory, it does have its critics. Gilmore and Green [14] strongly criticised programming plans claiming that it did make sense that they are a mental language for programming, but believed that they did not generalise to other languages. Furthermore, that plans do not represent the underlying structure of programming problems, and as such, tools should not be designed solely around them.

2.2.2 Theories of understandability

This section details what attributes a program might have that could increase its readability, and therefore understandability. Generally speaking, this research was the fore-father to program comprehension strategies and is now thought of as common place. Nevertheless, here is a brief overview of what is believed to make programs more understandable:

- **Indentation** — Miara *et al* [26] proved, through experimentation, that indenting a program (sometimes referred to as pretty printing) with between two and four spaces aids program comprehension as it highlights the control flow through the program. The absence of indentation would clearly burden the programmer as he/she would have to investigate control flow without a guide.

- **Variable naming** — Schneiderman and Mayer [20] demonstrated that programs with mnemonic variable names were *"statistically significantly easier to comprehend"* as the programmer could use the names of the variables to hypothesise about the function of the source code. Research is still being carried out in this area today by Laitinen [41] under the name of 'natural naming'. Laitinen suggests that names should be descriptive of their function and that naming a variable should follow two key rules: (1) no abbreviations and (2) should follow grammatical rules of natural languages. For example, the variable name *rmesg* should instead be *received_message*. Laitinen further applies his work to function naming suggesting that one way to improve comprehension would be to group the functions together by using object oriented naming. For example, the functions *send_message_to_modem*, *send_character_to_modem* and *disconnect_modem* might be better named as (in respective order) *modem__send_message*, *modem__send_character* and *modem__disconnect*.

- **Control constructs** Brooks [19] noted that the different control constructs used to implement a concept could effect the understandability. For example, it has long been regarded that GOTO's are bad programming practise, and the use of such statements can have adverse effects on the readability of a program. Schneiderman *et al* [37] believed that structured information is easier to understand and, therefore, is important to program comprehension.

- **Modularity** Modular design of software was investigated by Schneiderman and Mayer [20] and they found that modular software was most easily comprehended as opposed to non-modular and random modular. Interestingly, random modularity was the hardest to understand and the authors believed that this indicated the underlying importance of the proper selection of modules. Perhaps, though, the most remarkable discovery was that a second set of results favoured non-modular above both random and modular. Schneiderman and Mayer believed this to be because the participants (students) were subject to different teaching techniques, *"apparently the instructor in [their] course had not emphasised subroutines, and had not required subroutines in homework problems"*. Thus the resulting conclusion was that modularity could help experienced programmers (especially those that have had adequate training), because it allowed the software to be broken up into chunks. These chunks could then be inspected as separate entities and encoded into their mental model as such.

- **Commenting** Early research resulted in the attitude that the more comments the better. However, more recent research by Schneiderman and Mayer [20] suggests that this may not necessarily be the case. Schneiderman and Mayer classed comments into two bands; high level and low level. Low level comments would be statements about one or two lines of source code (e.g. 'add one to counter'), whereas high level comments would indicate the function of several lines of code or, more likely, a whole subroutine (e.g. 'this function calculates the length of a null-terminated string'). They further conjectured that low level comments could be a distraction to the comprehension process because they simply reiterated the function of a statement whose operation would be known to an experienced programmer anyway. The high level commenting, on the other hand, helped the programmers with the

construction of their mental model, as the comments provided more clues about the overall hierarchy of the software. This was later proved as statistically significant by conducting an experiment on students with the two different kinds of commenting.

- **Flow charts** — Flow charts have their supporters and their critics. In summary it would be fair to conclude that flow charts were a hindrance to those not familiar with them, and helpful to those that were knowledgeable about them. For example, see the work of Schneiderman and Mayer [20] and Brooks [42].

Clearly, it can be concluded that these types of aids to comprehension are mainly graphical in the way that they present the code in some acceptable fashion, for example, *blank line insertion* was also researched but is not worthy of mention as it would be implied by the theory of *indentation* (i.e. adding appropriate white spaces to make the source code more readable).

2.2.3 Understanding through visualisation

With the advent of desktop computers powerful enough to manipulate and display graphics, program representation through graphics is playing an ever-increasing role in the maintainer's toolkit. It is widely recognised that such program visualisations improve and simplify the process of understanding [(Roman and Cox) 43, (Price *et al*) 44].

In essence, program visualisation is the communication of information (about a program) through graphic representation. Roman and Cox [43] believed that *"the viewer often can easily reconstruct the original information from the graphical representation"*. In accordance with the adage that *a picture is worth a thousand words*, graphics can be used to express information without laying down too many specifics. The viewer is left free to formulate their own conclusions about the nature of the information being shown to them.

Program visualisation mainly occurs in two forms; control visualisation, data visualisation and sometimes a mixture of both.

Control flow visualisation

This is the most common form of visualisation and the simplest. In fact many programmers and software engineering companies use their own form of control flow visualisation, namely indentation of source code. Although perhaps not considered a form of visualisation, pretty printers are exactly that. For instance it could be argued that any high level language is a visualisation of the

software because the programmer then does not have to read the program as a series of ones and zeros, just as the computer has to. Pretty printers come in the form of simple ones (purely indentation) right through to complex ones which can colour the code, use different fonts, extract a subset of information only or paginate the code and lay it out as though it were a reference manual, for example the project SEE [44].

Data flow visualisation

This is a technique employed to trace and examine data as it flows through a program's life-cycle. More advanced techniques can carry out such examinations whilst the program is running (live), others carry out *post-mortem* inspections [44]. For instance, the BALSA visualisation program [44] could carry out *live* inspection and modification of data and data structures. BALSA would display a graphical representation of an abstract data type (ADT) and modify the drawings as the program executed allowing the viewer to visualise what was happening to the ADT at each stage of the program. Needless to say, faults in operation of the ADT could be picked up on almost immediately.

Mixture

Mixtures of both techniques could manifest themselves in almost any form, but one good and famous illustration is that of the *Sorting Out Sorting* demonstration [44]. It is a method for animating algorithms, such that graphics were used to depict different sorting algorithms in action. Nine different sorts were picked to demonstrate to students their advantages and disadvantages. In order to achieve this a two dimensional graph was used to depict the placement of 2500 randomly generated numbers within an array. A yellow dot on the graph indicated the magnitude of the number (up the y axis) and its position within the array (along the x axis). As the algorithms executed, a green rectangle would mark the number currently under consideration and a red dot would represent a number in its final resting position. Thus at the beginning of the test each sort would have a widespread 'cloud' of yellow dots and a finished representation would consist of a diagonal line of red dots running from bottom left to top right (assuming an ascending order sort). However the demonstration was only available on a thirty minute video as powerful enough computers were unavailable then (the video was made by taking shots from a computer one frame at a time).

Other endeavours have included the use of sound, sometimes referred to as *auralization*. LogoMedia [44] used such an approach to allow the viewer insert *probes* (breakpoints) into a program and then associate Logo instructions with these probes. Two types of probes existed; control and data probes each to be used with control and data flow trapping. Once a probe was encountered, LogoMedia would then execute the instructions associated with it, i.e. play a sound.

In all but the very latest two visualisation systems (Pavane and Zeus [44]) none of them constructed said visualisations in three dimensions. Computers are now powerful enough to manipulate and display three dimensional graphics in real time yet the opportunities that arise from such an obvious step are not being reaped.

2.3 Discussion

The overall intention for program comprehension is to accelerate and/or simplify the process by which a maintainer understands the operation of a piece of software. The majority of this time is spent studying the intent and style of the original programmer and there is an enormous variance between programmers [(Glass) 39]. Styles can be changed (via pretty printers), but intent cannot. It is this intent of the original programmer that provides the maintainer with a problem. With only source code (and sometimes documentation - although this if often useless) to go from, the maintainer must use all clues and cues to formulate those original intentions.

Now that a goal has been established, how does the maintainer set about achieving it? One point that all authors agree upon is the fact that a maintainer constructs a mental model of the software being comprehended. Therefore, once the mental model is complete and accurate, successful modifications can be instituted. This highlights two points; (1) What happens if the source code does not contain enough information, or if it does (2) how does the maintainer go about extracting the correct information? It is impossible for the source code to not contain enough information, as by definition, the source code is the specification for the function of the software. What is more likely is that the information is difficult to locate or extract from the source code (e.g. delocalized plans). This leads on to the second point, how should the maintainer set about extracting the information to construct a mental model?

This part of the process is split up into two key factors: the approach and the extraction method used. The approach can be viewed as the set of rules that determine how the maintainer picks a start point in the source code to search from and how he/she proceeds from that start point to gather information. As mentioned before, there are two basic types: systematic (start at the beginning of the program and execute it as though you were the computer, i.e. follow the data and control flow through the program) or as-needed (start at the statement(s) where the need for modification first manifests itself and then trace a path forwards and/or backwards through the program to attempt to locate the statement(s) which need to be changed in order to eradicate the original need for modification).

Unfortunately, the more robust systematic approach is only useful for small programs because it would take too long to examine large systems potentially containing hundreds of thousands of statements. Hence some kind of as-needed strategy is required (just as Littman *et al* [29] concluded).

The extraction method is the technique employed to acquire the information from the subset of code currently under consideration. As previously identified, authors have provided many differing algorithms for information extraction. However, it would seem almost ridiculous that a maintainer would restrict him/herself to one particular algorithm. Letovsky and Soloway [12] stated that *"the human understander is best viewed as an opportunistic processor capable of exploiting both bottom-up and top-down cues as they become available"*. This is a classic statement and to expand on such a point, it has to be realised that humans (maintainers) are opportunistic: picking up on any cue and acting upon it. You could liken the role of a maintainer ascertaining the intentions behind software to that of a detective examining the evidence to solve a crime. There is no systematic/algorithmic approach to solving a crime, the detective must employ all his powers of deduction to reason about the motives of a crime and hopefully arrive at a successful conclusion. This is precisely what a maintainer must do: the scene of the crime being the source code.

Each of the program comprehension theories detailed above are not separate, as most authors would have you believe, but are parts of the same jigsaw that fit together to provide the maintainer with a means to visualise, and construct a mental model of, the software. To illustrate this concept, the maintainer must locate sections of code to view. He/she can achieve this by employing program slicing and/or as-needed strategies. This eliminates code that is not relevant to the current change parameters (such code is ignored by the maintainer) and also helps with an initial study of the software as, in order to identify irrelevant code, data and control flow must be examined. The maintainer uses the remaining theories, described above, to reason about the function of the source code. Beacons, programming plans, semantic knowledge and visualisation all play a part in identifying the role of the code. Although, as stated earlier, Weidenbeck [34] believed that the concept of programming plans and beacons were related. For example, a beacon could be seen as an indicator of a programming plan, and that in the most probable cases, a particular beacon would always lead to the same plan, especially as a programmer does not change his/her style much. It would seem appropriate to conjecture that programming plans and beacons are part and parcel of the same overall concept of identifying intentions in the code.

Any of the above approaches can be used and it would seem obvious to conjecture that the maintainer would use which ever is the most appropriate to the cues available and then use one or more of the others to aid and/or corroborate his/her conclusion.

Hence information extraction is achieved via various methods, to construct a mental model or visualisation of the software. However, visualisation aids would seem to be advantageous when compared to beacons, plans or other theory as the visualisation is already in pictorial form, i.e. closer to the mental model. Visualisation also has the advantage that it can take place whilst a program is executing, the other methods are purely static analysis only, i.e. the maintainer is left to speculate about the precise operation of source code. Whereas visualisation offers the maintainer an accurate representation. It could almost be viewed as the software telling the maintainer what it is doing!

Even so, as indicated before, most visualisations are still in two dimensions, ignoring the fact that desktop computers can now handle advanced graphics. The opportunity exists to translate whole systems of software into virtual worlds ready to be explored by the maintainer.

3 Virtual Reality

The aim of this chapter is to present an introduction to virtual reality, by exploring what is needed for it, what the problems are with it and to what types of applications it is being used for. It is also the intent of this chapter to dispel some of the myths that have arisen from all the 'hype' that virtual reality has received [(Machover and Tice) 55, (Ellis) 58].

3.1 What is Virtual Reality?

There is yet to exist a definitive definition of virtual reality. In fact, not only is the definition in dispute, so also is the term itself. Other terms in use include *virtual environments, artificial reality* or *cyberspace*. The term virtual reality has most likely caught the imagination of the public because of the hype generated by the media of its unlimited capabilities to create an alternate existence. However, as will be demonstrated later, this is a falsehood in itself.

The expression virtual reality is credited to Jaron Lanier [55], the head of VPL Research. Although he created the phrase in the early 1980's, history must be delved into further in order to locate the origins of virtual reality.

The concept, or at least its underlying technologies, can be traced back to the 1960's [(Pausch) 50, (Machover and Tice) 55, (Wells) 59]. It was Ivan E. Sutherland (dubbed the 'father' of computer graphics [(Earnshaw *et al*) 60]) who, in 1965, first wrote a paper detailing *the ultimate display*. Three years later he published again, demonstrating the very first *Head Mounted Display*. This truly was the beginning of virtual reality.

As well as Jaron Lanier and Ivan Sutherland, there has been one other memorable contribution to virtual reality and that was Morton Heilig who, in 1962 (notably earlier than Sutherland), demonstrated *Sensorama* [(Stone) 56] which involved the showing of a pre-recorded motorcycle trip around New York complete with "fan generated wind and the noise and smells of New York" [(Gigante) 61]. The only factor missing from it was the ability to interact.

However, such technology was extremely expensive, running into millions of pounds, so unsurprisingly the few establishments that did continue this research were the likes of the military and

civil aviation authorities to produce flight simulators, and for many years that was the main propellant of virtual reality [56].

These days, however, the hype would have you believe that virtual reality is the new technology being used in computer games (for example Virtua Racing by SEGA) as well as any other possible use you can think of [(DeFanti *et al*) 48]. To some extent, these claims are true, virtual reality is being used in more applications that you would first imagine. Also, though, much of the hype is false. Virtual reality is beginning to cross the threshold of usability for simple applications [(Ribarsky *et al*) 57], but the technology to support the illusion of a half-way decent reality is many years away yet [48]. In fact, virtual reality is so young that the first British virtual reality conference was only held in 1991 [(Woolley) 63].

So what exactly is virtual reality? It is interesting to see that hardly two publications carry the same definition. Although, in essence they are 'virtually' the same idea. What follows is a select few of the definitions of virtual reality readily available:

> *The illusion of participation in a synthetic environment rather than external observation of such an environment. Virtual reality relies on three-dimensional, stereoscopic, head-tracked displays, hand/body tracking and binaural sound. Virtual reality is an immersive, multi-sensory experience* [(Gigante) 61].

> *Virtual Reality is a way for humans to visualize, manipulate and interact with computers and extremely complex data* [(Aukstakalnis and Blatner) 82].

> *A computer-generated simulation of a three-dimensional environment, in which the user is able to both view and manipulate the contents of that environment* [(Stampe *et al*) 65].

The first definition would lead to the belief that virtual reality requires certain technologies and certain devices, which to some extent is true, although it is more probable that the author wanted to convey their point that virtual reality is simply not any computer game. This is the view of virtual reality 'purists' who believe that for a virtual reality experience, complete immersion is required. Even so, this can be somewhat misleading as well, mainly because complete immersion is not available. Complete immersion would be the suspension of reality by fooling all of the five senses. However, almost no research has been carried out into taste and smell, especially when compared to

the amount of effort currently in place for sight and sound. Interestingly though, touch had much research directed towards it, but little success has emerged.

The other end of the spectrum is that virtual reality can apply to any experience that is not of the real world, generally though, they are referring to computer games. It can be argued that when the role of a character in a computer game is taken on, then 'actual life' is temporarily suspended and the person 'becomes' that character. Many analogies can be (and have been) made, for example the same logic could be applied to books, films or any other form of escapism.

Interestingly, though, there is growing support for the belief that virtual reality could be defined as a type of *human-computer interface* [(Helsel and Roth) 64].

So perhaps then, it would be true to say that virtual reality does not yet have a definition, more that it is a concept of some kind. One possibility is that virtual reality is still far too young for anyone to know where it is heading and therefore some kind of definition would soon be out of date. Nevertheless, the concept of virtual reality can certainly be explored in the light of today's technology and also of what virtual reality would be in a suitably technologically advanced world.

In the 'advanced' scenario, virtual reality would probably be defined as *any experience, for which the user does not have to enhance or forego any of their senses, that is indistinguishable from the real thing (i.e. reality) that is created, not by nature, but by some other person-made device (such as a computer).* This is, of course, inherently ambiguous as to exactly what technology will be involved as any number of new developments could be made in the ever changing growth of the virtual reality industry.

However, today's technology paints a very different picture of what virtual reality can achieve at the moment and is explored in later sections.

3.2 Types of virtual reality

No wonder then that it is difficult to define virtual reality when there are several forms of it in existence. Virtual reality can be sub-divided up into seven broad categories:

Desktop VR / Window on World Systems (WoW)

This form of virtual reality is the cheapest and, unsurprisingly, the most common. It involves the use of everyday high-performance computers or workstations to generate the virtual reality. A monitor then acts as a window onto the world through which the user gets feedback as to

what is going on in the virtual reality. Input is achieved using a keyboard/mouse or other inexpensive input device. An example of this would be any of the cheap virtual reality toolkits available for the PC. One such example is the public domain program REND386 [(Stampe *et al*) 65] which has now been rewritten as VR386 and was the forerunner to Superscape LTD's commercial product *Superscape* [65].

What tends to happen with these systems is that the user can be drawn into the three dimensional world and *mental and emotional immersion* takes place instead [(Robertson *et al*) 51].

Video Mapping

Involves the art of overlaying a user's body shape outline with that of a two dimensional computer or video generated picture. The resultant mix is then projected onto a screen whereupon the user can watch this to obtain information on how to react with the environment in which they have been placed. A demonstration of this technique can be found on the children's satellite channel 'Nickelodeon' which employs a Commodore Amiga to place contestants into a form of video game, on the program 'Nick Arcade' [(Isdale) 45].

Immersive VR

This is state-of-the-art virtual reality involving the most advanced technology and the biggest budgets! The whole aim of this type of virtual reality is to 'immerse' the user into some kind of virtual environment by employing such gadgets as a head-mounted stereoscopic display, which usually includes three dimensional audio feedback as well. Optionally a mechanical glove is placed over the user's hand so that the user can 'see' their own hand in the virtual reality and use it to navigate and manipulate virtual objects. Unfortunately, due to cost and/or secrecy, access to such systems is strictly limited to the odd conference or exhibition.

CAVE

Standing for 'Cave Automatic Virtual Environment' [(DeFanti *et al*) 48], this is a type of Immersive VR that explores the use of a room that is made from large screens, onto which the virtual reality is projected. Generally speaking this involves the use of four projectors, displaying pictures on three sides of the 'room' and on the floor. Users then stand in the room needing only a pair of LCD shutter glasses which give the projected objects the illusion of depth. Notably, this has the added advantage of being able to cope with multiple users, although current technology can only support one or two users comfortably, before the screen updates become too slow, breaking the illusion. An early example of this was the 'Closet Cathedral' which gave the impression of a large virtual area in a small physical space [45]. However, perhaps a more famous, fictitious, example of where this technology is leading, is the 'Holodeck' found aboard the *U.S.S. Enterprise D* in the television series "Star Trek: The Next Generation".

Telepresence

Although this is considered a form of virtual reality it could also be termed a spin-off or even an application of virtual reality, none-the-less it is included here for completeness. In essence this is the process of linking a user with a remote environment in which another real world object exists. It could be conjectured that this is the ultimate form of 'remote control'. Sensors attached to the remote object relay information back to the operator who then uses this information to make decisions about how to control the remote object in the remote environment. An example of this would be the use of robots by the bomb diffusal squad. Instead of sending in people, a remote controlled robot with cameras attached is used to diffuse a bomb, with a human operator manipulating the robot from a safe distance.

Mixed Reality

Simply, this is the combination of Telepresence and Immersive VR to the effect that computer generated images are overlaid onto what the user is really seeing in the real world. This is gaining much popularity of its own and can also be known as 'computer augmented environments'. The classic example of this would be the fighter pilot having a computer generated map being overlaid onto his vision (via sophisticated cockpit displays or from within his own helmet) to aid navigation or tactical manoeuvres.

Fish Tank VR

This is the extension of 'Desktop VR' in the true meaning of the word. Extra peripherals are added to the hardware and a little more functionality to the software to cope with the new hardware. The net result being the illusion of depth is then perceived by the user as well as some kind of head tracking device that updates the display accordingly allowing the user to look around in three dimensions simply by moving their head.

3.3 Virtual reality technology today

This section details some of the technologies and hardware behind what is conceptually virtual reality today [see Gigante 62]. Computer programs can be summarised as *input-process-output*. Virtual reality is no different.

3.2.1 Input devices

In order for a user to interact with a virtual reality they must be able to indicate their wishes to the virtual reality generator. There are several ways of communicating with a virtual reality:

Joysticks, mice and keyboards

Simple and crude they might be when used for three dimensional input they should, however, not be overlooked mainly because it is what most users will experience as other technologies involve a huge economic indulgence. Naturally they are two dimensional input devices but clever programming can go some way to overcoming this.

Spaceballs

A futuristic name for a futuristic input device, the 'spaceball' is an input device that has six degrees of freedom, namely up/down, left/right, forwards/backwards, yaw, pitch and roll. In essence it is a sphere on the end of a stick that can detect movements of the user holding the sphere in any of the previously mentioned directions.

Data Glove

There have been two main contenders in this arena. Mattel's 'Powerglove', intended to be used with console games and VPL's 'Dataglove', somewhat more state-of-the-art. The glove, as it suggests, is to be worn by the user. The glove consists of optical fibres running the length of the glove. A light source is projected down the fibres and as the hand moves into different positions, the amount of light reaching the other end of the fibre contrasts indicating the current overall resting position of the hand. This information is then relayed to the computer which then displays a computer generated hand within the virtual reality. In effect the user 'believes' that they can see their own hand. Then it is simply a case of using different gestures to indicate intentions. For example extending a fore-finger and pointing might indicate that the user wishes to 'move' in that direction.

Tracking devices

Another form of input, and the one that requires the least virtual reality generator knowledge, is the tracking device. This usually involves the user having to wear a special item that either sends information or is used to calculate information about where and in which direction the user is facing. The most common types of tracking are mechanical, magnetic, ultrasonic and optical. To illustrate one of these; *"Mechanical systems actually attach the user's head to a boom arm or pulley system, and rely on measurement of joint angles or cable lengths to determine position and orientation"* [(Stampe *et al*) 65].

3.2.2 Processing devices

The processing portion of any application can be considered the heart of the matter. Usually it is software which accepts data from the input devices, manipulates it in some way and then passes the result onto the output devices. However, in some special cases, as is with virtual reality, the processing also requires hardware designed specifically for the purpose.

The hardware fits into two categories: hardware that carries out software functions but at a much faster speed (for example a graphics card for a PC) and hardware that is used as a network to pass information between processing devices. An example of this would be from the CAVE where the four separate projectors need to be synchronised in what they display otherwise notable disjoints will be visible at screen intersections, thus spoiling the illusion of virtual reality. Therefore a specialised extremely high speed network link was built between the four projectors so that they could 'communicate' between themselves in order to prevent such a situation occurring.

Software plays the most important role, though, as it is the decision maker. It decides how objects within the virtual reality react to actions made by the user. It is quite important for this software to be 'correct', because if objects react in a completely unexpected way, the illusion could be lost. In order to be 'correct' the software will usually incorporate a database or knowledge-base of some kind that would describe objects and their attributes, allowing the software to make its decisions.

3.2.3 Output devices

Once the processing is complete, the resultant information must be conveyed back to the user so that they see what their action has produced and then the cycle begins again, that is, the user will then react to that reaction and so on, giving the impression of interaction. What follows is a description of the currently available output, or 'feedback', devices.

Visual feedback

Currently viewed as the most important sense to generate feedback for, most advances in virtual reality have been in this field. The most simple form of visual feedback is to use a standard computer monitor which is used as a window onto the virtual reality. Naturally enough, there is no perception of depth, but a slight illusion can be created as the user moves around as objects further away will move more slowly than objects near by.

However, by adding an extra peripheral (usually to be worn by the user) such as Liquid Crystal Display (LCD) Shutter Glasses or Fresnel Viewers, a monitor can be used to create 'stereoscopic' vision and depth is perceived. This works by the computer generating two pictures, one for the right eye and one for the left. It is the job of the added hardware to make sure that each eye only receives the view intended for it and that the other eye is unaware of this. In the case of the LCD Shutter Glasses, the computer will generate a picture for one eye (in this example, the right eye) and display it on the screen. At the same time, the Shutter Glasses have blocked the vision from the left eye, so it sees nothing, but the right eye sees a picture. Then the reverse happens, the computer generates a picture for the left eye, the left eye's vision is restored and the right eye's vision is blocked. As long as this happens quickly enough (about 60 frames per second; 30 for each eye) the

user is completely unaware that this is happening and the result is stereo vision. CAVE uses this technique as well, except it uses large screen projectors instead of a monitor, to give the feel of 'being there'.

State-of-the-art in vision perception is the Head Mounted Display, often shortened to HMD. In essence, this turns the Shutter Glass effect on it's head, as instead of using one monitor to generate two views, HMD's use two monitors to create 'one' view. Although the term one view is used loosely as, of course, two views are still generated, they are generated in parallel and the brain mixes them so the user only perceives one display in front of them. This option results in the best stereoscopic effect that can be achieved today. This technology is often used in virtual reality as not only does it render displays, it can be combined with input devices (tracking) and other output devices (sound) to create an 'all-in-one' virtual reality experience. The main problem is that the user ends up feeling like they are wearing a 'heavy hat'! Although, some alternatives have been put forward, such as the BOOM which, in essence, is a head mounted display suspended from a weighted boom, resulting in a 'weightless' HMD leaving the user to move freely. Incidentally, this technology also offers the advantage of being able to send back information to the virtual reality generator as to what the user's point of view is.

Audio feedback

A number of virtual reality applications do not bother generating sound, concentrating more on the display and input techniques. However, as the film industry has found, sound plays an important role in the illusion of being in a three dimensional space. Audio runs a close second to vision.

Again depending on the type of hardware available, audio feedback can be generated in a number of ways and can be affected by a number of attributes. Firstly, the sound must be of audible quality. All sounds produced by a computer are digital, and that digital representation must be good enough to sustain the illusion of it being the real thing. This is dependent on the 'sampling rate'. Compact Discs store digital information, in fact they are just one long sample of an analogue source. Compact Discs (CD's) have a sampling rate of 44.1KHz, that is they take a reading of the analogue source 44,100 times per second. This is sufficient for humans, as they can only perceive sound up to about 22KHz. So, as a sampling rate drops, so does the quality. If the sampling rate drops too low, then only very monotone bass tones are heard and would only be good for sampling someone snoring!

Once a decent sampling rate has been chosen, there are four ways in which the sound can be conveyed to the user. One way is to play it through a mono speaker, often found in PC's. This has little effect over associating a sound with an action, no distance or direction is perceived. The next alternative is to use stereo, either from a pair of speakers or headphones. This of course, only gives

the illusion of left/right. However, some work was done to 'move' sound beyond the speaker to fool the brain into thinking that sound could come from 'over your shoulder'. Although much interest was expressed in it as the time, currently it is only available on a handful of music CD's and the odd CD based computer game.

The third option is that now used by the film industry, namely some form of Digital Surround Sound. Many systems are available for use by the film and video industry and computers can employ the same techniques to generate sound 'all around you' by placing speakers in the appropriate places. Some of the more sophisticated home cinemas available in retail outlets can handle up to 32 different channels. The disadvantage being that it does involve a considerable cost.

State-of-the-art in three dimensional sound generation relies on a pair of headphones and a technique known as 'convolution'. The human ear can pinpoint the origins of sound by using such information as how loud the sound is and the difference in time between the sound arriving at each ear. Convolution is the involved mathematical process of modifying digital sounds and then playing them through the headphones in order to fool the brain into thinking that the sound has emanated somewhere from within a three dimensional space. The results that have been achieved are exceptional, in that users of the system *can actually point to the specific spot each sound is coming from* [(Stampe *et al*) 65].

The other senses

As mentioned before not much attention has been paid to the other senses taste, smell and touch. Little research has been carried out on digital tastes or digital smells, mainly because they are of much less importance to a virtual reality than the other senses. For example, taste is only used when you wish to eat, and that is very unlikely to happen in a virtual reality.

Touch, on the other hand, has met with some extensive research but been less successful in its findings. Some avenues explored have been to add pressure effectors on data gloves, but the time between the user touching something in the virtual reality and the pressure effectors responding was, simply, unacceptably long. The other main problem that was encountered was in making, say, an orange feel different from a glass sphere. The technology simply does not exist yet, to fool one of the most sensitive parts of the body, the hand.

3.4 Problems of virtual reality

Inherently, the problems facing virtual reality today, are because of the limiting power that current technology can offer. Although there is one real problem that came as a surprise to the virtual reality industry, namely *sim-sickness*.

Sim-sickness

Simulation sickness is the term given to a user who suffers ill effects after using virtual reality. These ill effects can manifest themselves in different forms, but generally users will become dizzy and feel nauseous and in some cases suffer from disorientation. To warn users of this effect most simulators have a sign displaying words to the effect of *"patrons are advised not to drive or operate machinery within two hours of using this simulator"*.

Research reveals two main causes for sim-sickness. The first is in the creation of the illusion of depth. A monitor is a flat, two dimensional piece of hardware being used to generate three dimensional environments. Many psychology institutes have indicated the problem lies in the fact that as a user perceives something to be further away, they try to shift the focus of their eyes accordingly, However, in reality, the object is still the same distance away, on the monitor. This then confuses the brain.

In fact, all of sim-sickness can be traced to the brain becoming confused, that is receiving conflicting information about the surroundings through different senses. Indeed the other main cause of sim-sickness is in the delay time between a user indicating an input (such as looking to the right) and the computer being able to generate and display the new view quickly enough [(Pausch) 50], often referred to as *Transport delay* [(Moshell) 52]. This also may lead to confusion and frustration on part of the user, as if the display is not updated quickly enough, the user may conjecture that the computer has not received or ignored the input and the user will try again. The net effect is that the user will see what they did a few moments ago. One analogy to this would be examining the sun from Earth. The sun is seen, by people on Earth, as it was nine minutes ago, so (theoretically) if something happened on the sun, people on Earth would not react until nine minutes later, and our reaction would not be seen by anybody within the proximity of the sun until nine minutes after that, resulting in an 18 minute action-response time. Of course, computers are not that slow, but it can demonstrate the problem at hand.

Interestingly though, CAVE systems do not seem to suffer from sim-sickness as much as other forms. One theory is that displays at a distance of three metres seem to minimise the risk of sim-sickness.

Limiting factors in virtual reality generation

Most of the problems associated with virtual reality are the hardware components that are simply not up to the job. Virtual reality is beginning to cross the threshold of usability in simple applications, but where more detail is needed or many objects are required (in say the design of an aircraft) the updates to the screen would be intolerably slow.

Currently, given enough time, computers can generate very sophisticated and detailed pictures from a few pieces of information, such as where an object is located and what light sources are available and so on. The problem for virtual reality is that these pictures must be generated at rates of 25 or more per second to keep the illusion of movement, otherwise the experience is lost completely.

As this is the 'fixed cost' so to speak, everything else must revolve around this. Objects in a virtual reality are made from one or more polygons. Needless to say, the more polygons used, the more detailed and realistic the experience becomes. In order to adhere to the 25 frames per second rule, today's desktop technology can handle a few thousand polygons [(Pausch) 50], but to be of real use, technology will have to cope with millions, and in some cases billions, of polygons.

Other factors that reduce the frame rate are the actual dimensions of the screen, not in physical terms but in 'computer' terms. That is, the resolution of the screen, namely how many pixels are used to draw the display and how many colours are allowed. All of this places extra strain on the processor. This is of more importance than would first be recognised as most virtual reality systems have such poor display resolutions that a user is considered legally blind by US law [50]. The average vision ability of a human is 20/20. State-of-the-art display devices that are monitor based can offer 20/110 at best and CAVE systems can offer anything up to 20/40 vision [50].

However, as technology moves on and hardware becomes cheaper these problems will become less and less significant as users will have access to far more capable systems.

3.5 Aspirations and applications of VR

Once it has been established what the technologies behind virtual reality are, what are the aspirations and real world problems being solved by it? Perhaps the most likely use of virtual reality

to jump to mind would be that of the next generation of computer games finding their way into consumer's homes along with all the hype generated by the media. Is this what virtual reality is all about? Will it change the way people socialise? Not for a long time yet, that is for certain, but there are other, in fact, many other applications employing virtual reality technology.

3.5.1 Applications

What is virtual reality being used for? Virtual reality has its virtual fingers in many virtual pots, and by no means is what follows an exhaustive list (see also the works of Encarnação *et al* [67] and Kahaner [68]).

Design

Virtual reality is saving time in the order of months and money in the order of millions of pounds. Many companies are intrigued by the idea of constructing prototypes in virtual reality as opposed to physically creating it in the real world. One example is the design of mechanical diggers [(Adam) 47]. The main aim is to test the visibility of the cab design. This is almost impossible using conventional CAD programs, but with virtual reality, cab designs can be tested and evaluated within two to three minutes. This resulted in a design being selected nine months earlier than usual and saved much cost in not having to build other faulty prototypes, which were rejected at the virtual reality evaluation stage.

Another classic example is that of the Hubble telescope [(Hancock) 49]. When the Hubble telescope went into space it had defects in its mirrors and this resulted in the quality of the pictures being about the same as an Earth bound observatory telescope. This of course meant that the whole effort was a waste of time and an incredible embarrassment. The National Aeronautics and Space Administration (NASA) needed to get the telescope fixed in order to curb its embarrassment and other problems likely to arise. However, they needed to get it right first time, as cost was of major importance. *"In taking steps to prevent a failure, NASA realized that a computerized 'virtual prototype' could identify potential collision problems months in advance of the installation of actual flight hardware"* [49]. The significance of a correct design was immeasurable. Not only would they have to try again, but they would have to foot the bill for another space flight.

Virtual reality came to the rescue as a model of the Hubble telescope was created, with the new additions modelled as well. Virtual reality allowed the designers to 'fit' the upgrades to Hubble, with the added advantage of being able to view from any angle and any location. This was important in itself, as there were many places in Hubble that would be considered 'blind-spots' as cameras would have been too large to be placed in those vantage points. With these extra views, and collision detection abilities, the 'Hubble fix' was redesigned as serious faults were discovered. Unsurprisingly

this ended in the 'Hubble fix' working first time, potentially saving NASA millions of dollars. Perhaps a more enlightening discovery though, was when one scientist noted that:

> *"One of the key lessons we learned from developing this application is that virtual reality need not be done at the multi-million dollar level to realize useful engineering results"* [(Hancock) 49].

Medicine

A great deal of research has been directed towards using virtual reality in the medical industry. Examples include using virtual reality to train doctors in surgery, so they can practice on virtual patients, or at least virtual parts of patients. A slightly different view on this is the work being done to use virtual reality as a type of physical therapy. Supporters of this theory believe that virtual reality could be used to re-teach basic skills to patients that have lost them, e.g. eating. A good analogy of this was described by Jaron Lanier [55] who outlined a situation where an ordinary person could learn to juggle in virtual reality, by slowing the balls down. By this means the balls would take longer to get from one hand to the next allowing the user more time to react. Then as the user becomes more competent, ball speed can be increased in small steps until the user can juggle in real time. So what has happened here is virtual reality has conquered the everyday restriction of gravity, not to mention time.

Another use, is that of the computer-augmented display. Researchers at the University of North Carolina [(Adam) 47] are working on projecting three dimensional views of the insides of a person's body, superimposed onto the body, allowing the doctor to see inside.

Data visualisation

Virtual reality, in its simplest form - three dimensional graphics, is being used to present data allowing great simplification. Today, the world is in a state of information overload. Most technologies are trying to reduce or limit the amount of data that has to be inspected to gain a result. Three dimensional graphics can be used to represent complex sets of data, but virtual reality can extend this allowing the user to interact and animate the data, thus gaining even further insight into what the data could be demonstrating.

One such example is that being employed by the NASA Virtual Wind Tunnel [(Gigante) 61]. The wind tunnel does not actually exist, except in the memory of a computer. Detailed shapes of aeroplanes, shuttles, or anything really can be placed in the wind tunnel. Virtual smoke is then used to demonstrate the flow of air around the object. The benefits are substantial because the user can view the air flow from any point allowing areas of *instability, separation of the flow from the object's*

surface, or any other interesting phenomena [61] to be seen. Another major benefit is the fact that any air flow pattern not in view does not have to be calculated, which results in real time interaction with a phenomenon that would normally require an order of magnitude of more computing power.

British Telecommunications Plc [(Walker *et al*) 54] are also exploring the use of virtual reality to visualise their telephone network. One example of work carried out was when virtual reality was used to overlay and animate data from lightening strikes and network failures across Britain. The masses of data involved meant that paper analysis was almost impossible, but after a single virtual reality demonstration, trends and correlations could be picked out immediately.

One interesting project the US Army is working on is being able to visually replay the contents of a black box recording so that they can view the last moments from any viewpoint in or out of the aeroplane [(Moshell) 52].

Entertainment

The entertainment sector of the economy is estimated to be the growing the fastest [(Stampe *et al*) 65]. Unsurprisingly, in the short term virtual reality technology will be mainly put to use to bring the next generation of entertainment to users. Also the media is most interested in the leisure uses of virtual reality because that is what catches the public's imagination and sells newspapers; the promise of being able to "*...stand on the bridge of the starship Enterprise and talk intergalactic strategy with Captain Kirk and Mr Spock*" [(Kay) 66].

Examples of what is available today can be found in the local arcades. W Industries' Virtuality has produced at least two arcade machines: Virtua Racing and Virtua Fighter. Virtua Racing involves driving a racing car on one of three tracks. The action is fast, with arguably the best graphics to date. What makes this game different is the fact that the action can be viewed from any angle (by the software), although the actual arcade machine only lets the user choose between four: in cockpit, two behind the car, and one above looking down.

State-of-the-art entertainment can be found in Chicago under the name of *Battletech*. This involves the use of 16 machines (8 machines per team) in which a single user is contained. The machines are meant to resemble tanks. Inside they provide screens onto the virtual reality, have an array of fire power capabilities as well as radio links between the machines for the teams to co-ordinate their 'attack'. In essence it is a battlefield simulation that is extremely realistic [(Grimes) 53].

Television is increasingly employing virtual reality technology to enhance its programs. For example, the British Broadcasting Corporation's (BBC's) 9 O'clock News sports a virtual studio -

only 10% of what the viewer sees is real. Babylon 5's graphics are all created using a Commodore Amiga A4000 and a 'video toaster' [(Stone) 56].

Architectural visualisation

Architects are also using the technology to produce virtual mock-ups of what buildings will look like before they even have their foundations dug. Technology now allows for subtle light shading and other effects to be included so that a potential house buyer could 'walk through' his or her house and get a feel for what it would be like [(Gigante) 61].

Education and Training

Virtual reality offers all the benefits of the real world but with none of the dangers. The most widely utilised virtual reality training is in the form of a simulator. Pilots, for example, notch up many hours simulated flight before they can fly the real thing. Not only does this save the company money (the plane that would have been used to train the pilot is now available for fare paying passengers) but it is also much safer. If the pilot makes a mistake in the simulator, he/she is more likely to survive than if in a real aeroplane!

Stampe *et al* [65] summed it up when they stated that *"Giving people hands on experience is always better then trying to describe things in words"*. Virtual reality is more than a picture saying a thousand words, it's speaking volumes.

3.5.2 Aspirations

The most common consensus among publishers in the virtual reality community is that the aim of virtual reality is to *"bridge the gap between how we interact with a computer and how we interact with the real world"* [(Wellner *et al*) 46]. There exist many other such paraphrases all along the same lines. The main point wanting to be conveyed is that of virtual reality being the next generation of human-computer interface. No longer will humans have to adapt themselves to interact with a computer, the computer is adapting itself to work with them [(Machover and Tice) 55].

Ironically, it would seem that the computer is coming full circle. When computer technology was in its infancy, they would take up entire rooms, even buildings, such that the users of the computers would be surrounded by it. That is to say that the computer became their environment. Computers have, inevitably, got smaller, but now this new emerging technology is fighting its way back from whence it came. Once again the computer will become the environment surrounding the user.

3.6 The virtual reality concept, a definition?

So far, virtual reality is not a singularly defined entity. It can be a lot of things, whilst at the same time be only a few. Especially with today's technology, virtual reality could certainly be viewed as the tricking of only two senses (vision and hearing) to sustain the belief that a user is in an alternate environment. No-one can really say, but to some, virtual reality is a defined set of technologies, such as head-mounted stereoscopic display, a Silicon Graphics Workstation and data glove. Whereas others would conjecture that something as simple as *Frontier: Elite II* (a computer game available for most PC's) is a virtual reality as the user takes on the role of a space trader, can interact with many things within the game and all objects are presented in three dimensions [(Braben) 76].

The line has to be drawn somewhere, or does it? Could it not be considered that such computer games are a very basic form of virtual reality and complete immersive systems are the current state-of-the-art virtual realities? No doubt, in years to come what is considered state-of-the-art virtual reality today will become the 'computer game' of virtual reality tomorrow.

Aukstakalnis and Blatner's [82] definition (*"Virtual Reality is a way for humans to visualize, manipulate and interact with computers and extremely complex data"*) of virtual reality is important to this work, because this is exactly what is intended to be done; to visualise complex data, i.e. programs. This is echoed in the section on *Data visualisation*, that complex sets of data can be presented in such a way that the load of information is reduced to a level acceptable by the maintainer. In order to facilitate this, the survey of virtual reality systems indicated that *Desktop Virtual Reality* is the best choice, especially as this would allow the visualisations to be widely available as it does not require the use of special hardware beyond that of a normal PC.

4 Visualising Spreadsheets

The aim of this chapter is to bring together the notions presented in the previous two chapters (chapters two and three) and to indicate how the technologies involved in virtual reality can be applied to program comprehension.

However, as a complete program understanding system is beyond the scope of this research, spreadsheets are introduced as a substitute for programming languages. Spreadsheets offer the same characteristics and share many of the same problems as a small programming language would have. To this end, this chapter explores how virtual reality technology can be applied to the problems associated with spreadsheet maintenance and, as a side-effect, spreadsheet construction and debugging.

To end this chapter, the theories presented below will be examined to see how they can be extended to program understanding.

4.1 Reasons for using spreadsheets

As stated above, to successfully visualise an entire programming language is beyond the scope of this research, it would be more the goal of longer term research. For example, longer term projects include creating dependency graphs for C language [75] programs [(Kinloch) 69], which is only the bare bones of visualisation, so to use virtual reality to allow a maintainer to walk around software would conceivably take even longer.

Therefore, in order to be able to complete this work, a smaller language to analyse is required. To this end, spreadsheets were chosen as an ideal candidate, as they share many properties with programming languages, for example cell referencing and formulas in spreadsheets are identical to procedure calling and data passing in procedural programming languages. The formula network is a dependency graph which depicts which cells are reliant on which other cells. It is conjectured that by being able to visualise and trace these dependencies, the function of the spreadsheet should become more apparent.

In essence, programs and their underlying code are very complex, so the aim here is to understand something simpler: spreadsheets.

4.1.1 Aspects of spreadsheets

Hendry and Green's [9] aim was to see if they could produce a tool that would help users of spreadsheets to *"make it easier to describe a spreadsheet to a co-worker"*. To facilitate this they needed to see how the spreadsheet display could be brought closer to the users own *cognitive map* of the spreadsheet. Hendry and Green believed that the cognitive map was *"the way in which the representation can suggest more or less accurately its* [the spreadsheet's] *inner workings"*. The ultimate aim of their research being to apply their results to newer languages still in the pipeline. Which in effect is also the ultimate aim for this work, namely to extend the ideas presented here to program comprehension.

Pointing out that the spreadsheet is split into two levels; the problem domain level (the 'what') and the computational level (the 'how'), Hendry and Green suggested that the spreadsheet's own display does not provide an adequate cognitive map of what it does and how it does it. Further to this they added that *"Programmers know far more about their programs than can be expressed directly in the code"* and that a parallel could directly be drawn to spreadsheets as well, i.e. spreadsheet authors know far more about their spreadsheets than can be expressed directly in the formulas. It would then be up to the spreadsheet reader to obtain some of this *metacodistic* knowledge to understand the spreadsheet.

In order to facilitate this, Hendry and Green developed their tool: *CogMap*. The premise behind it is to be able to aid the process of understanding spreadsheets. CogMap attempts to follow the Landauer theory, that is, to understand how a human would undertake a task and then get the computer to mimic this, but at the same time overcoming human limitations. The resulting output from CogMap produced *tags* and *links* to annotate spreadsheets with a heavy usage of colour to convey information to the spreadsheet reader.

Hendry and Green tested their theories on ten experienced spreadsheet users, hypothesising that *"such a simple tool would be an effective communications aid"*. From their interviews with the experienced spreadsheet users, two interesting points were raised:

- The point of comprehensibility was raised with the blame laid at the spreadsheet being two dimensional. It was this limitation that made it hard for one of the spreadsheet users to remember how the whole structure worked. Thus he was unable to predict how the structure would change if he were to modify it.

- Secondly, the authors pointed out that none of the spreadsheet users gave many assertions about the computational domain. That is, assertions were mainly made about the 'what', i.e. what the

data values visible in the spreadsheet meant. Often the data values were grouped somehow, usually into table formats as this is mostly all that a spreadsheet can offer. Hence, one spreadsheet user would point to a region on the spreadsheet and refer to it as the *input data*, whereas another area might be the *output data* and so-on.

As a result of this work, Hendry and Green completed the annotating tool for spreadsheets relying on the extensive use of graphics to convey information about the inner workings. For example, *"The fill patterns indicate the direction used to propagate formulas, and links indicate the blocks that are referenced by the formulas"*. They went on to conclude that:

- *"users do have difficulty when trying to comprehend their spreadsheets"*
- *"creating spreadsheets is easy, but understanding them afterwards is hard - even one's own"*
- *"Our adage is that designers need simple ways to externalise what they know about their designs"*

The aim of Hendry and Green's work is clear, to transfer knowledge of a spreadsheet from one person to another by detailing the inner workings through the use of colour graphics and annotations, because it has been identified as a real problem. This is also the aim of this work, however, Hendry and Green use two dimensional graphics to facilitate their needs. With the advent of faster computers and virtual reality it could be possible to represent the inner workings of a spreadsheet as a virtual world in which the user can freely traverse and interact with. It is this key factor of being able to explore that allows the user to construct their own ideas and representations of what is going on in this world, and by understanding this world they understand more about the spreadsheet.

This can be ascertained and likened to adventure games on home computers, where the user travels around a world learning about it and interacting with it, thus gaining an understanding of it. When the user starts he/she has no knowledge of the world, but with time each user can become expert in the ways of the world. A classic example of this is the game *Frontier: Elite II* [(Braben) 76]. A very popular space trading game based in a virtual environment using three dimensional graphics. The player can roam freely about the galaxy (modelled on our own real galaxy) within certain confines, opening up an almost infinite number of scenarios. Yet players have become very adept at abiding by the rules and being successful because they have gained the knowledge to make them so.

It is this factor that could aid maintainers to understand spreadsheets as well as software. Not forgetting that the ultimate destination of this research is to use virtual reality to represent program code, allowing the maintainer to roam around learning and interacting with it and so gaining a better understanding of it.

Although, one advantage that should not be overlooked, is that of fun. It cannot be denied that searching reams of code can be very laborious for the maintainer. There are many tools on the market to aid with program comprehension but they are mainly based on metric output, e.g. cross referencers, module decomposition, dependency analysis and so on. Yet more information to wade through. By letting the maintainer interact with the spreadsheet or program source code, it can be fun and interesting to him/her, thus aiding the process. Psychology tests have been carried on babies, involving placing mobiles above the baby's cot. Some of the mobiles were connected to a pad under the baby's pillow, such that any head movement by the baby had a direct movement result in the mobile. The researchers concluded that that babies without the 'interactive mobiles' soon lost interest in them, whereas the others played happily with theirs for a long time thereafter [70].

Virtual reality literally can offer spreadsheets a third dimension allowing the problem domain level and the computational level to be viewed simultaneously. Of course, this is still limited thinking as this assumes a viewpoint only theory, whereas, the user can change the layout and modify the spreadsheet whilst in virtual reality. This too decreases the time it would take to learn a spreadsheet as proved by Saariluoma and Sajaniemi [71] who stated that:

> *"Spreadsheet calculation causes a heavy memory load, since it is necessary to remember complex cell and calculation systems. Subjects are not able to abstract the deep structure and encoded formula networks. If subjects are able to induce the structure of a formula or a network of connected formulas, they usually learn it fast".*

This is indicative of the type of results hoping to be achieved here. By using virtual reality to provide the means for a user to induce the framework (the how, what and even possibly why) of a spreadsheet, this would decrease the time needed to become familiar with it.

4.1.2 Problems with spreadsheets

Spreadsheets are extremely susceptible to errors. Errors made by humans: garbage in equals garbage out. Hendry and Green [79] noted that *"the most powerful feature of spreadsheets is that people can see what is to be done and simply do it"*. That is, that the people who often use spreadsheets are not specifically trained to do so, they are people with other job roles. For instance, a secretary may be the main user and programmer of a spreadsheet, but she is not specifically trained to write spreadsheets. As spreadsheets are so widely available, powerful and simple most people regard them as easy to use, and therefore do use them. Unfortunately, it soon becomes apparent that they are not as simple to use as it may at first seem, they can contain many pitfalls and traps. For example, the Daily Telegraph [(Bray) 77] carried an article about a spreadsheet investigation and found that more

than 90% showed errors of more than 5% in the output, and 12% were so poor that the results were meaningless! Two further surveys indicated that over 30% contained errors, even in spreadsheets written by 'experts' [(Person) 78]. Other 'astounding' figures to whet the palette include *"One client lost a six-figure sum because of a mistake in a spreadsheet"* [77] and a construction company that lost a quarter of a million dollars because a SUM function did not include the full range [78].

Interestingly enough, it would be thought that the spreadsheet's popularity would in turn see a large number of works detailing the pitfalls of spreadsheet programming. This is not the case, as Hendry and Green [79] stated: *"...it's literature is tiny"*. Why this is the case is simply unknown. As the figures in the above paragraph demonstrate, virtually no spreadsheet is error free.

Nearly all of the errors made in spreadsheets are to do with human error, usually because of a typing mistake, a mis-understanding of what a particular feature actually does or simple carelessness. The remaining mistakes (a very small percentage) are made by the software package itself, due to an error in the software's code. What follows is a brief description of the main types of common human mistakes found in spreadsheets:

- **Typing mistakes**. Whether entering data into a spreadsheet or writing a letter, humans make mistakes when typing. The most common types of typing mistakes can be traced to four keyboard errors: leaving a character out, transposing two characters, typing one character when intending another and adding a character that is not supposed to be there. Not forgetting, of course, that these mistakes can happen in collusion with each other and manifest themselves into virtually undetectable 'compound mistakes'. For example, the formula *=b1+c2* could end up being *=b1+b2* (typing one character when intending another) or *b1+c2* (leaving a character out). The first example creates an incorrect cell reference and the second turns a formula containing cell into a label containing cell.

- **Mis-understanding**. Mis-interpreting the function of a feature can be dangerous. The most common example of this is absolute and relative cell referencing. Often, spreadsheet users do not understand the difference between the two, or they do understand the difference but get them muddled up (i.e. using absolute when meaning to use relative). Absolute cell referencing is when the user specifies a unique address of a cell (e.g. a1). Relative cell referencing is when a cell is referenced by directions from another cell (e.g. left 2, up 3 would reference cell a1 from cell d3 - assuming numbers labelling the horizontal axis). Naturally, this causes errors when cells are copied, moved or deleted, which is one of the most common operations carried out in spreadsheet construction.

- **Carelessness.** Three types of common mistakes were identified in this category, those of: inserting values or formulas in the wrong locations (the user does not have a good understanding of the layout of the spreadsheet), circular references (creating formulas that rely on themselves for input) and incorrect summations (i.e. only summing 11 months worth of data instead of 12).

Each of these easy-to-make mistakes can attribute to the incorrect functioning of a spreadsheet which in turn sheds a great deal of light on to why one of the above mentioned surveys reported over a 90% failure rate.

Although these mistakes are regarded as common and are well known about by experienced users, they are still extremely hard to track down and eradicate from the spreadsheet. Hendry and Green [79] stated that *"informants often have trouble spotting mistakes and debugging formulae"*. Spreadsheets do not offer the ability to see the whole the picture at once, i.e. to visualise the dependency graph within a spreadsheet. In essence, that is exactly what a spreadsheet is, a dependency graph indicating the connections and relations between cells. This work relates directly to the problems incurred by call graphs and dependency graph visualisation in computer programs.

In order to establish the function of a spreadsheet, the user often has to get into the computational domain to examine relationships and discover why particular formulas have been constructed. Hendry and Green pointed out that

> *"Usually, the spreadsheet surface consists of neatly arranged columns, rows and blocks; but this presentation view can hide much of the underlying complexity. Headings, graphic annotation, and policies for spatial arrangement are often employed in an attempt to reveal the underlying structure, but these are often inadequate"* [79].

Furthermore, spreadsheet users have voiced the desire that a facility for ascertaining the underlying structure would be helpful, especially with debugging. Thus it would seem clear that any visualisation of a spreadsheet should be able to concentrate on both the computational and problem domain and allow the user to be able to view both simultaneously.

4.2 Visualising spreadsheets

This aim of this section is to detail the concepts of how spreadsheets can be visualised using virtual reality. Although unlimited resources are assumed, this does not greatly effect the way in which the virtual world is seen or manipulated, what is at the heart of the matter is how the world is

presented to the user. Whether the user is wearing a head-mounted display or using Window-on-World techniques is of little consequence as it is the configuration of the information that is the quintessential factor. For the purposes of future research, the theory presented in this chapter will assume that the user has access to some means of viewing, navigating and manipulating the visualisation.

Selecting a start point

In order to quicken the pace of learning, the first sight of the environment should be as un-disorienting as possible. One thing that is trying to be avoided is to complicate the situation by presenting the maintainer with more information to decode about how to decipher what is being displayed. In essence, the display should be as familiar as possible to the maintainer in order to lessen confusion and cognitive load.

Navigation also provides a problem in that the maintainer should be able to ascertain where he/she is at all times and what it is they are looking at. Especially to start with, the maintainer should be deposited in a position that is akin to the fact that all maintainers, when first assessing a program, start by examining the overall structure of the program and that their attention is then guided by the need for more detailed information about the program.

To facilitate the above two points a display centred around the physical appearance of the spreadsheet would be ideal as opposed to, say, a display orientated around the data processing activities. Not only would this preserve the physical property of the spreadsheet and, hence, create a sense of familiarisation, it would also allow the maintainer to bring any knowledge they may already have about the spreadsheet to the visualisation.

What shapes will be used?

It should be pointed out at this stage that all of the ideas that will be presented in this chapter are providing a strong physical and familiarity link between the spreadsheet and the visualisation in order to reduce the time needed to comprehend. However it should be noted that all humans are different and as such may find it more convenient to use other representations or layouts to portray their spreadsheets when trying to understand them. To this end, the facility should exist for the user to define their own preference settings, for example making a cell appear as an object of their choice, and for the user to be able to alter the layout of the spreadsheet.

What follows is a guide to the 'default' representations chosen to visualise spreadsheets. It can be considered a start point from which the user is able to redefine shapes and colours as well move objects about if necessary. However, from this point on, it will be assumed that the maintainer is happy with the default selection and uses it to visualise spreadsheets.

	A	B
1	Pounds	Dollars
2	10	15.436

Figure 4.1 A simple spreadsheet. This is an example of a simple spreadsheet that takes an arbitrary number input in cell A2 and calculates another number by the means of a formula stored in cell B2. This actual spreadsheet is used to convert Pounds Sterling to Dollars.

One of the quickest ways to learn something is if you are already familiar with it. A plainly obvious statement, and with the idea of trying to keep things as simple as possible the concept of using objects that resembled the physical representation of the spreadsheet came to light. A three dimensional rectangle could be used to represent a cell, with different colours used to represent the function of that cell. There are three types of cell:

- **Label cells**. These are the cells that are not connected to the underlying formula network, but none-the-less contain information about the spreadsheet. An example would be the cells 'Pounds' and 'Dollars' in the demonstration spreadsheet presented in figure 4.1. Although they play no part in the determination of the spreadsheet's output, they do provide the spreadsheet user with a clue as to what the data and underlying formula's purpose is. In order to depict this type of cell, the colour grey was chosen as this colour is often used in graphical user interfaces to depict something that is there but is currently a redundant operation. In essence, that is what a label cell is, it is there, but is redundant in the fact that it does not affect the outcome of the spreadsheet's calculations.

- **Input cells**. This is the cell type that stores data (usually numerical) for use by cells that contain formulas. These are the 'used by' cells. Other cells use the information contained within them for the calculations that are to be performed by the spreadsheet. In the spreadsheet described in figure 4.1, cell A2 is the input cell, as this is the cell that is used to calculate the spreadsheet's output which appears in cell B2. To represent this cell type, the colour green was chosen as this is commonly regarded as a colour to represent 'go', that is, these are the cells that should be used to make the spreadsheet calculation 'go'.

- **Output cells**. These cells represent the output of the spreadsheet's function. Results of formula calculation appear in these cells. In the spreadsheet in figure 4.1, the cell marked B2 is such a cell as it uses the formula associated with it to produce the figure presented in it. In order to represent these cells, the colour white was chosen as a 'display only' type of cell. Unfortunately no particular colour could be found to represent 'output' so white was chosen as it was distinctly different from green and also as white was a bright, vibrant colour as almost as though it were the 'light at the end of the tunnel', that is, the result.

Thus, that is the schema used to represent the different types of cells but there are some features that cells have in common. For more information to be available to the maintainer, the actual information stored by the spreadsheet (labels, data and formula) should be contained within the visualisation as well. To this end, it was conjectured that there should be two ways in which this information should be accessed. Firstly by being able to 'enter' the cell to find the relevant information being displayed on the walls of the cell. For example, in the case of the output cell, the formula would be displayed on one wall, and the current value of that formula being displayed as well. Secondly, there should be some mechanism for the maintainer to be able to swap the information in one or more of the cells from being displayed on the inside walls, to be displaying on the outside walls. This, for example, would then allow the maintainer to see what all the label cells contain without actually seeing the data, or having to enter the cells.

This only leaves the links to be defined. For this, a small pipe object was used as it seemed typical of the way in which links are defined in real life, i.e. a straight line between objects. The link could also provide some further impact on the visualisation by colour coding them as well. A link can be made up from two coloured halves. Essentially, for a link to join two cells together, this means that one cell is using the information contained within the other, therefore, the first cell is *using* the other, whilst the other is being *used by* the first. The colour scheme selected was based on the brightness of the colours. Yellow was chosen to convey the concept of activity or vibrancy, i.e. the cell nearest to the yellow half of the link is the active one (that is, it is the 'user'). Whereas black, meaning passive, is nearest to the cell that is the supplier of the information (that is, it is the 'used'). For example, in figure 4.2 below, the visualisation of the spreadsheet presented earlier (in figure 4.1) is shown.

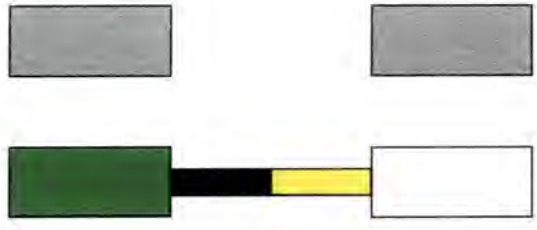


Figure 4.2 A 2D visualisation of the spreadsheet in figure 4.1 (above). For an explanation please see the text below.

Figure 4.2 depicts the colour schemes used to identify cells and their roles within the spreadsheet. Hence, the two cells at the top of the diagram represent cells A1 and B1 which are labels and so are 'greyed out' although their position is still indicated. The green cell is cell A2 (bottom left) and represents where data is fed into the spreadsheet. By examining the colour of the link it can

be ascertained that the green cell is used by (because the black half of the link is nearest to it) the white cell which, of course, is the output cell. This cell in turn can be viewed as using (because the yellow half of the link is nearest to it) the green cell.

Thus the overall layout of the spreadsheet is retained whilst at the same time shedding light onto the underlying connections between the cells. Another advantage is that only the cells that are in use are displayed to the maintainer, whereas looking at a normal spreadsheet, the only clues to cells that are in use is by examining what information is located on the surface of the display.

4.2.1 Making use of the third dimension

In the above example visualisation, the spreadsheet was presented in two dimensions, so why employ virtual reality to provide a third dimension? Although it was only designed to depict the colour schemes used, it does provoke the asking of this question. The case for virtual reality is as follows:

Firstly, virtual reality will allow for more complex spreadsheets to be visualised. Many graph dependency projects produce a two dimensional output that essentially becomes a 'mess'. Modules can be laid out on the page, but when the links are added to depict what module calls another, there can be so many that the links obscure the modules and it becomes increasing difficult to trace the links. Virtual reality allows depth to be utilised to place the links. Whilst the cells remain at one depth, links can be regressed on to other depths allowing for there to be no line crossing what-so-ever. Of course, it may appear from the front that this is the case, but the ability to manoeuvre one's point of view and substantiate a different angle to view from, this quickly eradicates any ambiguity that there might otherwise be.

Secondly, as hinted above, the ability exists for the user to change his or her viewpoint of the spreadsheet. Not only allowing different perspectives to be perceived, it also creates the opportunity to *get into* the spreadsheet and view the rest of the spreadsheet from one particular cell, by invoking the option to make the walls of the cell transparent. The maintainer could go into a cell and count the number of yellow and black links on the walls to ascertain how many other cells are using and are used by this cell. As well as this, the maintainer should have the option to travel along the links, i.e. to be transported from one cell to any other connecting cell. With interaction added this would allow the maintainer to 'travel' with the data. Examples of this are detailed at the end of this chapter in figures 4.4a and 4.4b.

Thirdly, and possibly most usefully, the added option of interactivity and animation. By allowing the maintainer to manipulate the contents of the virtual world, 'what if' situations can be

played out. For example, what if a cell or link was missing, what if the value in this cell was ten, not twenty. This feature also allows the user to change the layout of the spreadsheet to, perhaps, one organised around that data flow, as opposed to the physical representation. Once the maintainer has his/her representation in place, they would be able to 'play' the workings of the spreadsheet. By having a control panel consisting of a traditional directional keypad (play, rewind, fast forward and so on) the spreadsheet calculations can be seen in action, played in slow motion, with a use of graphics to represent data flowing through the spreadsheet. For example, using the spreadsheet in figure 4.1, when the spreadsheet is first visualised, the input cells would contain the appropriate data and as the user clicks on the 'play' button, the input cell would light up to indicate the start of the data's journey and then the highlight, with the data's value displayed in it, would flow along the link to the output cell where the formula would be used to calculate the new value and that value would then be displayed in the output cell. Further to this, as the formula is being used, this would also be displayed to indicate what is happening to the data. An example of this is demonstrated in figure 4.5 (at the end of this chapter).

Fourthly, as stated before, the current state is one of information overload. With extremely large spreadsheets there may be the problem of too many cells. The option to 'prune' the visualisation should be provided. This ties in with the idea of program slicing. The maintainer should be able to select a cell and ask for all cells not connected to its formula network to be removed from the display, allowing the maintainer to focus only on the part of the spreadsheet that interests him/her. A further option to this would include the ability to remove all cells that are not directly linked to a particular cell by 'x' number of links, or to simply remove cells at random from the display with the capability for them to be replaced at any stage. Please refer to figure 4.6 for a demonstration of this.

Finally, for the maintainers work not to have gone to waste, the visualisation should have the power to be able to write out the changes made to the spreadsheet back from the source that it came. In essence this provides the ability to construct spreadsheets in virtual reality (by visualising an empty spreadsheet), debug spreadsheets as well as comprehend them.

So, as a side effect it turns out that this research could in fact lead to a complete virtual reality spreadsheet construction kit, which in turn would be the first stepping stone to program construction kit where, conceivably, a programmer could design a program without even touching a keyboard.

4.3 Advantages of Virtual Reality visualisation

What are the advantages of the ideas expressed above? How can they be used to aid in the process of comprehension? The aim of this section is to explore how these concepts can help by examining how the visualisation can be employed to facilitate initial comprehension.

4.3.1 Overall Comprehension

When the maintainer is first placed in the visualisation, the aim is to give a general impression of the overall layout of the spreadsheet, including physical properties and underlying dataflow network. As Hendry and Green [9] stated earlier, spreadsheet users normally refer to areas of the spreadsheet as input or output. These areas can be recognised by the colour of the cell and the direction of the data flow within the spreadsheet. Dataflow can be examined by looking at the links between the cells. Data flows along a link from black to yellow.

Figure 4.3 Example visualisation of a spreadsheet.

For example, viewing figure 4.3 it can be seen that there are a bank of input cells in the middle of the spreadsheet which are connected to three output cells at the bottom of the spreadsheet. Data flows from the input cells to the output cells, indicating that the input cells are recording information that is being used to create a single piece of data below (perhaps a total). Each of those 'totals' are then used in the calculation of another cell over in the bottom right hand corner. Finally that cell is then used to pass information on to the cell above it. The grey cells, representing the labels, surround the spreadsheet possibly indicating that the input area is some kind of table and that the information on the right hand side is probably some inferences of that data, like overall total or average per column. Overall, certain workings of the spreadsheet can be identified, such as data flows from the table of input cells, 'down' the spreadsheet, 'along' to the right, and then back 'up' again.

4.3.2 Further 'side-effect' advantages

As mentioned in chapter two, program comprehension is a skill required at many stages of the software life cycle, whether it be software maintenance, testing (debugging), software re-use or software documentation. Therefore, any tool that can aid with program comprehension can be of use to any of the above-mentioned stages. The same applies to spreadsheets. The visualisations could also be used to debug spreadsheets, help document them or aid with locating re-usable parts of spreadsheets as well as general understanding.

Another advantage of the visualisation outlined is the fact that data can be manipulated and 'played' forwards, backwards and in slow motion to see how the data was formed at every step. Spreadsheets offer the ability to allow varying input data which affects the output data. However, with the ability to 'play' data backwards, the visualisation could be used to alter the output data to the desired value and enquire as to what input data is required to achieve this.

Adding up the visualisations ability to manipulate data and its ability to write back changes made to the spreadsheet, the visualisation can be seen as a complete spreadsheet maintenance kit. That is, the actual spreadsheet package that was used to construct the spreadsheet, is no longer required. Now, design could be seen as the maintenance of 'nothing', i.e. maintaining an empty spreadsheet. Bearing this in mind, the visualisation could now be said to be a complete virtual reality spreadsheet construction kit. Inevitably this visualisation technique could replace the conventional spreadsheet package, as the spreadsheet user will be able to see the computational domain at the same time as the problem domain - which, currently, is unusual in normal spreadsheet packages.

However, it would probably be more sensible to have the two concepts run side by side, allowing the spreadsheet user a choice. Especially at the present time, as the virtual reality construction kit would be much slower than a conventional spreadsheet package.

4.4 Extension to Program Comprehension

The subject matter thus far has mainly been concentrating on spreadsheets, but one of the main aims of this work is to provide a grounding to programming language visualisations. In order to achieve this the concepts presented here must, in some way, be applied to program comprehension. Source code is similar to spreadsheets, in that cell references and formulas are just like procedure calls and dataflow, so many of the ideas here can be transcribed straight over. For example links can be used to demonstrate calls between modules, and modules could be represented as the cell is, in spreadsheets.

This is, of course, just an example but the physical representation of the visualisation is of little importance. What is required is for the maintainer to be presented with minimal information about the structure of the software and for him/her to be guided by their requirement for knowledge. Many of the authors quoted in chapter two agree that this type of approach is desired and undertaken when in the process of understanding programs.

To illustrate this idea, a maintainer could be presented with an overall structural visualisation of the software depicting the modules used, and which modules call routines from other modules. Then, as the maintainer begins to navigate, say towards a particular module, more information could be displayed, such as the module's name, its size and percentage of activity within the system. As the maintainer approached the module, as though to enter it, further visualisations could be introduced to depict the number of procedures within the module and the links could be refined to indicate which procedures were called by other modules, or indeed by other procedures within the same module. Such refinements could take place until the maintainer reaches the source code level.

Then there is the idea of interactivity and animation. The ability to be able to send data around the model and watch what happens to it. The ability would exist to 'cut' links to see what happens to the data if it were not passed through a particular procedure, module or even (at the most detailed level) statement. Also to watch data being sent backwards, being able to change the output to decide what input is required.

In fact, all of the ideas presented in this chapter can be prescribed to software with a little thought. For example, pruning cells so to only display cells of interest can be directly mapped to program slicing in software. Being able to visualise only parts of a system as required, thereby reducing the cognitive load required to decipher it.

Perhaps, though, the major advantage of this work would be that of a virtual reality program maintainer/debugger that presented the maintainer with a powerful selection of features to accomplish the task. The visualisation would already be closer to the maintainers mental model of the program, included with this the data manipulation and code generation abilities, the maintainer could realistically modify programs without touching a single key.

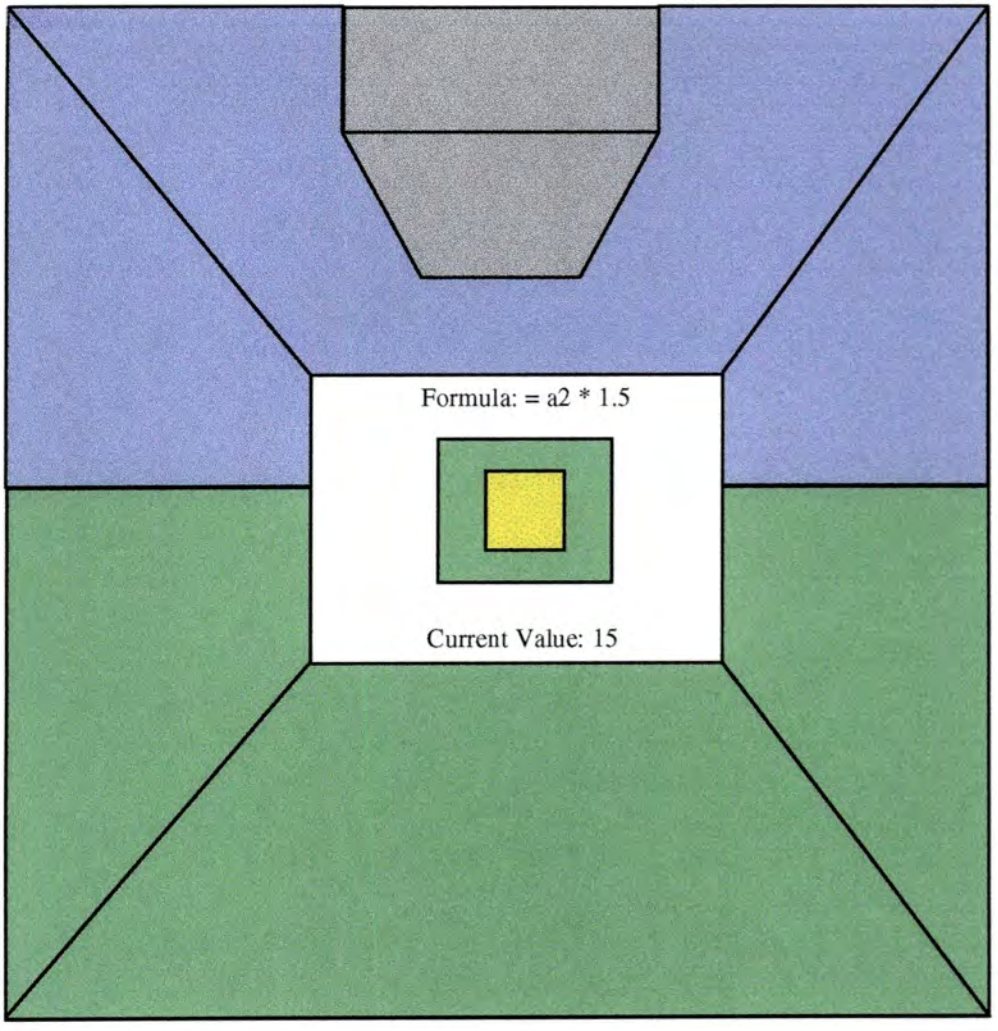


Figure 4.4a View from inside a cell. Whilst inside a cell the viewer has the option to make the walls of the cell transparent in order to view the rest of the spreadsheet from the cell's point of view. Other information can be examined from within the cell, like the formula, the current value of that formula and the number of other cells that depend on this cell or the number of cells which this cell is dependent on.

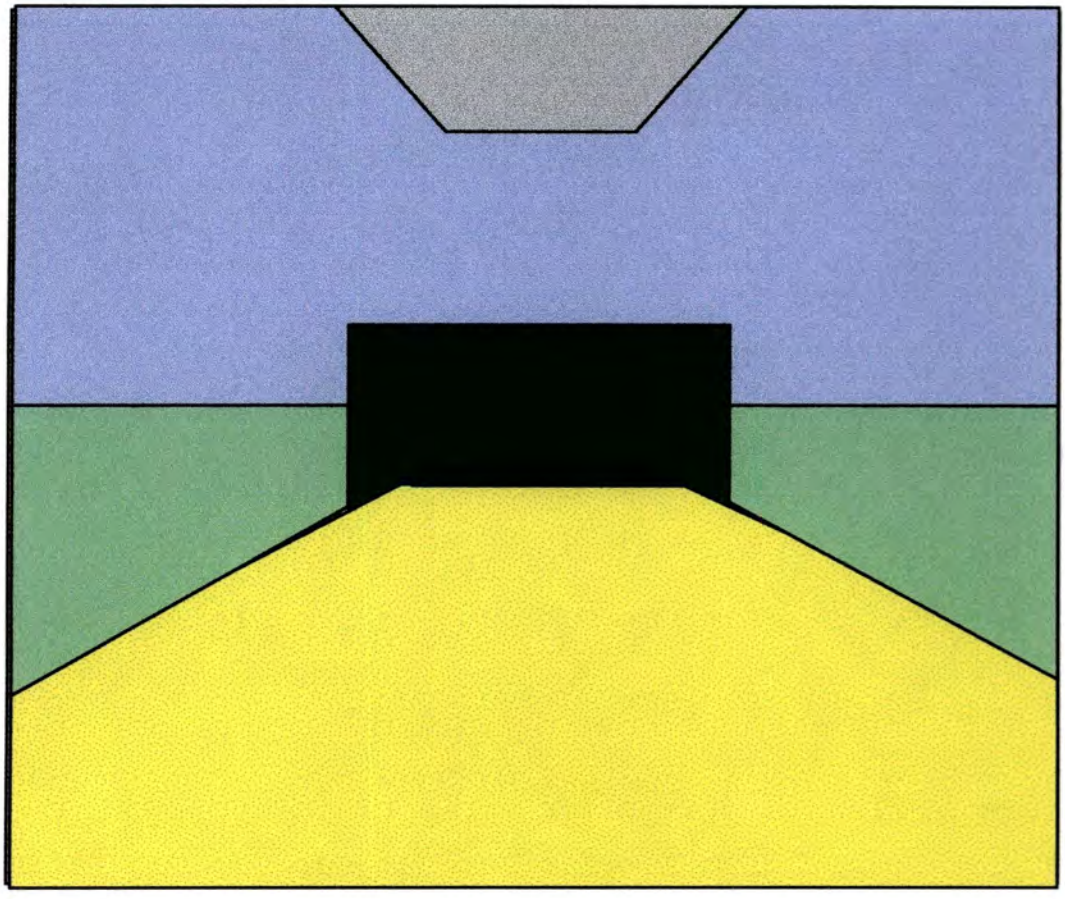


Figure 4.4b Travelling from one cell to the next along a link, as though the viewer was an item of data.

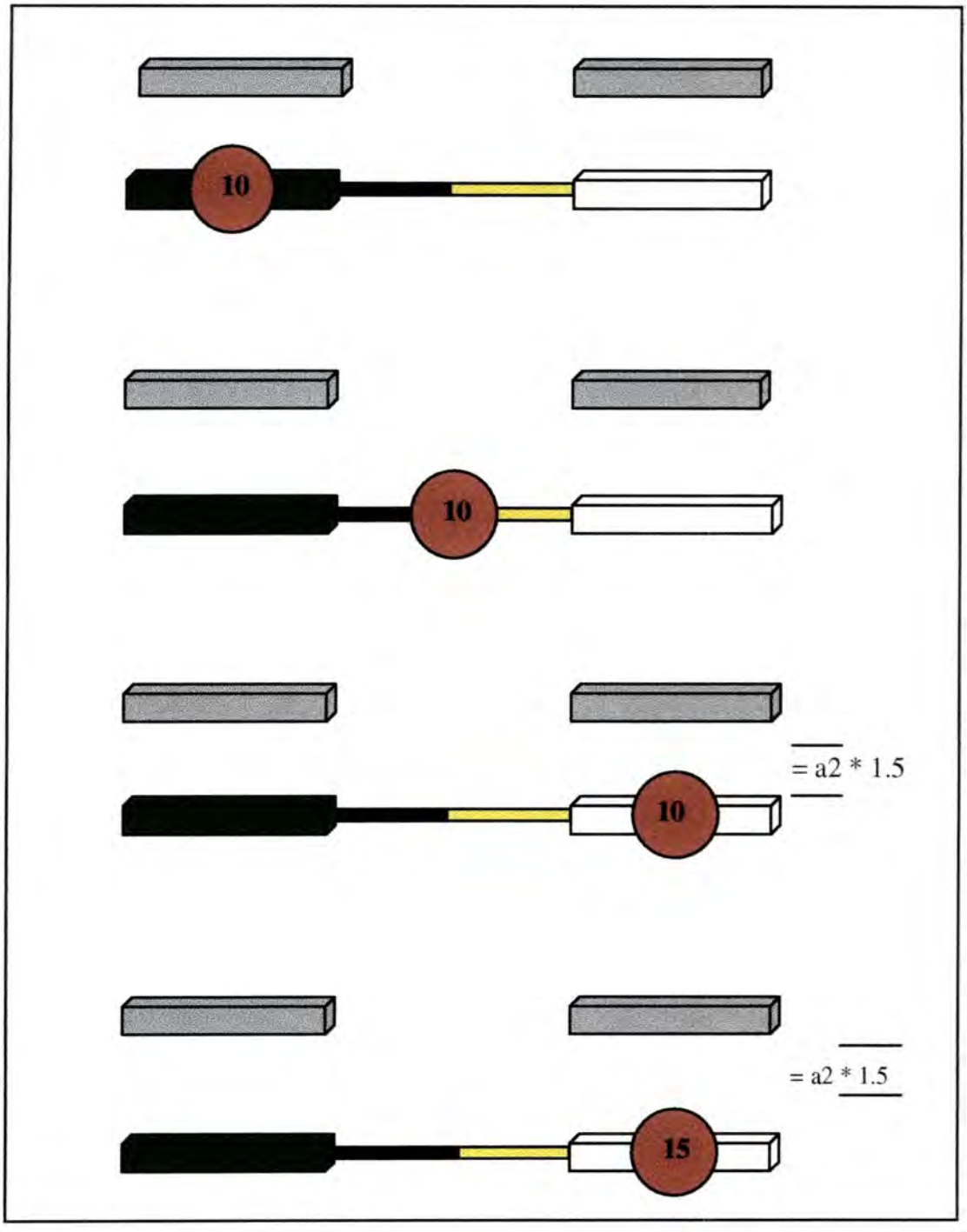


Figure 4.5 Data animation. This feature of the visualisation allows the viewer to see the spreadsheet calculations at work. The number 'ten' starts out in the input box (green) and traverses to the output box (white) where the formula is highlighted and the data is changed accordingly. The option would also exist for the viewer to play the animation backwards, forwards and in single steps.

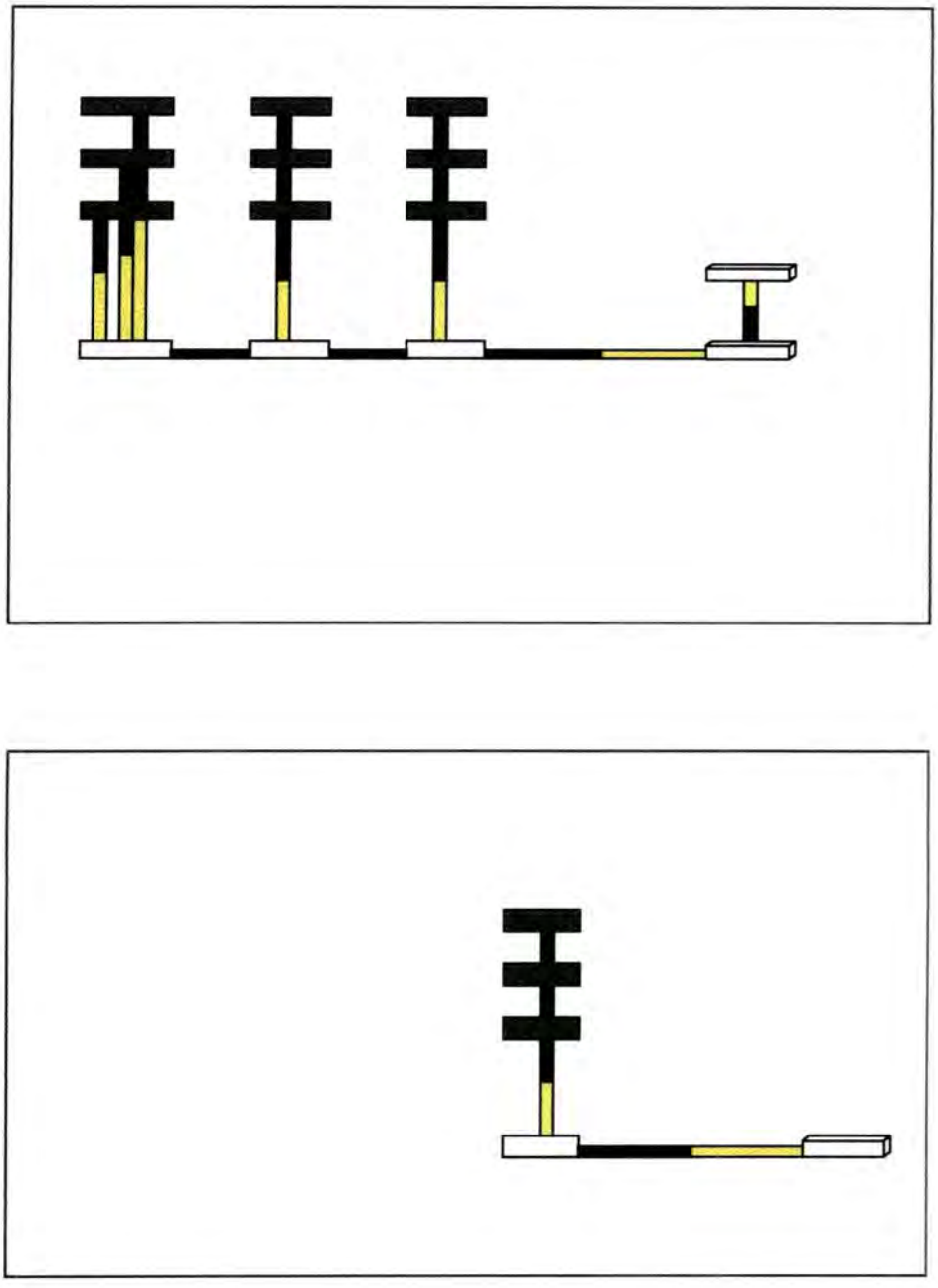


Figure 4.6 Spreadsheet slicing. By selecting a particular cell (in this case the output box at the bottom of the right-most row of input boxes) and a certain number of links (in this case one), the spreadsheet can be sliced to show it with less information for the viewer to comprehend. The top-most diagram shows the spreadsheet before the slice and the bottom-most diagram depicts the spreadsheet after the slice using the criteria mentioned above. That is, all cells that are directly linked to the chosen cell by no more than one link, are removed temporarily from the display.

5 Implementation

The aim of this section is to describe how the ideas in chapter four were turned into reality with the use of current technology and applications. Incorporated into this is a description of 'trans' (a C language [75] program) and SDL (Spreadsheet Description Language) both constructed specially for this research. Details of REND386, VR386, Minicalc and Microsoft Excel are also included as the 'already available' applications used to facilitate the research.

5.1 Introduction and model

Given the ideas in the previous chapter, some way had to be found to map this to current software and technology. As explained in chapter three, state-of-the-art machinery included Silicon Graphics Workstations and so on, but the budget of this research simply did not allow for this. In fact, armed only with a PC 486 50Mhz, the object was to locate some software that could implement a virtual reality visualisation.

In order to visualise a spreadsheet, a conceptual model was constructed to highlight the processes at each stage. The identifiable objects of the model are: the spreadsheet, a spreadsheet description language (SDL) and a virtual reality visualisation system. Interconnecting these would then be some form of generation which would create SDL from a spreadsheet, and then translate SDL to the virtual reality visualisation. The model can be viewed in figure 5.1 below.

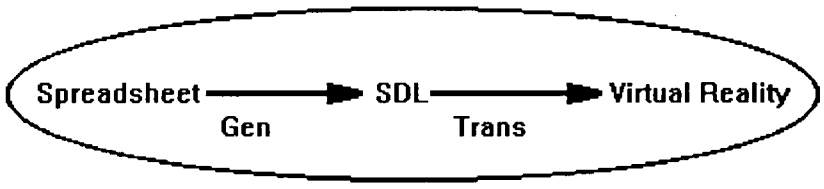


Figure 5.1 From spreadsheet to virtual reality, a model indicating the stages and objects required facilitate such a concept.

5.2 The System

The model depicted above indicates that there are three objects and two processes to be examined. The aim of this section is to detail each of the actual software packages used to facilitate the needs of each stage.

5.2.1 REND386

The book *Virtual Reality Creations* [(Stampe *et al*) 65] actually came with a virtual reality program, called REND386. Until recently REND386 was the only public domain software that is considered a virtual reality construction kit. Other products do exist, such as Superscape LTD's *Superscape* [65], however these program cost in order of thousands of pounds and was beyond the economic reach of this research. Obviously with no other choice, REND386 was used as a 'front-end' with which to view the spreadsheet visualisations.

The aim of this section is to give a brief overview of REND386 and to discuss how it can be used to create spreadsheet visualisations, and what limitations REND386 imposes on them.

Virtual Reality Creations describes REND386 as *"...the first software package for the PC designed specifically for hobbyist VR* [virtual reality]. *It needs no special hardware to create graphics at speeds comparable to commercial VR systems - only a 386 or 486 PC and a VGA card".* As such it was seen as ideal for the research, the results of the work could be run on virtually any desktop computer, meaning that spreadsheet visualisations could be viewed on the same platform that created the spreadsheet in the first place.

However, REND386 is, essentially, just a viewing package. Although there is the possibility to have interaction to some extent, by way of selecting objects and moving them to a new position, for the purposes of this work it was not sophisticated enough to cope with altering the layout of the spreadsheet or allowing any of the interaction options, such as watching data move around the spreadsheet. Further to this, REND386 is incapable of displaying text, thereby disallowing the ability to store information about a cell in the visualisation (such as its data content, it if were an input cell for example).

Due to the architecture of a PC computer, REND386 only had a limited amount of memory available to store internal data structures, and as such was only able to create just over 100 objects within each visualisation. Obviously this had some impact on the visualisations that could be produced, that is, only relatively small spreadsheets could be visualised. However, REND386, did

offer the opportunity to navigate and view the visualisation in 256 colours and three dimensions, easily allowing for the use of the colour schema to be used.

REND386 draws its visualisations by the use of scripts. The script, in ASCII format, details the objects to be placed in the visualisation and the general aspects by the use of two different types of files.

- **The SHAPE file.** Shape files must end with *.PLG* (short for polygon) using the MS-DOS file-naming technique. These files contain instructions telling REND386 how to draw particular shapes. Apart from this, no other information is kept here.

- **The WORLD file.** World files contain most of the information for REND386 to act upon. All the information to do with an object's attributes are stored here, such as colour, placement, orientation, whether it is fixed or allowed to move and so on. Further general information is also stored in this file, including current lighting levels, where the user is placed in the world to start with, colour-map definitions, where *shape* files can be located on disk, for example.

Thus after investigation the features of REND386 could be used to create static visualisations of spreadsheets that the user could freely navigate around by the use of its own programming language 'REND386 Control Language' (RCL) that allows shapes and worlds to be described through the use of ASCII text script files. An example of an RCL file can be found in figure 5.4.

5.2.2 SDL: A Spreadsheet Description Language

In order to analyse spreadsheets a custom built spreadsheet description language was developed. This would have the advantage of being a small language, i.e. a strong connection with a small programming language as well as forming a universal base that could describe all spreadsheets, meaning that the results of this work would not be dependent on any particular spreadsheet. The inclusion into the spreadsheet of a simple output generator is all that would be required for an analysis to take place, or alternatively an external program could also be employed.

To visualise a spreadsheet, certain information must be extracted and detailed in the spreadsheet description language. To keep it as simple as possible, the requirements were as follows:

- only include those cells that are actually in use
- cell labels
- cell formulas

- cell data
- which cells referenced other cells
- which cells were referenced by other cells (optional: for cross referencing purposes only)
- name of the spreadsheet
- orientation of the spreadsheet (i.e. how are the cell references defined)

Rules for the Spreadsheet Description Language (SDL)

To satisfy these requirements, the following language was devised with the aim of being as simple as possible in order that it may be used to describe all manner of spreadsheets, no matter how complicated their workings may be.

<file> : <heading> <cell statements> <end>
<heading> : **spreadsheet** <name> { <orientation> [<checking>] }
<orientation> : **letters across the top | numbers across the top**
<checking> : **used by is disabled | used by not disabled**
<end> : **end spreadsheet**

<cell statements>: <cell statement> | <cell statement> <cell statements>
<cell statement> : <cell heading> { <cell body> }
<cell heading> : **cell** <cell name>
<cell body> : <text line> | <text line> <descriptions>
<text line> : <text type> = "<string>"
<text type> : **text | formula | data**
<descriptions> : <description> | <description> <descriptions>
<description> : <link type> <cell name>
<link type> : **uses | used by**

<cell name> : <word> <numbers>
<name> : <word>

<word> : <letter> | <letter> <word>
<string> : <character> | <character> <string>

<character> : '**space**'..'~'
<letter> : **a..z, A..Z**
<numbers> : <number> | <number> <numbers>
<number> : **0, 1, 2, 3, 4, 5, 6, 7, 8, 9**

A guide to the notation used

In the spreadsheet description language above, a B.N.F. style notation is used to describe it. The following diagram gives an example of a typical SDL file layout:

```
spreadsheet <name>
{
        [letters / numbers] across the top
        used by [is / not] disabled
}

        # The hash symbol is used for comments.  A comment begins at the hash symbol and lasts
        # until the end of the line.

        # Where an option is indicated by surrounding [ ]'s, this means that only one of the choices
        # should be selected.

        cell <cell>
        {
                [ text / formula / data ] = "<string>"      # 1 or more of this line must be present
                [ uses / used by ] <cell>                   # 0,1 or more of this line allowed
        }

        # plus 0,1 or more uses of the above cell declaration

end spreadsheet
```

As an example to illustrate how this actually describes a spreadsheet, imagine a small spreadsheet that converts Pounds Sterling into American Dollars, with an arbitrary conversion rate of £1 is equivalent to $1.5436. For the sake of argument, entering £10 into the spreadsheet the appearance of the spreadsheet would be as follows:

	A	B
1	Pounds	Dollars
2	10	15.436

The inner workings of the spreadsheet would be the same except for cell B1 which would contain the formula *B1 = A2 * 1.5436*. The SDL representation of this would appear as:

```
spreadsheet poundstodollars
{
      letters across the top
      used by not disabled
}

      cell a1
      {
            text = "Pounds"
      }

      cell b1
      {
            text = "Dollars"
      }

      cell a2
      {
            data = "10"
            used by b2
      }

      cell b2
      {
            formula = "a2 * 1.5436"
            uses a2
      }

end spreadsheet
```

As the above language is simple, i.e. just using simple referencing techniques, it can represent any features that spreadsheet vendors may care to include in their packages. The language could be likened to that of a dependency graph for programming languages. Although, there is the current limit that it will not cross reference to external sources (such as another spreadsheet).

SDL is, in essence, describing the underlying formula network of the spreadsheet and, as such, could be viewed as a description language for dependency graphs. From this stance it is possible to see how this language can be applied to programs in that software analysis requires the use of dependency graphs to trace the likes of setting and using variables, or what procedures are called by others.

5.2.3 TRANS

Thus far, the work entailed a spreadsheet description language and the script language used by REND386 (RCL). RCL afforded the ability to be able to navigate and view visualisations, and SDL described spreadsheets. In order, then, to visualise spreadsheets the gap between the two languages had to be bridged. Therefore, a translator was required that would use SDL as its input and create RCL as its output.

Trans was written to meet the requirements mentioned above. In summary these requirements, detailing what *trans* must do, were:

- translate SDL to RCL
- the resulting RCL output should conform to ideas presented in chapter four as far as REND386 would allow

5.1.3.1 An overview of how *trans* functioned

Trans was constructed using a simple Object-Oriented Design approach. It was identified as a three stage process, namely a parser to read the input file, an internal representation to hold the information gained from the parser and an output generator that created RCL scripts from the internal representation.

The modular style design allowed for the internal representation to be viewed as the 'backbone' of the program that needed to be interfaced to, thus any further file formats that needed to be either parsed or generated could simply be 'slotted' into the system with a minimum of effort. Thus should either of the virtual reality visualiser or the SDL change, only the relevant section of *trans* would need to be modified to cope with the change in environment.

In order to code the program, the language C [75] was used, employing the services of the TopSpeed compiler and environment. C offered the ability of migration between platforms, in fact the resulting software was compiled and executed on an Amiga A500 computer. Another advantage included the strong support for structures which reduced the effort needed to code the separate objects identified to construct the software.

The parser

The parser's function was to analyse the SDL file, to make sure that it adhered to the language, and then pass the information onto the internal representation. The parser was coded such that the lexical analyser and the file operations did not need to know anything about the language being parsed or the structure of the internal representation. A single separate module was the only module that 'knew' the rules of the language and had access to the routines to pass information onto the internal representation. Due to this, any new languages that needed to be parsed only required the coding of this one module, as it could access the routines for file access, lexical analysis (such as 'get the next token from the file') and internal representation access, which would already be present.

The internal representation

Essentially, the internal representation had to be able to store information about the underlying formula network of the spreadsheet, that is, what cells referenced other cells - a

dependency graph. Many structures could have been constructed to hold the internal representation, but a balanced binary tree was selected because, conceivably with large spreadsheets, the information required could be enormous and a binary tree has fast insert, search and delete properties.

Two binary trees were required, one to hold information about cells that *used* other cells, and one to hold information about which cells were *used by* other cells. However, this was for data consistency checking and to speed up the information access time. SDL allows for the use of *used by* commands to be turned off. This was allowed to simplify the programmers task when creating an SDL generator for a spreadsheet, as *used by* commands would not have to be generated. *Trans* coped with this by, in essence, copying the information from the *used* binary tree to the *used by* binary tree, thereby fooling the semantic checker into thinking that the parser had encountered the **used by** commands.

Once both binary trees had been constructed, a semantic checker was used to make sure that the cell references were consistent so that, for example, cell 'a' was not referenced by cell 'b' whilst cell 'b' did have a reference to cell 'a'. Thus, once the semantic checker is satisfied, the output generator could use the information to create the RCL script.

The output generator

This can be viewed as a code generator, as the function of the output generator is to extract the information from the binary trees and generate an RCL output file that is acceptable to REND386, thus creating a graphical representation, or visualisation of the spreadsheet.

In order to preserve easy modifications (to the *trans* software), the object attributes of cells and links (i.e. how they are drawn, how they are placed and what colour they are to be) were all defined in a separate module. From these two resources of information the output generator's objective was to create visualisation scripts that conformed, as far as REND386 would allow, to the concepts presented in chapter four. Hence, placing the cells was not a problem as the cell names could be used as a guide (i.e. cell 'a2' would always appear either one below or one to the right of cell 'a1', depending on the orientation of the spreadsheet). However, when it came to placing links between the cells, this proved to be a substantial problem especially because it was a requirement that none of the links were to be allowed to cross. Sedgewick describes an algorithm in his book *Algorithms* [72], from which a solution to the problem could be implemented and included in the *trans* software. The solution included the use of stalks to connect a link that was at a different depth to the cell. Stalks appear as a grey/blue colour in the visualisation.

5.2.4 The spreadsheet

The spreadsheet object was approached from two different angles. Firstly, there would exist spreadsheets for which an SDL generator could exist within the spreadsheet itself. Secondly, there would exist those packages that would be in binary format only and would have no facility for producing an SDL file.

To cope with these two approaches, two different spreadsheet packages were selected to reflect the needs of each.

5.2.4.1 Minicalc

Minicalc offered the ability to modify its inner workings as the source code was available in the book *Advanced Programming: Design and Structure Using Pascal* [(Miller) 73]. Thus the SDL generation could be included within the spreadsheet itself, allowing the generation section of the model to become one with the spreadsheet object.

Fortunately, *Minicalc* had also been programmed in a very modular way allowing the modifications to be made with relative ease. In essence all that needed to be done was to traverse the spreadsheet's dependency graph and write out any cell references indicated by the formulas.

5.2.4.2 Microsoft Excel (V4)

Excel, for obvious reasons, was only available in binary format (i.e. no source code) and thus there was no ability to include an SDL generator within the spreadsheet. This problem could be solved in one of two ways. Firstly by adhering strictly to the model and writing a program that would parse an Excel file and create the SDL output. However, secondly, there existed a more efficient way of achieving this. *Trans* already has the ability to parse and generate files from one form to another. Therefore, why not include into *trans* the ability to decode an Excel file and place that information straight in to the internal representation and output RCL from there.

Hence, using the information and software provided with the book *Microsoft Excel Software Development Kit (Version 4)* [74] it was possible to update *trans* to be able to parse and decipher Microsoft Excel Version 4 spreadsheet files and thus create REND386 script files for visualisation. *Trans* was programmed to automatically recognise which file format it was dealing with (SDL or Excel 'BIFF' format), hence whichever system was used to create and describe a spreadsheet did not alter the way in which the model functioned.

Figure 5.6 depicts how the spreadsheet demonstrated in figure 5.2 is represented in Microsoft's Excel. Once saved to disk, the resulting binary file needed to be converted to a readable

BIFF file by the use of *dumpbiff* provided with the software development kit. Then is was possible to use trans as before, i.e. by typing 'trans *input_filename*' and to follow the original model from there onwards.

5.3 The working model

This section is intended as an guide to the overall working model. To do this, an example spreadsheet visualisation from spreadsheet construction in *Minicalc* through to virtual reality visualisation in REND386 is presented. For comparison, figure 5.6 contains a screen shot of what the same spreadsheet would look like in Excel.

The spreadsheet to be visualised is a simple one that collects (fictitious) data about rainfall at certain times of the day and on certain days of the week. The data is then summarised as daily totals, a week total and a cell displaying the average amount of rainfall per day. The *Minicalc* spreadsheet can be seen in figure 5.2.

The next step is to obtain the SDL representation of the spreadsheet. The resulting SDL output can be viewed in figure 5.3. Once the SDL file had been created, it was the turn of *trans* to convert this to a REND386 script, which is achieved by typing 'trans *input_filename*'. The shape files that accompany the world file are also produced at the same time. The REND386 script file can be examined in figure 5.4. Finally, REND386 can be executed with the resulting RCL script as a parameter and the maintainer is the free to navigate the visualisation. Some example visualisations can be seen in figure 5.5.

Hence, visualising a spreadsheet using this model is a simple task of using three programs to create, translate and view the spreadsheet.

```
Change to Cell : a1

Current cell is : a1
          1          2          3          4          5          6
    +----------+----------+----------+----------+----------+----------+
a   |          |Monday    |Wednesday |Friday    |          |          |
    +----------+----------+----------+----------+----------+----------+
b   |9.00am    |      0.00|      1.00|      3.00|          |          |
    +----------+----------+----------+----------+----------+----------+
c   |3.00pm    |      0.00|      2.00|      0.00|          |          |
    +----------+----------+----------+----------+----------+----------+
d   |9.00pm    |      1.00|      5.00|      0.00|          |          |
    +----------+----------+----------+----------+----------+----------+
e   |          |          |          |          |          |          |
    +----------+----------+----------+----------+----------+----------+
f   |Daily TOT |      1.00|      8.00|      3.00|Week TOTAL|     12.00|
    +----------+----------+----------+----------+----------+----------+
g   |          |          |          |          |          |          |
    +----------+----------+----------+----------+----------+----------+
h   |          |          |          |          |Avg/Day   |      4.00|
    +==========+==========+==========+==========+==========+==========+
C(ell), H(elp), E(xpr.), L(abel), (loa)D, S(ave), X(SDL Save), Q(uit)
```

Figure 5.2 Screenshot of Minicalc. This is what the user sees when viewing a spreadsheet in *Minicalc*. The spreadsheet collects data about rainfall during a five day period, and summaries the data by the use of totals and an average.

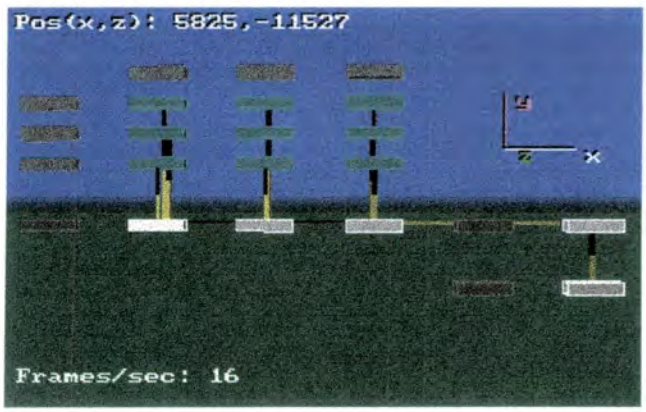
```
spreadsheet minicalcoutput
{
 numbers across the top
 used by is disabled
}

 cell a2
 {
  text = "Monday"
 }

 cell a3
 {
  text = "Wednesday"
 }

 cell a4
 {
  text = "Friday"
 }

 cell b1
 {
  text = "9.00am"
 }

 cell b2
 {
  # contains the value : 0.00
 }

 cell b3
 {
  # contains the value : 1.00
 }

 cell b4
 {
  # contains the value : 3.00
 }

 cell c1
 {
  text = "3.00pm"
 }

 cell c2
 {
  # contains the value : 0.00
 }

 cell c3
 {
  # contains the value : 2.00
 }

 cell c4
 {
  # contains the value : 0.00
 }

 cell d1
 {
  text = "9.00pm"
 }

 cell d2
 {
  # contains the value : 1.00
 }

 cell d3
 {
  # contains the value : 5.00
 }

 cell d4
 {
  # contains the value : 0.00
 }

 cell f1
 {
  text = "Daily TOT"
 }

 cell f2
 {
  formula = "(b2+c2)+d2"
  uses b2
  uses c2
  uses d2
 }

 cell f3
 {
  formula = "(b3+c3)+d3"
  uses b3
  uses c3
  uses d3
 }

 cell f4
 {
  formula = "(b4+c4)+d4"
  uses b4
  uses c4
  uses d4
 }

 cell f5
 {
  text = "Week TOTAL"
 }

 cell f6
 {
  formula = "(f2+f3)+f4"
  uses f2
  uses f3
  uses f4
 }

 cell h5
 {
  text = "Avg/Day"
 }

 cell h6
 {
  formula = "f6/3.00"
  uses f6
 }


end spreadsheet
```

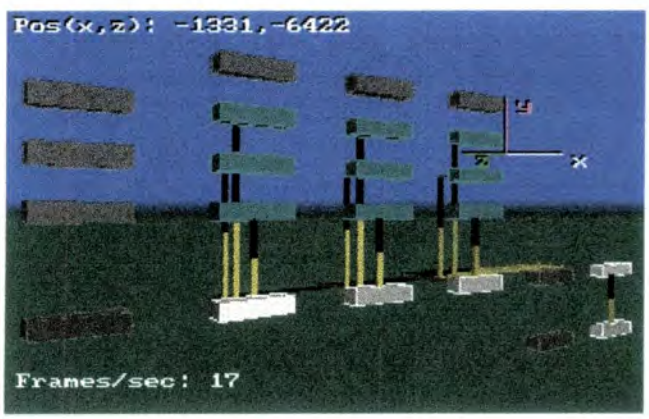
Figure 5.3 SDL representation of the spreadsheet presented in figure 5.2

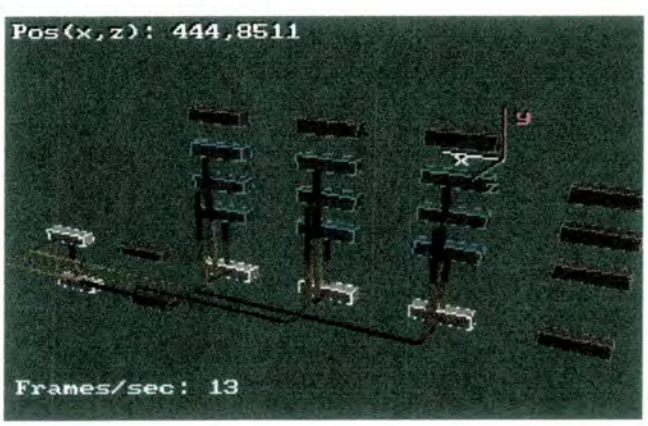
```
# The following file contains the Rend386 Control Language statements
# to produce a 3D representation of a spreadsheet.

# *** THIS FILE HAS BEEN COMPUTER GENERATED ***


worldscale 1.0  # 1mm per unit

# clipping distances

hither  10
yon 200000

# camera 1 <x,y,z posn>    <x,y,z looking>    <zoom>
  camera 1  0,0,-5000       0,5,0              2
  camera 2  0,0,5000        0,175,0            2
  camera 3  4000,0,-5000    0,0,0              2
  camera 4  4000,0,5000     0,175,0            2
  camera 5  2000,5000,-5000 30,0,0             2
  camera 6  2000,5000,5000 30,180,0            2

# if you wish to add a loadpath, then do it here :
# loadpath <drive>:[<directory><directory>...]
# for example:  c:rend386myspread

stepsize 1
anglestep 5

light 2000,10000,-10000
ambient 76

# surface definitions

surfacedef gry   0x1eff
surfacedef rd    0x11ff
surfacedef blu   0x1bff
surfacedef grn   0x19ff
surfacedef yllw  0x16ff
surfacedef blck  0x1000
surfacedef whte  0x1fff

surfacemap grey
surface 1 gry
surface 2 gry
surface 3 gry
surface 4 gry
surface 5 gry
surface 6 gry
surface 7 gry
surface 8 gry
surface 9 gry
surface 10 gry

surfacemap white
surface 1 whte
surface 2 whte
surface 3 whte
surface 4 whte
surface 5 whte
surface 6 whte

surfacemap green
surface 1 grn
surface 2 grn
surface 3 grn
surface 4 grn
surface 5 grn
surface 6 grn

surfacemap left_yellow
surface 1 yllw
surface 2 blck
surface 3 yllw
surface 4 yllw
surface 5 yllw
surface 6 yllw
surface 7 blck
surface 8 blck
surface 9 blck
surface 10 blck
```

```
surfacemap right_yellow
surface 1 blck
surface 2 yllw
surface 3 blck
surface 4 blck
surface 5 blck
surface 6 blck
surface 7 yllw
surface 8 yllw
surface 9 yllw
surface 10 yllw

# what follows is the customised bit for each spreadsheet

object a2=cell 1,1,1 0,0,0 2000,5200,0 256 grey fixed
object a3=cell 1,1,1 0,0,0 4000,5200,0 256 grey fixed
object a4=cell 1,1,1 0,0,0 6000,5200,0 256 grey fixed
object b1=cell 1,1,1 0,0,0 0,4500,0 256 grey fixed
object b2=cell 1,1,1 0,0,0 2000,4500,0 256 green fixed
object b3=cell 1,1,1 0,0,0 4000,4500,0 256 green fixed
object b4=cell 1,1,1 0,0,0 6000,4500,0 256 green fixed
object c1=cell 1,1,1 0,0,0 0,3800,0 256 grey fixed
object c2=cell 1,1,1 0,0,0 2000,3800,0 256 green fixed
object c3=cell 1,1,1 0,0,0 4000,3800,0 256 green fixed
object c4=cell 1,1,1 0,0,0 6000,3800,0 256 green fixed
object d1=cell 1,1,1 0,0,0 0,3100,0 256 grey fixed
object d2=cell 1,1,1 0,0,0 2000,3100,0 256 green fixed
object d3=cell 1,1,1 0,0,0 4000,3100,0 256 green fixed
object d4=cell 1,1,1 0,0,0 6000,3100,0 256 green fixed
object f1=cell 1,1,1 0,0,0 0,1700,0 256 grey fixed
object f2=cell 1,1,1 0,0,0 2000,1700,0 256 white fixed
object f3=cell 1,1,1 0,0,0 4000,1700,0 256 white fixed
object f4=cell 1,1,1 0,0,0 6000,1700,0 256 white fixed
object f5=cell 1,1,1 0,0,0 8000,1700,0 256 grey fixed
object f6=cell 1,1,1 0,0,0 10000,1700,0 256 white fixed
object h5=cell 1,1,1 0,0,0 8000,300,0 256 grey fixed
object h6=cell 1,1,1 0,0,0 10000,300,0 256 white fixed
object ld2f2=link 1150,50,50 0,0,270 2500,3100,125 256 right_yellow fixed
object ld3f3=link 1150,50,50 0,0,270 4500,3100,125 256 right_yellow fixed
object ld4f4=link 1150,50,50 0,0,270 6500,3100,125 256 right_yellow fixed
object lf6h6=link 1150,50,50 0,0,270 10500,1700,125 256 right_yellow fixed
object lf2b2=link 2800,50,50 0,0,90 2500,1825,550 256 left_yellow fixed
object lslf2b2=link 300,50,50 0,270,0 2500,1825,250 256 grey fixed
object rslf2b2=link 300,50,50 0,270,0 2500,4625,250 256 grey fixed
object lf2c2=link 2100,50,50 0,0,90 2500,1825,850 256 left_yellow fixed
object lslf2c2=link 600,50,50 0,270,0 2500,1825,250 256 grey fixed
object rslf2c2=link 600,50,50 0,270,0 2500,3925,250 256 grey fixed
object lf3b3=link 2800,50,50 0,0,90 4500,1825,550 256 left_yellow fixed
object lslf3b3=link 300,50,50 0,270,0 4500,1825,250 256 grey fixed
object rslf3b3=link 300,50,50 0,270,0 4500,4625,250 256 grey fixed
object lf3c3=link 2100,50,50 0,0,90 4500,1825,850 256 left_yellow fixed
object lslf3c3=link 600,50,50 0,270,0 4500,1825,250 256 grey fixed
object rslf3c3=link 600,50,50 0,270,0 4500,3925,250 256 grey fixed
object lf4b4=link 2800,50,50 0,0,90 6500,1825,550 256 left_yellow fixed
object lslf4b4=link 300,50,50 0,270,0 6500,1825,250 256 grey fixed
object rslf4b4=link 300,50,50 0,270,0 6500,4625,250 256 grey fixed
object lf4c4=link 2100,50,50 0,0,90 6500,1825,850 256 left_yellow fixed
object lslf4c4=link 600,50,50 0,270,0 6500,1825,250 256 grey fixed
object rslf4c4=link 600,50,50 0,270,0 6500,3925,250 256 grey fixed
object lf6f2=link 8000,50,50 0,0,180 10500,1825,1150 256 left_yellow fixed
object lslf6f2=link 900,50,50 0,270,0 10500,1825,250 256 grey fixed
object rslf6f2=link 900,50,50 0,270,0 2500,1825,250 256 grey fixed
object lf6f3=link 6000,50,50 0,0,180 10500,1825,1450 256 left_yellow fixed
object lslf6f3=link 1200,50,50 0,270,0 10500,1825,250 256 grey fixed
object rslf6f3=link 1200,50,50 0,270,0 4500,1825,250 256 grey fixed
object lf6f4=link 4000,50,50 0,0,180 10500,1825,1750 256 left_yellow fixed
object lslf6f4=link 1500,50,50 0,270,0 10500,1825,250 256 grey fixed
object rslf6f4=link 1500,50,50 0,270,0 6500,1825,250 256 grey fixed
```

Figure 5.4 REND386 script language created by *trans* to describe the visualisation presentation of the spreadsheet displayed in figure 5.2







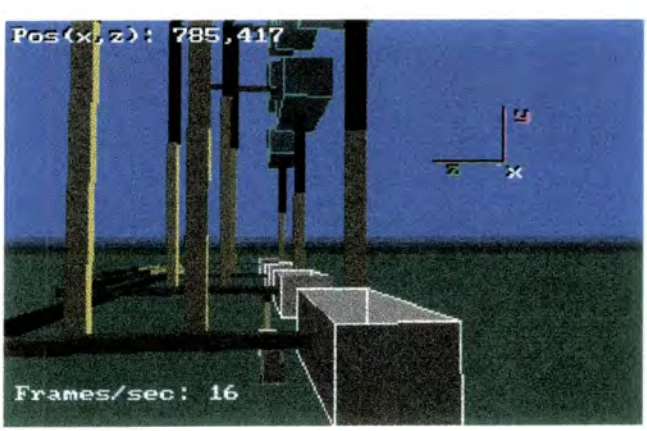


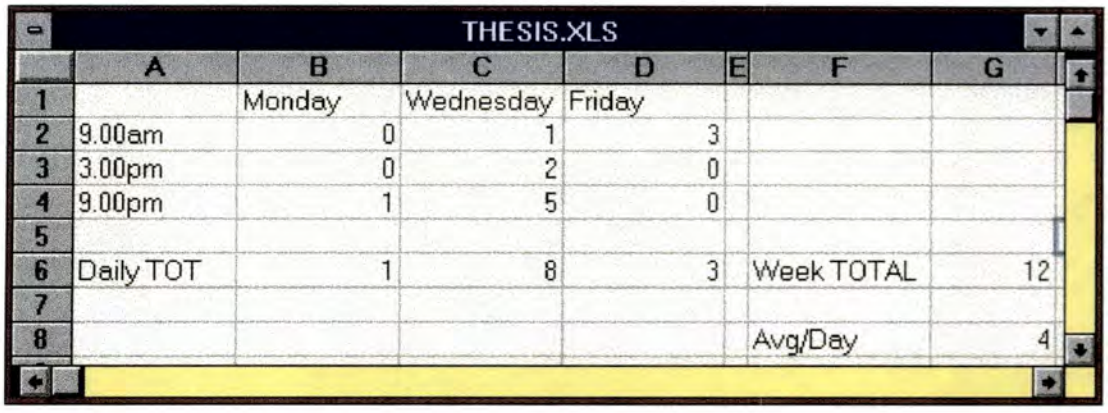
Figure 5.5 Screenshots taken from REND386 depicting how a maintainer would see the visualisation of the spreadsheet in figure 5.2.

This first picture shows how the maintainer is able to see the spreadsheet in a form close to that of the original spreadsheet package display. This enables the maintainer to quickly become familiar with his/her surroundings.

Here is shown how the maintainer is able to change their point of view to get a better look at the underlying network of formulas.

This is a view from the back of the spreadsheet, as though the maintainer were the spreadsheet itself, looking out towards the user (using the spreadsheet package).

This final view represents how the maintainer is able to get inside the spreadsheet and view certain aspects of it. Here the view is from cell F2 (the cell nearest the view) looking along spreadsheet in an easterly direction (assuming up is north).



	A	B	C	D	E	F	G
1		Monday	Wednesday	Friday			
2	9.00am	0	1	3			
3	3.00pm	0	2	0			
4	9.00pm	1	5	0			
5							
6	Daily TOT	1	8	3		Week TOTAL	12
7							
8						Avg/Day	4

Figure 5.6 Screenshot from Microsoft's Excel. This is one way in which the spreadsheet depicted in figure 5.2 could be seen by the user in Microsoft's Excel.

6 Evaluation

The aim of this chapter is to evaluate the work presented so far by exploring examples of how common errors in spreadsheets can be recognised and corrected using the model presented in chapter five.

6.1 Evaluating the model

Chapter four saw the introduction of common mistakes made in spreadsheets. Chapter five attempted to present a model that could ease comprehension of spreadsheets. Comprehension is one of the processes that takes place when debugging spreadsheets, as the function of a spreadsheet must be understood before any solutions can be implemented.

The aim of this section is to tie these points together by evaluating whether common mistakes in spreadsheets can be found using the model presented in chapter five, and therefore establish whether a useful system for comprehension has been instituted. In order to achieve this, some of the common mistakes first detailed in chapter four will be recreated and then explored in the virtual reality visualisation to depict how they would be represented using such a visualisation technique.

6.1.1 Visualising the mistakes

In order to visualise the mistakes, the typical approach will be to demonstrate a view of the correct spreadsheet (i.e. the spreadsheet with no errors) and then to display the incorrect version, noting the differences that occur and the way in which errors manifest themselves pictorially.

6.1.1.1 Typing mistakes

The spreadsheet in figure 6.1 adds two numbers together (in cells 'b1' and 'b2') to produce a total displayed in cell 'b3'. Cells 'a1', 'a2' and 'a3' are label cells. Figure 6.1 also shows the visualisation of this spreadsheet with no bugs in it.

	A	B
1	1st Half	21
2	2nd Half	36
3	Total	*'=b1+b2'*

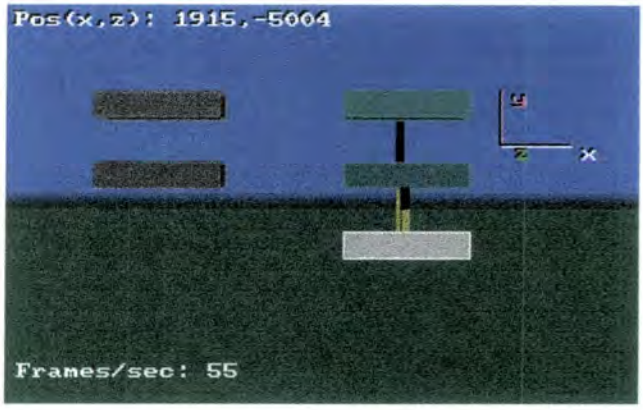


Figure 6.1 A simple spreadsheet that adds two numbers together is shown above, with its corresponding virtual reality visualisation depicted below.

By introducing various typing mistakes into cell 'b3', an evaluation of the usefulness of the visualisation can take place. Figure 6.2 demonstrates how forgetting the equals sign in front of the formula can effect the visualisation. Immediately it is very clear that something is incorrect as all the cells appear as grey in colour and there is no inter-connectivity between the cells (i.e. the absence of links) indicating that all the cells are label cells and that none of them contain any formulas. Assuming that the maintainer has had previous access to the spreadsheet and is aware that the it should sum the contents of two of the cells, the maintainer is able to conjecture that one of the cells should contain a formula, but in reality, does not due to the lack of links in the visualisation. From this evidence the maintainer is then able to return to the spreadsheet package and investigate, to find that cell 'b3' does, in fact, contain a typing mistake.

Figure 6.2 A virtual reality visualisation of the spreadsheet in figure 6.1, but with a typing error in cell 'b3' resulting in that cell being regarded as a label cell and not a formula containing cell.

Further to this, imagine that the spreadsheet user had mis-read the column headings and typed in the formula *=c1+c2* into cell 'b3'. Figure 6.3 depicts how this type of mistake is visualised. Notice that the cells are still not of the correct colours, but the real give away is the fact that the links do not connect to another cell - they are left hanging in space. This is what a reference to an unused cell looks like. The maintainer will be able to deduce that one of two possibilities could occur from this. Firstly, that the formula stored in cell 'b3' is incorrectly referencing the cells it is supposed to be acquiring its information from. Or secondly, that the formula in 'b3' is correct and the spreadsheet developer has forgotten to define cells 'c1' and 'c2'. Either of these cases could be true, but the maintainer is now aware of this and can return to the spreadsheet package to investigate the likely possibilities of which is the correct assumption.

If the 'used by' feature of *trans* is not disabled, this type of error will actually get picked up by the semantic checker in *trans*, disallowing the visualisation to take place until the fault is corrected.

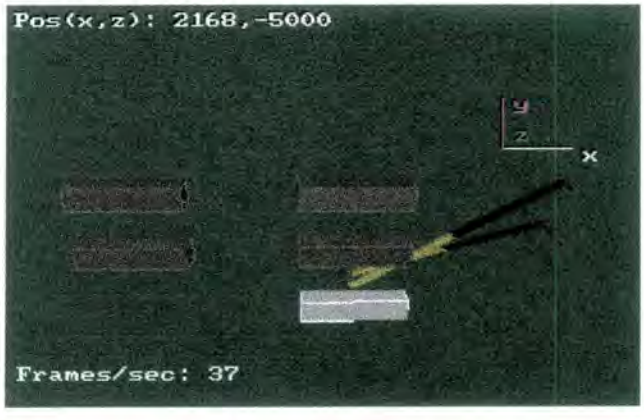


Figure 6.3 A virtual reality visualisation of the spreadsheet in figure 6.1, but with a typing error in cell 'b3' such that it is summing the contents of cells that do not contain anything.

6.1.1.2 Mis-understanding

The most common form of mis-understanding is between absolute and relative cell referencing. This error occurs because similar blocks of formulas are usually required throughout a spreadsheet and the quickest way to do this is to define it once and then copy it to wherever it is required. Thus if the spreadsheet developer is confused between the two types of cell referencing, incorrect cell references can be encountered. To illustrate this example, figure 6.4 depicts a spreadsheet that adds two columns of data together, together with its virtual reality visualisation.

	A	B	C
1	68	16	
2	48	74	
3	45	37	
4	18	53	
5			
6	*=a1+a2+a3+a4*	*=b1+b2+b3+b4*	<< Totals

Figure 6.4 A small spreadsheet that adds two columns of numbers together (above) with virtual reality visualisation (below).

Assume now, that a spreadsheet user has defined the first formula series (cell 'a6') incorrectly by using absolute cell referencing instead of relative cell referencing. Once the formula is copied to cell 'b6', the user believes that the result being displayed would be that of the sum of cells 'b1' to 'b4', but in reality this is not the case. Cell 'b6' will display the addition of cells 'a1' through to 'a4'. However, by inspecting the visualisation, shown in figure 6.5, it can be seen that straight away that the right-most cells are all grey (except for 'b6'), indicating that they are not used in the spreadsheet's calculation. Add to this the fact that the links indicate that cell 'b6' is using the left-most column of cells for its calculation, as is cell 'a6'. This would indicate to the maintainer that it is extremely likely that something is wrong with the spreadsheet. For example, why would a column of labels be required? Is it likely that two output cells would use precisely the same input cells for their calculation? These are the types of questions that a maintainer would ask him/herself and lead to the conclusion that these cells are in need of further investigation, as more than one correct scenario is possible under these circumstances.

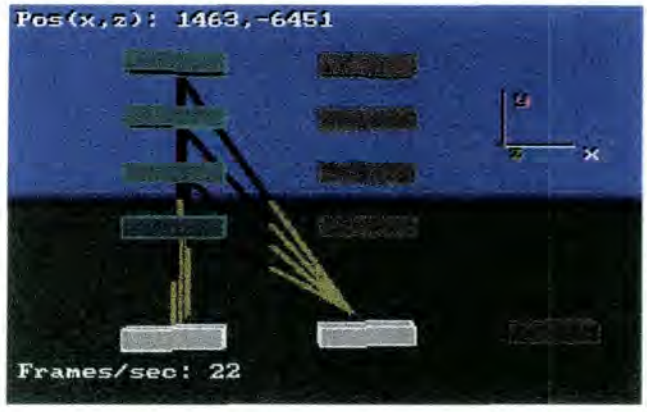


Figure 6.5 A virtual reality visualisation of the spreadsheet presented in figure 6.4, although this time there is an error in the formula of cell 'b6'. 'b6' should reference the cells directly above it, but because absolute referencing has been used instead of relative referencing, when the formula was copied from cell 'a6', the formula remained the same instead of automatically being updated to reflect the modification.

6.1.1.3 Carelessness

An example cited in chapter four, was that of not including all of the range within an addition, resulting in at least two companies losing 'six figure incomes'. The example shown here demonstrates how such a problem could be easily spotted using the virtual reality visualisation. Using the spreadsheet presented in figure 6.4 as the reference model, figure 6.6 depicts how such an error would be visualised. In the left-most column, only cells 'a1' to 'a3' have been included in the calculation - cell 'a4' has been 'accidentally' left out. This can be immediately picked up as the cell is grey, when neighbouring cells suggest that it should be green. Furthermore, there is no link to the greyed out cell. Although not definite proof that something is wrong - it could conceivably mean to be like that - it is a strong clue that something could be wrong. This can be highlighted in the right-most column which has cell 'b2' missing from the formula in cell 'b6'. The maintainer would conjecture that this is an extremely unlikely place for a label cell to be placed, although (again) it may be the case. In either situation, it definitely warrants and investigation.

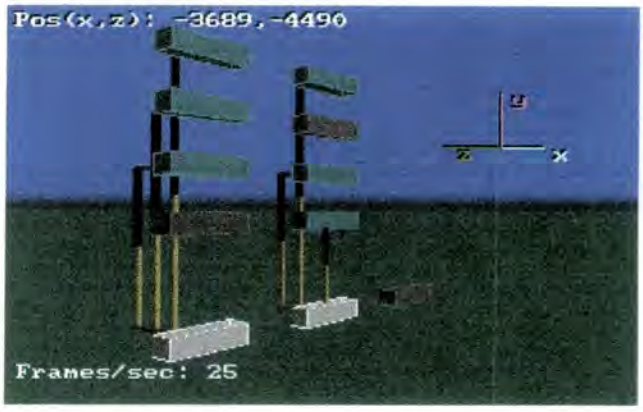


Figure 6.6 A virtual reality visualisation of a bugged version of the spreadsheet presented in figure 6.4.

6.1.2 Assessment of the visualisations

From the evidence presented above it is conjectured that the visualisations are useful because they allow the maintainer to grasp a better appreciation of the inner workings of a spreadsheet, i.e. the computational domain (or 'how'). Although many spreadsheets do allow the user to display the formulas and the surface layer of the spreadsheet at the same time, none of them allow for pictorial representations. If a user needs to ascertain which cells are linked to a particular formula, he/she can look at the formula for the answer, but no physical link is made. That is, the spreadsheet does not draw a visible line between the connecting cells. As demonstrated above, this can be a powerful feature - especially for those formulas that reference empty cells. The problem domain (or 'what') is represented in the visualisations by the physical similarity of the spreadsheet, and the use of coloured graphics to indicate the typical regions on a spreadsheet as most often indicated by users, for example the input region, the output region and so on. Thus if a maintainer believes one section of a spreadsheet to be a table of input values, but unexpectedly finds a label cell in the middle of it, the maintainer could conjecture that there is an error, for example, an addition that does not contain the full range of cells that it is meant to. Although, of course, it may mean to be like that. The visualisation is a tool for pointing the maintainer in the direction of possible errors.

In each of the cases above, the particular error in the spreadsheet could be demonstrated and spotted easily. However, without these visualisations these types of errors would be virtually impossible to detect as Hendry and Green [79] have discovered. Therefore, from this fact alone, the only conclusion that can be reached is that the visualisations can provide appropriate information, in a compact and easy to understand form, for a maintainer to be able to successfully modify an incorrect spreadsheet or enhance the functionality of a correct (or incorrect) one.

Due to the limitations imposed by the software and hardware used, only about 100 objects could be generated in any one visualisation. This meant that large spreadsheets could not be visualised to give a better understanding of how an overload of information could cloud the virtual reality visualisations. Add to this the fact that REND386 was incapable of addressing the features of text presentation, data manipulation and animation of data, the visualisation as an entire system as described in chapter four, had much reduced functionality. Although, the resulting visualisations were of sufficient use to aid with comprehension and, in particular, debugging.

7 Conclusion

The aim of this chapter is to conclude the work undertaken in the previous chapters by briefly detailing and critically appraising it. The criteria for success will be examined and discussed. This will then be followed by a section detailing possible future work in this subject area.

7.1 Synopsis

The purpose of this section is to give a brief synopsis of the work actually accomplished.

7.1.1 Introduction and background work

The first three chapters of this thesis comprise of the background work that was carried out to establish the motives and possible directions of work to visualise software. Chapter one is a springboard from which the concepts of program comprehension are introduced into a wider sense of software engineering. Its main function is to establish the origins of the research and to place the work within the confines of computer science.

Chapters two and three are the result of research into the appropriate subject areas. Formulating the idea that virtual reality could be used to visualise software, chapter two describes in detail the current program comprehension theories, such as programming plans, beacons, program slicing and mental models. All of the theories have a notable margin of overlap, suggesting that they should be used in conjunction with each other as opposed to instead of each other. From the information obtained at this stage it was possible to derive an insight into how the process of program comprehension is undertaken. All of the authors agreed that a mental model is perceived by the maintainer when inspecting a program. It is the strength of this model that in turn reflects directly on the quality of the modification, such that an incorrect mental model would result in a poor modification of the software. Therefore, strategies were required to enhance the construction of the mental model.

Visualisations have been shown to aid in the accomplishment of mental model construction, but often, these visualisations are a passive experience represented in two dimensions. The lack of a third dimension results in conventional visualisations to be overloaded with information, leaving the

maintainer with a more complex task than before, as they then have to understand the visualisation. A natural next step would be to involve the third dimension into the visualisations to increase their flexibility. Hence chapter three explores the realm of virtual reality and how three dimensions could be employed to construct visualisations by examining other work that is currently in development, or in use. Virtual reality can take place on a budget of millions as well as hundreds of pounds meaning that the work carried out here is available to all, not just the financially endowed. Virtual reality also offers the ability to get into the visualisation and interact with the contents, allowing data manipulation and physical restructuring, which in turn results in the ability to play 'what if' situations. Further to this, the virtual reality visualisations can display more information as the maintainer requires it, as opposed to seeing it all at once. This is directly reflected in the findings that maintainers, when initially comprehending programs, are directed by their desire for knowledge about the software. Finally, these types of visualisations are pictorially closer to the mental model possessed by maintainers, this reduces the cognitive load on them and allows the maintainer to get on with the real job at hand: maintenance.

7.1.2 A model for visualisation

Chapter four saw the introduction of spreadsheets to be visualised, as opposed to programming languages for the simple reason that to visualise a complex language (a programming language) would be beyond the time-scale of this work. Spreadsheets are a simpler language that offer many similarities to programming languages and are already partly visual. For example, cell referencing and formulas in spreadsheets are similar to procedure calls, variable referencing and data manipulation in conventional programming languages; basically, a spreadsheet is a dependency graph.

Just as programming languages do, spreadsheets have their own problems associated with them, as investigated by Hendry and Green [9, 79]. The most common form of mistake is an incorrect cell reference in a formula which can be caused in any manner of ways (e.g. typing mistake, mis-understanding of a function or simple carelessness) and can be extremely difficult to spot, even amongst expert spreadsheet developers and users.

In order to visualise a spreadsheet, a spreadsheet description language (SDL) was developed to cope with the many features of spreadsheets. This meant that the language was required to be as simple as possible as all features of a spreadsheet could be described in terms of one cell referencing another. Once an SDL description of a spreadsheet could be generated, the following stage required a translator to interpret SDL and phrase this information into a form acceptable by the virtual reality visualisation system, REND386, which used an ASCII based script language, REND386 Control

Language (RCL), to describe its virtual reality environments. *Trans* (a parser/generator) was developed to cope with such a transformation.

Chapter five documents the model used to visualise spreadsheets and all the stages in between. The resulting work allowed for spreadsheets constructed in *Excel* to be visualised in virtual reality, however, with the limitations of the virtual reality system, some of the functionality of the concepts expressed in chapter four could not be implemented. However, a working visualisation was illustrated in chapter five, demonstrating that the work described is possible to implement.

7.1.3 Evaluation of the model

Chapter six carries on the work by evaluating the model presented in chapter five, by using the model to visualise spreadsheets infected with typical mistakes as described in chapter four. The evaluation found that the virtual reality visualisations were useful in that they can present the maintainer with sufficient information to locate an error in a spreadsheet that would previously have been extremely difficult, if not impossible, to track down.

7.2 Appraisal

The criteria for success was to successfully address the following points:

- examining program comprehension theories and how they may be put to use in comprehending spreadsheets
- that spreadsheets can be represented in a virtual reality world
- and that it is possible to implement this technique (of visualising spreadsheets)

The criteria for success stated that three points must be met in order for this research to be a success. The first of these was to 'examine program comprehension theories and how they may be put to use in comprehending spreadsheets'. The theories of program comprehension were thoroughly investigated in chapter two, to extract information that could be useful in developing a visualisation technique. Indeed, chapter four describes concepts that are directly related to the program comprehension theories, for example: the visualisation concept details the philosophy of allowing the maintainer to be guided around the visualisation by their need to gather information. Just as was reported in chapter two, that this is exactly what happens in the initial stages of program comprehension. Further to this, by presenting the visualisation in a pictorial format, it is closer to the

maintainer's mental model of the system and thus reduces the cognitive load on the maintainer - improving the comprehension process.

The second point of the criteria for success was to substantiate 'that spreadsheets can be represented in a virtual reality world'. The visualisations presented in chapter five are proof that spreadsheets can be represented in virtual reality. This is the result of a multi-stage process that constructs visualisations by way of the spreadsheet description language. Although spreadsheets could have been represented in many different ways in virtual reality, a physical mock-up of the spreadsheet was chosen in order that a maintainer with previous knowledge of the spreadsheet would be able to bring that knowledge to the visualisation. Also, by presenting the maintainer with a familiar start point, the maintainer does not have to waste time by deciphering their location and their surroundings before they can begin comprehending the spreadsheet.

Point three of the criteria for success details the goal 'that it is possible to implement this technique (of visualising spreadsheets)'. Chapter five describes the implementation and chapter six evaluates it. The technique of using virtual reality to visualise spreadsheets was possible and was implemented. It was concluded that the visualisations were useful as common mistakes in spreadsheets could be located and corrected with little effort on the part of the maintainer. Without this visualisation technique these mistakes are more difficult to trace.

However, due to the limitations of the packages used, notably REND386, some of the features described in chapter four could not be implemented. Such as animation of data, data manipulation and the presentation of text. Although this did have an adverse effect on the total visualisation system hoping to be offered, visualisations were still of sufficient use to be able to aid understanding and debug spreadsheets.

The other major drawback, was the memory limitation imposed by REND386, although this is partly due to Microsoft's MS-DOS and the PC's architecture. Depending on memory usage at the time, only about 100 objects could be generated and displayed by REND386 in any one visualisation resulting in the fact that only small spreadsheets could be visualised. Thus, it is impossible to show that the model works, or is of any use for large spreadsheets as it is impossible to visualise them using the software employed for the research.

Even so, it is conjectured that this research is successful as it achieved its goal of addressing all the points of the criteria for success.

7.3 Future work

The research presented here is only the first stepping stone to greater achievements. As far as the author is aware, no task like this has been undertaken elsewhere. This work can be used to create visualisations for software by applying some of the techniques employed. Future work could also include the implementation of concepts, in this work, not developed due to software and hardware limitations, and also the creation of more advanced techniques for visualising software.

Obviously, one direction of future work would be to implement the ideas presented in chapter four that could not be implemented due to the limitations of the environment. Being able to include the presentation of text, spreadsheet slicing, and data manipulation and animation would be a major bonus to the visualisation so that these techniques could be evaluated to see what kind of a benefit they would bring to spreadsheet comprehension.

The one major drawback that would be nice to eradicate would be the memory restrictions imposed by the PC's architecture and software. Currently, only relatively small spreadsheets can be visualised - no more than about 100 objects in the virtual reality at once. It would be interesting to see what the limits of the visualisation would be before it, itself, became overloaded with information and what techniques could be employed to tackle this problem.

In the light of what has been achieved, the following extensions can be made.

Firstly, more advanced spreadsheets have a small programming language built into themselves. For example, *Excel* can have 'IF' statements within cells. Although these types of cells can be illustrated using the technique implemented already, it will be worth increasing the flexibility of the link visualisations, such that different colours (or even shapes) can be used to indicate that a particular cell includes an 'IF' statement and that certain links from that cell are connected to that 'IF' statement, such that only one and not both (or more) of the links are actually executed in the calculation.

Secondly, as in programming languages, the use of *types* can be visualised. For example, one column in a spreadsheet might be adding the amount of rainfall, whereas another might be adding costs. It would be ridiculous to add amounts of rainfall to monetary values, so the visualisation can include techniques for indicating the *type* of data that a cell currently contains.

Thirdly, chapter four details the concept of slicing, i.e. being able to select a cell and then temporarily remove from sight all cells more than 'x' number of links away from the selected cell. It would be interesting to be able to endow the visualiser with some abilities of its own to trace

dependencies throughout the graph and to highlight, say, all those cells that influence the output of a selected cell, or (perhaps) to highlight all those cells whose output is influenced by a selected input cell, and so on.

Finally, the notion of being able to edit the data and formulas and then have the visualisation write the changes back to the spreadsheet would allow the implementation of a virtual reality construction kit. This would, conceivably, include the means to generate RCL from the virtual reality system, then to modify *trans* to carry out the SDL to RCL process in reverse and then, finally, to reconstruct the spreadsheet from the SDL. Other methods, of course, could be employed such as making the virtual reality system generate the spreadsheet code straight from it.

Ideas for how the visualisation of programming languages could take form are; to begin with, the overall module hierarchies could be demonstrated by depicting only the modules in the system and linking those together that have a dependency on each other. In essence, this would result in a call graph being demonstrated. As a maintainer takes interest in a particular module, as he/she gets close to one, the module could be replaced with a more detailed visualisation depicting the number of procedures within the module and the links could be expanded in detail to show which other modules call which particular procedures. Then, as the maintainer selects a procedure, the software code could be visualised as a dependency graph, showing variable referencing, data flow and control flow. Finally, the actual code itself could be displayed as and when the maintainer requires.

The overall picture is to have more and more stages of detail as the maintainer requires it, not by displaying it all at once. By introducing the maintainer to detail over a period of time, the sheer information overload is greatly reduced.

As with the spreadsheet system above, the final attainment would be to be able to generate code from within the virtual reality visualisation, resulting in a virtual reality software construction kit. This would, conceivably, lead to the construction of software systems in virtual reality. For example, with object oriented software, the objects could be displayed in virtual reality and then connected by the viewer, resulting in the construction of software.

Acknowledgements

I would like to thank the following people:

My parents who have supported me (at the very least, financially!) throughout my degree years and have made all of this possible. *Thank you, and I promise that I will pay you back... in a few years!*

Malcolm Munro for all his support and advice on this thesis and the research undertaken to develop it. Also to Jill Munro for so ably sorting out the appearance of my 'references' section. *Thank you, now I might even stand a chance of passing!*

Thomas Green for all his help locating information about spreadsheets and their problems. *Thank you. Sorry that I didn't quite get around to making the video, but you never know...!*

Louise Hudson for conquering my inept ability to draw pictures. The fruits of her work can be viewed in figures 4.4 to 4.6 inclusive. *Thank you, and can I borrow your stapler, please?*

Nigel Scriven for advising me on the most amazing ways to get a picture into my thesis, as well as other annoying UNIX/MS-DOS problems. *Thanks, and could I possibly borrow those Windows disks just one more time?*

Nick and Catherine for always being on hand for a bit of advice (about anything!) and most of all for helping me come out on top in the job hunting process. *Thank you, and the next pavlova is on me (not literally you understand)! And just to insure that history never forgets us, long live the Durham Stealers, the finest ten-pin bowling team there ever was.*

Mark Lambert for his invaluable help with Microsoft products and for always being at the other end of an 'e-mail'. *Thank you, and I hope that your cunning plan works some day.*

Finally, last but by no means least, (in order of appearance!) Tara, Lindsay, Nancy-Ann, Michelle and Sarah for giving me one of the best holidays I've had and also when I needed it most: when I was writing my thesis! *Thank you.*

Amiga is a registered trademark of Commodore UK LTD.

Babylon 5 is a trademark of Babylonian Productions Inc.

Microsoft and **MS-DOS** are registered trademarks of the Microsoft Corporation.

Star Trek: The Next Generation, **U.S.S. Enterprise** and **U.S.S. Enterprise D** are all registered trademarks of Paramount Pictures.

Superscape is a registered trademark of Superscape LTD.

TopSpeed is a registered trademark of Jensen & Partners International.

References

[1] Macro A and Buxton J, *The Craft of Software Engineering,* Addison-Wesley Publishing Company, (1987).

[2] Sommerville I, *Software Engineering,* Addison-Wesley Publishing Company, 4th Edn., (1992).

[3] Pressman RS, *Software Engineering: A Practitioner's Approach,* McGraw-Hill, 3rd Edn., (1992).

[4] ANSI/IEEE, *Software Engineering Standards,* Wiley-Interscience, (1984).

[5] Foster JR, Jolly AEP and Norris MT, *An Overview of Software Maintenance,* British Telecom Technology Journal, Vol. 7, No. 4, pages 37-46, (1989).

[6] Robson DJ, Bennett KH, Cornelius BJ and Munro M, *Approaches to Program Comprehension,* Journal of Systems Software, Vol. 14, No. 2, pages 79-84, (1991).

[7] Cornelius BJ, Munro M and Robson DJ, *An approach to Software Maintenance Education,* Software Engineering Journal, Vol. 4, No. 4, pages 233-236, (1989).

[8] Arnold RS and Martin RJ, *Guest Editor's Introduction,* IEEE Software, Vol. 3, No. 3, pages 4-5, (1986).

[9] Hendry DG and Green TRG, *CogMap: A Visual Description Language for Spreadsheets,* Journal of Visual Languages and Computing, Vol. 4, No. 1, pages 35-54, (1993).

[10] Bennett KH, Cornelius BJ, Munro M and Robson DJ, *Software Maintenance: A Key Area for Research,* University Computing, Vol. 10, No. 4, pages 184-188, (1988).

[11] Munro M, Centre for Software Maintenance, University of Durham.

[12] Letovsky S and Soloway E, *Delocalized Plans and Program Comprehension,* IEEE Software, Vol. 3, No. 3, pages 41-49, (1986).

[13] Younger EJ and Bennett KH, *Model-Based Tools to Record Program Understanding,* Proceedings of the IEEE Second Workshop on Program Comprehension, IEEE Computer Society Press, pages 87-95, (1983).

[14] Gilmore DJ and Green TRG, *Programming Plans and Programming Expertise,* The Quarterly Journal of Experimental Psychology, Vol. 40, No. 3, pages 423-442, (1988).

[15] Osborne WM and Chikofsky EJ, *Fitting pieces to the Maintenance Puzzle,* IEEE Software, Vol. 7, No. 1, pages 11-12, (1990).

[16] Canning R, *The Maintenance 'Iceberg',* EDP Analyzer, Vol. 10, No. 10, (1972).

[17] Lientz BP and Swanson EB, *Software Maintenance Management,* Reading MA: Addison-Wesley, (1980).

[18] Standish TA, *An Essay on Software Re-use,* IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, pages 494-497, (1984).

[19] Brooks R, *Towards a Theory of the Comprehension of Computer Programs,* International Journal of Man-Machine Studies, Vol. 18, No. 6, pages 543-554, (1983).

[20] Schneiderman B and Mayer R, *Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results,* International Journal of Computer and Information Sciences, Vol. 8, No. 3, pages 219-238, (1979).

[21] Weiser M, *Program Slicing,* IEEE Transactions on Software Engineering, Vol. SE-10, No. 4, pages 352-357, (1984).

[22] Gallagher KB and Lyle JR, *Using Program Slicing in Software Maintenance,* IEEE Transactions on Software Engineering, Vol. 17, No. 8, pages 751-761, (1991).

[23] Weiser M, *Programmers use Slices when Debugging,* Communications of the ACM, Vol. 25, No. 7, pages 446-452, (1982).

[24] Wiedenbeck S, *Processes in Computer Program Comprehension,* Empirical Studies of Programmers, Ablex Publishing Corporation, 2nd printing, pages 48-57, (1987).

[25] Brooks R, *Using a Behavioral Theory of Program Comprehension in Software Engineering,* The Third International Conference on Software Engineering Proceedings, pages 196-201, (1978).

[26] Miara RJ, Musselman JA, Navarro JA and Schneiderman B, *Program Indentation and Comprehensibility,* Communications of the ACM, Vol. 26, No. 11, pages 861-867, (1983).

[27] Basili VR and Mills HD, *Understanding and Documenting Programs,* IEEE Transactions on Software Engineering, Vol. SE-8, No. 3, pages 270-283, (1982).

[28] Letovsky S, *Cognitive Processes in Program Comprehension,* Empirical Studies of Programmers, Ablex Publishing Corporation, 2nd printing, pages 58-79, (1987).

[29] Littman DC, Pinto J, Letovsky S and Soloway E, *Mental Models and Software Maintenance,* Empirical Studies of Programmers, Ablex Publishing Corporation, 2nd printing, pages 80-98, (1987).

[30] Soloway E, *Learning To Program = Learning To Construct Mechanisms And Explanations,* Communications of the ACM, Vol. 29, No. 9, pages 850-858, (1986).

[31] Johnson WL and Soloway E, *PROUST: Knowledge-Based Program Understanding,* IEEE Transactions on Software Engineering, Vol. SE-11, No. 3, pages 267-275, (1985).

[32] Soloway E and Ehrlich K, *Empirical Studies of Programming Knowledge,* IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, pages 595-609, (1984).

[33] Détienne F, *An Empirically-Derived Control Structure for the Process of Program Understanding,* International Journal of Man-Machine Studies, Vol. 33, No. 3, pages 323-342, (1990).

[34] Wiedenbeck S, *The Initial Stage of Program Comprehension,* International Journal of Man-Machine Studies, Vol. 35, No. 4, pages 517-540, (1991).

[35] Lientz BP, Swanson EB and Tompkins GE, *Characteristics of Application Software Maintenance,* Communications of the ACM, Vol. 21, No. 6, pages 466-471, (1978).

[36] Schneidewind NF, *The State of Software Maintenance,* IEEE Transactions on Software Engineering, Vol. SE-13, No. 3, pages 303-310, (1987).

[37] Schneiderman B, Shafer P, Simon R and Weldon L, *Display Strategies for Program Browsing: Concepts and Experiment,* IEEE Software, Vol. 3, No. 3, pages 7-15, (1986).

[38] Rajlich V, *Guest Editor's Introduction: Special Issue on Software Maintenance,* IEEE Transactions on Software Engineering, Vol. 18, No. 12, page 1037, (1992).

[39] Glass RL, *Editor's Corner: Software Maintenance is a Solution - Not a Problem,* Journal of Systems Software, Vol. 11, No. 2, pages 77-78, (1990).

[40] Genuchten M van, Brethouwer G, Boomen T van den and Heemstra F, *Empirical Study of Software Maintenance,* Information and Software Technology, Vol. 34, No. 8, pages 507-512, (1992).

[41] Laitinen K, *Natural Naming,* Technical Research Centre of Finland, talk given at the University of Durham (Department of Computer Science), (14th April 1994).

[42] Brooks F, *The Mythical Man-Month: Essays on Software Engineering,* Addison-Wesley Publishing Company, (1975).

[43] Roman G and Cox KC, *A Taxonomy of Program Visualization Systems,* Computer, Vol. 26, No. 12, pages 11-24, (1993).

[44] Price BA, Baecker RM and Small IS, *A Principled Taxonomy of Software Visualisation,* Journal of Visual Languages and Computing, Vol. 4, No. 1, pages 211-266, (1993).

[45] Isdale J, *What Is Virtual Reality? A Homebrew Introduction and Information Resource List,* Document available on the Internet, Version 2.1, (1993).

[46] Wellner P, Mackay W and Gold R, *Computer-Augmented Environments: Back To The Real World,* Communications of the ACM, Vol. 36, No. 7, pages 24-26, (1993).

[47] Adam JA, *Virtual Reality Is For Real,* IEEE Spectrum, Vol. 30, No. 10, pages 22-29, (1993).

[48] DeFanti TA, Sandin DJ and Cruz-Neira C, *A 'room' with a 'view',* IEEE Spectrum, Vol. 30, No. 10, pages 30-33, (1993).

[49] Hancock D, *'Prototyping' the Hubble fix*, IEEE Spectrum, Vol. 30, No. 10, pages 34-39, (1993).

[50] Pausch R, *Three Views of Virtual Reality: An overview*, Computer, Vol. 26, No. 2, pages 79-80, (1993).

[51] Robertson GG, Card SK and Mackinlay JD, *Nonimmersive virtual reality*, Computer, Vol. 26, No. 2, pages 81 and 83, (1993).

[52] Moshell M, *Virtual environments in the US military*, Computer, Vol. 26, No. 2, pages 81-82, (1993).

[53] Grimes J, *Virtual Reality 91 anticipates future reality*, IEEE Computer Graphics and Applications, Vol. 11, No. 6, pages 81-83, (1991).

[54] Walker GR, Rea PA, Whalley S, Hinds M and Kings NJ, *Visualisation of telecommunications network data*, British Telecom Technology Journal, Vol. 11, No. 4, pages 54-63, (1993).

[55] Machover C and Tice SE, *Virtual Reality*, IEEE Computer Graphics and Applications, Vol. 14, No. 1, pages 15-16, (1994).

[56] Stone RJ, *Reality - who needs it?*, IEEE Review, Vol. 39, No. 6, pages 243-246, (1993).

[57] Ribarsky W, Bolter J, Op den Bosch A and Van Teylingen R, *Visualization and Analysis Using Virtual Reality*, IEEE Computer Graphics and Applications, Vol. 14, No. 1, pages 10-12, (1994).

[58] Ellis SR, *What Are Virtual Environments?*, IEEE Computer Graphics and Applications, Vol. 14, No. 1, pages 17-22, (1994).

[59] Wells M, *An Introduction to VR*, Document available on the Internet, (1991).

[60] Earnshaw RA, Gigante MA and Jones H, *Virtual Reality Systems: Introduction*, Academic Press Limited - Harcourt Brace & Company (Publishers), (1993).

[61] Gigante MA, *Virtual Reality: Definitions, History and Applications*, Appears in [60], pages 3-14.

[62] Gigante MA, *Virtual Reality: Enabling Technologies*, Appears in [60], pages 15-25.

[63] Woolley B, *Virtual Worlds*, Blackwell Publishers, (1992).

[64] Helsel SK and Roth JP, *Virtual Reality: Theory, Practice and Promise*, Meckler Publishing, (1991).

[65] Stampe D, Roehl B and Eagan J, *Virtual Reality Creations*, Waite Group Press, (1993).

[66] Kay W, *The real future in Virtual Reality*, Article in the Sunday Express, pages 74-75, (September 12th 1993).

[67] Encarnação J, Göbel M and Rosenblum L, *European Activities in Virtual Reality*, IEEE Computer Graphics and Applications, Vol. 14, No. 1, pages 66-74, (1994).

[68] Kahaner D, *Japanese Activities in Virtual Reality*, IEEE Computer Graphics and Applications, Vol. 14, No. 1, pages 75-78, (1994).

[69] Kinloch D, *A Combined Representation for the Maintenance of C Programs*, PhD. Thesis (in preparation), Department of Computer Science, University of Durham.

[70] Psychology lectures given at the University of Durham, 1990/1.

[71] Saariluoma P and Sajaniemi J, *Visual information chunking in spreadsheet calculation*, International Journal of Man-Machine Studies, Vol. 30, No. 5, pages 475-488, (1989).

[72] Sedgewick R, *Algorithms*, Addison-Wesley Publishing Company, 2nd Edn., (1988).

[73] Miller LH, *Advanced Programming: Design and Structure Using Pascal*, Addison-Wesley Publishing Company, (1986).

[74] Microsoft Corporation, *Microsoft Excel Software Development Kit (Version 4)*, Microsoft Press, (1993).

[75] Kernighan BW and Ritchie DM, *The C Programming Language*, Prentice-Hall, 2nd Edn., (1988).

[76] Braben D, *Frontier: Elite II*, Konami (Publishers) and Gametek (Distributors), (1993).

[77] Bray P, *Beware the Curse of the Amateur Tinkerer*, Article in the Daily Telegraph, (September 15th 1992).

[78] Person R, *Excel Tips, Tricks, and Traps*, Que Corporation, (1989).

[79] Hendry DG and Green TRG, *Creating, comprehending and explaining spreadsheets: a cognitive interpretation of what discretionary users think of the spreadsheet model*, (in preparation), (1994).

[80] McKee JR, *Maintenance as a function of design*, Proceedings of the 1984 AFIPS National Computer Conference, pages 187-193, (1984).

[81] Lehman MM and Belady L, *Program Evolution. Processes of Software Change*, London Academic Press, (1985).

[82] Aukstakalnis S and Blatner D, *Silicon Mirage: The Art and Science of Virtual Reality*, Peach Pit Press, (1992).