Visualising Software in Cyberspace

Peter Young

Ph.D. Thesis

Research Institute in Software Evolution

Department of Computer Science

University of Durham

October 1999

Abstract

The problems of maintaining software systems are well documented. The increasing size and complexity of modern software serves only to worsen matters. Software maintainers are typically confronted with very large and very complex software systems, of which they may have little or no prior knowledge. At this stage they will normally have some maintenance task to perform, though possibly little indication of where or how to start. They need to investigate and understand the software to some extent in order to begin maintenance. This understanding process is termed program comprehension.

There are various theories on program comprehension, many of which put emphasis on the construction of a mental model of the software within the mind of the maintainer. These same theories hypothesise a number of techniques employed by the maintainer for the creation and revision of this mental model. Software visualisation attempts to provide tool support for generating, supplementing and verifying the maintainer's mental model.

The majority of software visualisations to date have concentrated on producing two dimensional representations and animations of various aspects of a software system. Very little work has been performed previously regarding the issues involved in visualising software within a virtual reality environment. This research represents a significant first step into this exciting field and offers insight into the problems posed by this new media.

This thesis provides an identification of the possibilities afforded by 3D graphics for software visualisation and program comprehension. It begins by defining seven key areas of 3D software visualisation, followed by the definition of two terms, visualisation and representation. These two terms provide a conceptual division between a visualisation and the elements of which it is comprised. This division enables improved discussion of the properties of a 3D visualisation and particularly the identification of properties that are desirable for a successful visualisation. A number of such desirable properties are suggested for both visualisations and representations, providing support for the design and evaluation of a 3D software visualisation system.

Also presented are a number of prototype visualisations, each providing a different approach to the visualisation of a software system. The prototypes help demonstrate the practicalities and feasibility of 3D software visualisation. Evaluation of these prototypes is performed using a variety of techniques, the results of which emphasise the fact that there is substantial potential for the application of 3D graphics and virtual reality to software visualisation.

Acknowledgements

I would not have been able to produce this thesis without the direct help and support of a number of people. There are also many others who have unwittingly contributed by providing me with an interesting and enjoyable time throughout my research. For those not named below, I thank you all.

First and foremost, this thesis is dedicated to:

my fiancée Joanne,
my family Brian, Diane and Mark,
my late grand-parents, Don and Peggy,
and my grand-parents, Frans and Louise.

Joanne deserves a further mention for her continued support throughout my years of studying, her tolerance of my erratic work patterns, and for understanding the pressure I have been under. I will never be able to fully repay the debt I owe her.

My parents have also provided continual support and inspiration throughout my life. Without their guidance and desire for me to succeed, I would certainly not have made it this far in academia.

I would like to thank Malcolm Munro for being both an excellent supervisor and a very good friend. His continual support and desire for me to enjoy whatever I choose to do, has made my stay at Durham an enjoyable and enlightening experience.

My profound thanks must also go to Joe Wood, Claire Knight, Liz Burd, James Ingham (have you finished reading it yet?), and my mother for proof reading this thesis and offering comments, constructive criticism and continual support.

The following should all be read with an implicit smiley :-) Steve 'Shabba' Glover and Alex 'West Ham' Jones both deserve thanks for tolerating sharing an office with me, though at times I often wonder just who suffered the most. Little Jimmy Ingham and Pete Biggs deserve a mention for their friendship and shared enjoyment of multiplayer gaming, without them my productivity would have doubled. Claire Knight, Liz Burd and Pui-Shan Chan have provided superb support and I will miss our 'discussions' on the merits of 3D and VR. All of the staff and students at the Department of Computer Science have been excellent colleagues and friends throughout my stay at Durham. Without all of these people, and the many more who shall remain nameless, my life at university would have been considerably less enjoyable.

This work was funded by an EPSRC award in conjunction with British Telecommunications Plc, both of which I owe greatly for the foresight to fund this research. Finally, I wish to thank my examiners, Marc Roper and Rachel McCrindle, for a friendly, professional and thorough approach to my viva.

Copyright

Declaration

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

This research has been documented, in part, within the following publications:

- **P. Young** and **M. Munro**, *A New View of Call-Graphs for Visualising Code Structure*, Technical Report 03/97, The Centre for Software Maintenance, University of Durham, January 1997.
- **P. Young** and **M. Munro**, *Visualising Software in Virtual Reality*, 6th International Workshop on Program Comprehension (IWPC'98), pages 19-26, Ischia, Italy, June 1998.
- **P. Young** and **M. Munro**, *3D Software Visualisation*, 1st International Conference on Visual Representations and Interpretations (VRI'98), Liverpool, UK, September 1998.

Table of contents

List of figures

Chapter 2

Chapter 3

Chapter 4

Chapter 5

Chapter 6

List of tables

Chapter 1.

Introduction

1.1 Research background

It is a widely accepted fact that software maintenance consumes the majority of resources within the software life cycle [Sommerville92, Standish84]. The high cost in time, money and manpower incurred by maintenance is usually attributed to shortcomings in the design and implementation cycle. A number of factors such as project deadlines, cost overruns and inexperience often result in hurried, badly written and badly documented software systems. Such systems pose significant problems during maintenance.

Regardless of the maintenance task being performed, it is highly likely that the maintainer will require at least some understanding of the code. This understanding often comes from a hard fought battle in which the maintainer must attempt to rediscover the intentions and design issues behind the code, typically from scarce resources and, in the worst case, from the source code alone. For this reason, program comprehension is generally accepted as being the central task within software maintenance.

Visualisation, and more specifically computer generated visualisation, is used widely in almost all areas of human endeavour, with often significant gains. One example is the field of scientific and data oriented visualisation which makes extensive use of computer generated images to view complex data sets and visualise phenomena that are otherwise invisible. Applying visualisation in this manner often reveals trends and irregularities, which may be missed when looking at the raw data. One area which does not benefit from visualisation to any large degree is the role of the software engineer or maintainer. It is strange that the technology, which makes possible[1] such a powerful and widely used tool as visualisation, should benefit so little. The main reason for this lack of useful visualisation in software engineering is the abstract nature of the information being presented. Software systems are inherently abstract structures, they are information artefacts and have no physical form or intuitive appearance. For this reason, software systems are very difficult to visualise.

Various theories of program comprehension put emphasis on the construction of a mental model of the software within the mind of the maintainer. These same theories hypothesise a number of techniques employed by the maintainer in the construction of this mental model. Software visualisation attempts to

[1] Computers only make the use of visualisation practical; the technique is not dependent on the technology.

provide tool support for generating, supplementing and verifying the maintainer's mental model or understanding.

The mental models proposed by various researchers in the field of program comprehension have a commonality in that they are all composed from semantic constructs. These constructs are typically abstractions, at various levels, of program features. The network formed from these constructs constitutes the maintainer's understanding and representation of the program. Software visualisation attempts to provide a mapping from the program code to a visual (and other media) representation, which matches the maintainer's mental model as closely as possible. Highlighting and automatically creating these semantic constructs would remove a great deal of effort from the maintainer, specifically they would require less time scanning the source code. The field of software visualisation is thus considered an important area of research within the field of program comprehension.

Software visualisation is concerned with all areas of generating and manipulating these graphical models. The majority of software visualisation systems to date have concentrated on producing two dimensional representations and animations of various aspects of a software system. In addition, a growing area of research is investigating the application of 3D graphics and virtual reality (VR) technology to software visualisation. This research into 3D software visualisation attempts to address the problem both in terms of what data to represent and the user's ability to navigate this data. The former aspect stems from a growing need for more expressive ways to represent the inherently multidimensional data, facts and relationships that constitute a software system. The latter plays upon the naturally developed capabilities which humans possess for navigating and interacting within 3D environments.

Information visualisation (IV) is concerned with the visualisation and representation of semantic information. Often confused with the field of data visualisation, which represents numeric and scientific data, information visualisation concentrates on more abstract components that have no inherent visual representations, for example the structure of a knowledge base or document collection. A significant amount of research has investigated the application of 3D graphics and VR technology to information visualisation. Software systems can be considered as simply another collection of abstract data, facts and relations: as information. With this in mind, it should be possible for techniques and ideas generated from research into 3D information visualisation to be applied to the visualisation of software systems.

Virtual reality has received an enormous amount of public attention recently. Along with this attention have come exaggerated claims, unrealistic expectations, conflicting and imprecise terminology, and a large degree of uninformed commentary [Gigante93]. Research into VR techniques and systems has also grown progressively over recent years, with a great many systems being produced. There are a large number of definitions of the term VR, many of which place emphasis on different aspects of the reality illusion such as the display, interface, and level of immersion. Purists argue that the goal is to provide a 'suspension of disbelief', an environment in which the user is not distracted by flaws or

inadequacies which point to its artificial nature. A flexible though possibly under-defined view of VR is given by Adam:

> "A combination of various interface technologies that enables a user to intuitively interact with an immersive and dynamic computer-generated environment." [Adam93]

This definition dilutes the preconception that VR must involve certain technologies such as head-mounted displays (HMDs), data gloves and tactile feedback systems, allowing a more open interpretation. The definition also removes one implicit assumption often made from the term VR: that the goal is to simulate reality. This allows for more creative applications such as the visualisation of abstract data and the creation of environments which do not adhere to the restrictions of reality.

The term *cyberspace* was originally coined by William Gibson in his science fiction short story, *Burning Chrome* [Gibson93]. A number of researchers have since proposed ideas and suggestions on how cyberspace could be implemented and on the characteristics of its environment [Benedikt91]. Gibson depicts cyberspace as a populated, global, information abstraction and interaction mechanism in which all data transactions and operations take place. While Gibson's view is very much fictional and far beyond the scope of current technologies, the term *cyberspace* and the notion of abstract data spaces suit well a marriage between the fields of VR and information (including software) visualisation. VR deals predominantly with the creation of an illusion of immersion within a realistic, although computer generated, environment. Conversely, the term cyberspace will be taken here as the use of 3D and VR technologies to represent abstract information: to create an information environment. The illusion of reality is not a necessary feature here. The data being represented has no physical form or appearance, it merely has the abstraction and representation given to it by the designer of the cyberspace.

The term *cyberspace,* as defined in this thesis, describes the use of 3D graphics and virtual reality (VR) techniques in creating abstract visualisations of semantic information. The goal of this research is to investigate the process of creating useful software visualisations within cyberspace; to investigate the application of 3D graphics and VR technology to software visualisation. Visualisation allows us to endow software with a physical (visual) form, or rather, map it to an artefact which can be mentally reasoned and manipulated in a similar manner to which humans reason about real world objects. Such a representation would allow software engineers to make use of all their natural, intuitive reasoning and perceptual skills to aid in the software maintenance process. It may also be the case that, if software is given such a form, then patterns or visual design rules may emerge. Much in the same way that the properties of a chemical molecule or compound can be derived from structural information, software quality or structural properties may become apparent from the use of software visualisation.

1.2 Problem statement

Software systems, and in particular the large, highly inter-related information bases that they constitute, are inherently multidimensional and thus difficult to visualise [Brooks87]. Current software visualisation systems typically isolate one aspect or view of this information for display, effectively

hiding all remaining information. This isolated view generally consists of a large and complex directed graph, which is displayed on a 2D canvas using one of the many layout algorithms available. Regardless of the quality of the layout algorithm or viewing technique used, the user is often presented with an incomprehensible birds-nest graph. Alternatively, they are forced to reduce the amount of information displayed, thus removing potentially important information from view [Fairchild88]. While the use of 3D or VR technology cannot immediately solve this problem, it does offer a number of advantages, for example:

- greater working volume for information presentation (with respect to displayable space);
- focus, context and point of interest managed dynamically by the user's viewpoint on the information;
- greater flexibility for representation, organisation and presentation of the information.

Although the full potential of using three-dimensional views and interaction techniques has not yet been realised, a number of software systems have been developed to investigate its worth. In general, these systems do not make full use of the 3D graphics or advance software visualisation to any great extent. The majority simply extend and adapt already established two-dimensional visualisation techniques into 3D. These techniques consist mainly of displaying a graph or network structure showing the relationships between components in a software system. The visualisations rarely extend beyond the simple node-link representations evident in almost all software visualisation systems to date. This is not necessarily a bad thing, graph representations are extremely useful for displaying such relationships and dependencies. However, the field of software visualisation should not be restricted to solely one view of a software system, other possibilities must be investigated.

The intention of this research is to investigate the application of 3D graphics and VR technology to the visualisation of software systems. Graph structures are useful for small collections of information, however, solutions must still be found to enable the effective management of the large collections of information which are common to modern software systems. This research will investigate other representations, visual abstractions and appropriate metaphors which make full use of the 3D virtual environment afforded by cyberspace. The use of graph representations will not be abandoned or ignored, as these are integral parts of the software structure. Investigation will be made into how these structures or visualisations can be improved, augmented or superseded using more intuitive, truly three-dimensional representations.

There are a number of facets to this research:

- **Identifying which aspects of a software system to display.**
  Software systems consist of many different components at varying levels of abstraction. These should be depicted clearly with relationships between components also shown.
- **Establishing useful and meaningful representations.**
  What is the best representation to convey the information clearly, concisely and efficiently?

- **Investigating the problems of scale.**
  Meaningful visualisations of both single components and variable size collections of components are needed.

- **Introducing a greater level of flexibility.**
  Techniques enabling the user to customise and control the visualisation should be investigated.

- **Evaluation of visualisations.**
  A method for evaluating the effectiveness of visualisations produced must also be investigated. At present there is little empirical evaluation of software visualisation tools.

It is most important not to forget the actual role of software visualisation within this research. The aim is to provide useful and meaningful visualisations to aid program comprehension or software maintenance, not simply to produce pretty pictures.

To summarise, the intention of this research is to investigate the application of 3D graphics and VR technology to the visualisation of software systems. This will include research into new representations, visual abstractions and appropriate metaphors within the 3D environment. The object of this study is not to produce a fully functional software visualisation tool, but rather to demonstrate the feasibility and potential for such a tool. The goal is the identification and evaluation of new visualisations and the development of a concept demonstrator, which can be used both for the evaluation and discussion of these visualisations.

1.3 Software visualisation in cyberspace

The research work described within this thesis investigates the application of 3D graphics and VR technology to software visualisation. Three-dimensional software visualisation is a direct extension of, and one could say is included within, the field of software visualisation. A simple assumption could be made that 3D software visualisation is merely an extension of current 2D visualisation techniques into a 3D environment. Such an assumption would, however, be wrong. The use of 3D graphics and VR technology provides a completely new working environment, one so different from traditional 2D interfaces that the capabilities, requirements and issues involved are mostly non-transferable.

3D software visualisation not only draws on some of the goals and ideas of software visualisation, but it also draws upon other areas such as information visualisation which has seen many research projects investigating the use of 3D graphics for visualisation. The field of 3D information visualisation addresses many of the issues involved with large, complex and inherently abstract information structures, all of these terms being easily used to describe a typical software system.

This research is concerned with the structural visualisation of software systems. Such visualisations provide a gross overview of a software system, presenting various information on the many constituent components. One approach is to construct an environment or landscape with features directly related to the software system. A software engineer is then able to navigate within the software, identifying important or interesting features merely by surveying his or her surroundings.

This research focuses primarily upon single-user software visualisations yet provides occasional insight and concepts for use within a multi-user environment. Multi-user software visualisations would constitute a natural extension of this work. Software maintainers rarely work alone, more often they work as a team. Being able to support a team-orientated approach within these visualisations would be of great value, but is beyond the scope of this research.

1.4 Criteria for success

The goals of this research span both the theoretical and the practical.

The theoretical aspect investigates the issues associated with 3D software visualisation. The aim is to identify the key areas of 3D software visualisation, and to investigate what constitutes a good or bad 3D software visualisation. These factors are considered important for both the design and evaluation of such visualisations. The criteria for success are therefore:

A. The identification of the key aspects of 3D software visualisation.

B. The development of a framework to assist in both the design and evaluation of a 3D software visualisation.

The practical aspect of this research involves the production of a number of prototypes, providing demonstration of the concepts and the practicalities of 3D software visualisation. The success of this research will not simply be determined by the quality of the visualisations produced, it will be deemed a success if it isolates a number of new metaphors or representations for 3D software visualisation. Whether these visualisations are better than conventional 2D or 3D representations is not the issue, providing that their relative worth is clearly documented here. Further criteria for success are:

C. Demonstrating the possibility of a move away from traditional graph-based visualisations of software systems by providing a more intuitive visualisation which makes full use of the additional flexibility afforded by VR.

D. The representations or abstractions used must be intuitive and easily understandable (with practice).

E. The visualisations should scale up satisfactorily, allowing visualisation of both small programs and large software systems.

F. A suitable level of automation, or the possibility for automation, must be demonstrated.

An evaluation of this research against these criteria for success is provided in Chapter 7.

1.5 Thesis structure

The remainder of this thesis is structured as follows:

Chapter 2 presents a review of research within the fields of program comprehension and software visualisation. This begins with a synopsis of various program comprehension theories, which attempt to explain how a software maintainer extracts information from the source code. Following this is a brief introduction to the field of software visualisation, including a definition of the term software visualisation and an identification of more specific areas within it.

Chapter 3 focuses on virtual reality systems and 3D information visualisation. The chapter begins with an introduction to virtual reality systems and an explanation of what software and hardware constitutes such a system. The primary content of this chapter is an authoritative review of 3D information visualisation, including both the techniques and various prototype systems developed. This review of 3D information visualisation also contains details of a small number of 3D software visualisation systems. The content of Chapter 3 is extensive but provides a valuable insight into a relatively new research field, one very closely related to the subject of this thesis.

Chapter 4 presents the main theoretical work within this thesis, presenting a framework for the visualisation of software in cyberspace. The chapter begins by identifying seven key areas of 3D software visualisation. These seven areas must be considered when creating such a visualisation and provide a basis for describing various qualities of a visualisation. This is followed by a definition of the two terms, 'visualisation' and 'representation', which provide a framework for improved discussion of the various properties of a visualisation. The desirable properties of both visualisations and representations are then presented. These assist both the design and evaluation of a 3D software visualisation. Finally, the chapter concludes with a detailed example of some of the concepts involved, with reference to a prototype software visualisation developed within this research.

Chapter 5 presents the results of this research by way of a number of prototype visualisations and concept demonstrators. These prototypes were developed throughout the course of this research, providing a practical investigation of the various possibilities afforded by 3D software visualisation. This chapter concentrates purely on a description of the prototypes, leaving the evaluation of their merits and deficiencies to Chapter 6.

Chapter 6 provides an evaluation of the prototypes described in Chapter 5. This evaluation takes a variety of forms, including informal discussion, discussion against the seven key areas (identified in Chapter 4), evaluation against the desirable properties and also evaluation against a framework of cognitive design elements. All of the concept demonstrators described in Chapter 5 are evaluated in some form, with a more detailed investigation performed on one particularly well-developed prototype.

Chapter 7 presents a summary and conclusion of this research and evaluates its success against the criteria defined in Section 1.4.

Chapter 2.

Program comprehension and software visualisation

2.1 Introduction

This chapter presents a review of research in the fields of program comprehension and software visualisation. Program comprehension is the process of understanding any aspect of the operation of a software system. This can include understanding the operation of a program, the structure of the system, the various programming techniques used, or the algorithms involved.

Program comprehension can often be thought of as the overall goal or purpose of software visualisation. Software visualisation attempts to convey information about a software system with the goal of imparting some knowledge of the construction, structure and workings of that system onto the maintainer. Software visualisation is another medium to support the understanding of a software system.

Section 2.2 presents a review of various program comprehension theories that attempt to explain how a software engineer extracts information from the source code. Most of these theories are the result of observational studies in which programmers are presented with a set task. Their actions and understanding of the software are then documented in various ways. Many of these observational experiments support the notion that program comprehension is performed using either a top-down or bottom-up approach. Some researchers [Chan97, Letovsky86a, Mayrhauser94, Mayrhauser95] believe that the comprehension process takes an opportunistic approach where maintainers will utilise both top-down and bottom-up techniques as appropriate.

Section 2.3 provides an introduction to the field of software visualisation. Various definitions of software visualisation are discussed, as are the relationships between specialised areas within the field. 2D software visualisation systems are numerous and have been described to a great extent in other publications and taxonomies [Chan98, Myers90, Price93, Roman93, Stasko92b, Shu88]. The area of 3D software visualisation is also introduced, though further discussion on this more specialised area of software visualisation is provided in Chapter 3.

2.2 Program comprehension

Program comprehension is widely recognised as the main activity within many software maintenance or even software development tasks. Particularly within software maintenance activities, program comprehension is the vital first task of any maintenance, whether it be perfective, adaptive, corrective or preventative [Mayrhauser95]. Despite being such a key activity, program comprehension has not received much research activity until relatively recently with work from a number of key authors. This initial flurry of activity has since receded and, again, research into program comprehension has made little major progress. The research effort has now distributed over a larger number of related 'spin-off' research areas, software visualisation being one such example.

This section summarises a selection of the key theories developed in program comprehension and highlights any commonalties between them. Throughout this thesis the term *maintainer* will be used wherever possible to describe the person performing the comprehension task. The reason for this choice over the more popular term *programmer* is due to the almost total monopoly which software maintenance has on program comprehension activity. Thus, the term maintainer removes the generality and places program comprehension within its typical context. Another terminology issue is the use of the terms *cognitive model* and *mental model*. Within this thesis, the term *mental model* will refer to the maintainer's internal (mental) representation of the software or results of the comprehension process. The term *cognitive model* will refer to the complete set of processes, knowledge, heuristics and mental model used during program comprehension. Thus it is the cognitive model which is the subject of the various theories and the focus of this section.

Various theories [Basili82, Brooks77, Brooks83, Letovsky86a, Letovsky86b, Littman86, Robson88, Shneiderman79, Shneiderman80, Soloway84, Soloway88] differ in detail but share a common structure on how programs are comprehended. That is, they all subscribe to the idea of a knowledge base, mental model and some form of assimilation process (each described below). The exact representation and formation of these components does, however, vary throughout the theories.

2.2.1 Cognitive Models

This section aims to provide an overview of a number of program comprehension theories, highlighting terminology and key factors from each theory.

Mayrhauser [Mayrhauser95] identifies a commonality between the various program comprehension theories. This commonality exists in the maintainer's cognitive model, and to some extent in the way the model is created and maintained. All the comprehension theories agree that the program comprehension process uses existing knowledge coupled with a comprehension strategy in order to acquire new knowledge. This combined knowledge should, through a process of additions and revisions, eventually achieve the required level of understanding of the code. The various comprehension strategies used in obtaining this new knowledge have a commonality in that they all

formulate hypotheses which can then be refined, revised or rejected. Formulating and verifying these hypotheses makes use of both existing knowledge and the newly acquired knowledge.

The three main components in the cognitive model are the knowledge base, mental model and assimilation process. The *knowledge base* contains the maintainer's understanding of the domain and any related fields. The *mental model*, which is different from the cognitive model, is the maintainer's internal representation of the program. Finally, the *assimilation process* is the glue that binds the knowledge base to the mental model, manipulating both to create and revise hypotheses that can then be used to update the knowledge base or mental model. The actual processes involved in this assimilation are the subject of various theories of program comprehension.

The three main components of the cognitive model, which are the knowledge base, mental model and assimilation process will now be described in greater detail.

I. Knowledge base

The knowledge base consists of both existing knowledge and newly acquired knowledge. This knowledge will consist of two main types, general knowledge and task-specific knowledge. During the comprehension process the maintainer will obtain more task-specific knowledge, but may also require additional general knowledge that will typically be obtained on an 'as-needed' basis. General knowledge represents information not specifically related to the software itself. Such knowledge could include experience with programming languages, understanding of the implementation platform, or even knowledge of the operational domain of the software, for example business rules, control systems or database systems. Task specific knowledge is directly related with the comprehension task and will include knowledge such as design information, programming styles used during implementation and the overall goals of the program.

Existing knowledge and general knowledge are strongly related, with the majority of existing knowledge being non-task related. However, if the maintainer has prior knowledge of the program then the existing knowledge may contain fragments of task-specific information related to the program, including any mental models of the software. In contrast, new knowledge is oriented more towards task specific knowledge that is gleaned during the comprehension process. This new knowledge could include information on control flow, data flow, software structure, implementation details or programming styles. New knowledge can be obtained at any abstraction level and can range from low level implementation details to higher level structural information.

II. Mental model

The mental model is the maintainer's internal representation of the program and can be composed of a number of inter-dependent semantic constructs. The various theories of program comprehension proposed offer a wide variety of forms for these semantic constructs. Constructs can include *text structures*, *chunks*, *plans*, *hypotheses*, *conjectures*, *beacons*, and *rules of discourse*. The mental model is

continuously updated and amended by the assimilation process. Mayrhauser [Mayrhauser95] identifies the dynamic process used in creating these semantic constructs as a combination of behaviours such as *strategies*, *actions*, *episodes*, and *processes*.

Text structure constructs are formed from the program source code and its structure. In well-presented or styled code, an initial structure can be derived from the text layout using indentation, comment blocks or blank areas to identify textual chunks. This initial structure can then be refined further to give it a more syntactic meaning such as function blocks, iteration blocks, and conditional constructs. Organisation of text structures within the mental model can include nesting and dependencies. For example, an iteration block may be nested or contained within a function block, and two function blocks may be related by a function call.

Chunks are syntactic or semantic abstractions of text structures within the source code. Shneiderman [Shneiderman80] describes a process in which maintainers abstract portions of the source code (text structures) into chunks. Collections of these chunks can then be abstracted further into higher level chunks. Chunks can be constructed from either semantic or syntactic knowledge and need not be fully understood or identified immediately, being returned to at a later time for revision. It is reasonable to assume that a chunk formed on the basis of syntactic knowledge, a syntactic chunk, will be revised later to form a semantic chunk. For example, a syntactic chunk derived from a function block or a nesting level will be revised later with semantic meaning or purpose, such as a sort algorithm.

Plans, best described by Mayrhauser [Mayrhauser95], are "knowledge elements for developing and validating expectations, interpretations, and inferences". Plans, otherwise known as *clichés* or *schemas*, are derived from research by Soloway and Ehrlich [Soloway84] that attempted to apply theories from text comprehension to program comprehension. The notion of plans proposed by Soloway and Ehrlich corresponds directly to the notion of schemas in text comprehension, where schemas are "generic knowledge structures that guide the comprehender's interpretations, inferences, expectations, and attention when passages are comprehended" [Graesser81]. Similarly, plans in the context of program comprehension also capture the maintainer's attention during the comprehension process.

Plans can be further categorised into *programming plans* and *domain plans* [Mayrhauser95]. A simple analogy can be drawn between these types of plan and the knowledge contained within the knowledge base. Programming plans, analogous to task specific knowledge, are plans specific to the task in hand. They can exist at a variety of abstraction levels, ranging from the low-level program code to the abstract function of the program as a whole. Domain plans, analogous to general knowledge, are plans that incorporate all other knowledge of the problem area excluding that contained within the program plans (i.e. the detail of the software). Domain plans give a general concept of the task and how it can be completed. These generally exist at a higher level of abstraction than the actual software system.

Hypotheses are introduced by Brooks [Brooks83] suggesting that programmers use an iterative, hierarchical method of forming, refining and possibly rejecting hypotheses at varying levels of

abstraction during the comprehension process. Brooks states that the program comprehension process is complete when the maintainer's mental model contains a complete hierarchy of correct hypotheses. This hypothesis hierarchy is described as a bridge between various domain levels of the problem, the complete model thus bridging from the high level "This program is a ..." hypothesis to the low level hypotheses regarding implementation details.

Letovsky [Letovsky86a] investigates the use of hypotheses further in an experiment involving the study of maintainers undergoing a given comprehension task. Letovsky refers to hypotheses as *conjectures* and describes the mental processes of the maintainers as they create, question and revise these conjectures. Conjectures are associated with a degree of certainty that affects the way in which they are revised or discarded at a later stage. The level of certainty can range from a guess to complete certainty. From the study, Letovsky identifies three main types of conjecture:

- *Why* conjectures are hypotheses that question design choices, implementation decisions or function purpose. For example "Why use a bubble sort?";
- *How* conjectures are hypotheses which attempt to discover the method with which to accomplish a particular task. For example "How is the array indexed?";
- *What* conjectures hypothesise classification, for example a variable, function or variable type.

The latter two components of the mental model, *beacons* and *rules of discourse*, are described in the following section.

III. Assimilation and knowledge acquisition

The assimilation process is the method that the maintainer uses in undergoing the comprehension task. This process makes use of the mental model and the knowledge base, in conjunction with the source code and any related documentation, to build upon or revise the mental model and thus the maintainer's understanding of the program. The exact assimilation process used by the maintainer is the subject of a number of research papers, discussed in greater detail below. A number of methods or processes can be identified which contribute to the assimilation process and belong to various comprehension theories. Mayrhauser [Mayrhauser95] classifies these as being dynamic elements of the mental model, however they appear more fitting as a process by which the mental model is created. Before describing these processes, two features which aid in facilitating them, *beacons* and *rules of discourse*, will be described.

Beacons, first identified by Brooks [Brooks83] then explored further by Wiedenbeck [Wiedenbeck86, Wiedenbeck91], are recognisable or familiar features within the source code or other forms of (possibly higher level) knowledge. Beacons thus act as cues to the presence of certain structures or features and can be used in either the verification or creation of hypotheses. Brooks' well-cited example of a beacon is the swapping of two variables. This variable swapping could be a beacon for a sorting routine. Even at a higher level, beacons such as the name of a particular procedure or source module are equally valid

and useful information. Beacons are used in gaining a higher level of understanding than that in which the beacon occurs, i.e. abstracting from a code chunk to a process or function.

Rules of discourse, as identified by Soloway and Ehrlich [Soloway84], are rules or conventions within programming such as code presentation or naming standards. For example one rule of discourse is that a variable name should usually agree with its function. Rules of discourse set up expectations in the mind of the maintainer. These expectations can be very powerful, allowing the pre-fetching of knowledge from long term storage, though they can also be problematic if discourse rules are violated. Soloway and Ehrlich [Soloway84] describe an experiment which shows that comprehension performance is greater for programs which obey the rules of discourse (plan-like programs). Conversely the study also shows that when the rules of discourse are violated, then comprehension performance is hampered, both for expert and novice maintainers. This implies that unconventional algorithms, unusual data structures or unexpected use of a known algorithm as well as unconventional programming styles will render programs more difficult to comprehend.

Strategies define the sequence of actions performed by a maintainer while following a plan to achieve a particular goal. Littman *et al* [Littman86] identify two such strategies, the *systematic* strategy and the *as-needed* strategy. These two strategies are defined within the context of comprehending an entire program, hence the systematic approach tries to comprehend the program as a whole before any attempt at maintenance is made. The as-needed approach attempts to localise the comprehension process to only the areas of the program that must be understood to perform the maintenance task.

Two comprehension mechanisms that are aided by strategies are *chunking* and *cross-referencing*. The chunking process creates new, higher level chunks from lower level structures or chunks. As structures at the lower level are recognised they are abstracted into a chunk and labelled. These labels can then be used again in forming new, higher level chunks. An example of the chunking process could be the formation of the 'bubble sort' chunk from iteration, comparison and variable swapping chunks. Cross-referencing forms relations between different abstraction levels by mapping program parts to functional descriptions. Cross-referencing is essential to building a mental representation spanning a number of abstraction levels.

2.2.2 Summary

This section has summarised a number of key theories developed for program comprehension. These theories vary significantly, yet all maintain a high level of commonality in the structure of the cognitive model. This commonality can be summarised in that all theories subscribe to the idea of the cognitive model being composed of a knowledge base, mental model and assimilation process. The exact interpretation and composition of each of these components is the subject of the various theories.

It is well known that program comprehension plays a critical role in the majority of software engineering activities, particularly in software maintenance. The level of program comprehension

required can vary greatly depending on the task undertaken and the complexity, quality and size of the target system. Similarly, program comprehension is not limited solely to extracting information from the source code alone. Any form of system documentation or previous maintenance records are used as an integral part of the comprehension process, if available. However, more often than not this documentation will be inaccurate, inconsistent, incomplete or even non-existent. This leaves the maintainer with the complex task of attempting to recover not only the operation and structure of the software system, but many design issues and implicit assumptions made during its implementation. All this information must be recovered from the source code alone. Clearly this is a non-trivial task. Program comprehension research is thus vital to discover methods or tools that could aid maintainers to recover the intentions behind the code.

One point of note which can be derived from this section is that although the actual comprehension process can be observed, described and predicted, what is really needed are tools to aid or supplant this process. This aid could affect any or several of the main areas of the cognitive model. Systems or databases could extend the maintainer's knowledge base. Visualisation or automated program comprehension tools could enable swifter construction or verification of their mental model. Sophisticated searching and cross-referencing mechanisms could aid the assimilation process by integrating other tools. Suffice to say that there is a large scope for automation and computer supported work in program comprehension tasks, and more generally in software maintenance.

2.3 Software visualisation

Software visualisation is one area of research that is attempting to aid program comprehension and, more generally, software maintenance. Software visualisation makes use of the fact that the human brain is adept at processing, manipulating and recognising visual images and structures. Reading text is a special case of this visual processing, however, this is translated or interpreted at a character, word, or possibly even phrase level. Such detail is too great and the brain will abstract each character, word or sentence into an internal meaning or representation. This is described in the program comprehension literature at a higher level, for example beacons [Brooks83] and chunks [Shneiderman80] are abstract internal representations of low-level textual structures.

Software visualisation attempts to aid the comprehension process by providing these abstractions in a visual form, which is hoped will reduce the interpretation load. By presenting the viewer with a pictorial model of more complex lower-level information, he or she can rapidly generate an initial mental model of the software and use this as a basis for further investigation. The field of software visualisation is concerned with all areas of generating and manipulating these graphical models.

The following sections review various literature regarding software visualisation and highlight specific points of interest. This begins with an identification of the need for software visualisation, reiterating some of the issues that were briefly mentioned in Chapter 1. Section 2.3.2 then addresses the definition of the terms related to this research field and in particular definition of the term *software visualisation*

itself. Following this, Section 2.3.3 provides a discussion of the role that software visualisation plays within software maintenance and program comprehension. A review of the current state of research in software visualisation is provided in Section 2.3.5, including a brief look at various systems developed. This includes an introduction to the more specific area of three-dimensional software visualisation, which is expanded with a detailed review of current systems utilising 3D graphics in Chapter 3.

2.3.1 The need for software visualisation

It is an acknowledged fact that software maintenance consumes the majority of resources within the software lifecycle [Sommerville92, Standish84]. Another fact is that program comprehension plays the major role in any maintenance activity [Mayrhauser95, Oman90]. Whether the activity is adaptive, perfective, preventative or corrective, all require an understanding of the software system to some extent. The task of gaining this understanding falls under the broad scope of program comprehension.

Problems and deficiencies within the software development process often result in software systems that are badly designed, badly implemented, and possibly worst of all, badly documented. Flaws in design and implementation do cause problems, however, if these flaws or processes are properly documented, then steps can be taken to correct or work around them. By contrast, if the documentation is lacking in any respect then such flaws can go unnoticed and cause severe difficulties during maintenance. Even well designed and implemented code can prove a problem if not correctly documented. Maintainers must then attempt to discover the design intentions behind the implemented system, in the worst case from the source code alone.

Unfortunately, problems with system documentation are widespread. Time constraints imposed upon projects often result in rushed implementations, with documentation receiving a low priority. As a result, documentation is often non-existent, incomplete, inaccurate, or worst of all, misleading. For this reason, program comprehension is often limited to discovering an understanding of a system from the source code and any scant documentation. Various theories of program comprehension put emphasis on the construction of a mental model of the software within the mind of the maintainer (see Section 2.2). Theories on the construction of this mental model hypothesise various techniques employed by maintainers in extracting information from the program source code. Software visualisation attempts to provide tool support for generating this mental model or understanding.

The mental models proposed by various researchers in the field of program comprehension have a commonality in that they all are composed from semantic constructs. These constructs are typically abstractions, at various levels, of program features. The network formed from these constructs constitutes the maintainers understanding and representation of the program. Software visualisation attempts to provide a mapping from the program code to a visual (and other media) representation that matches the maintainer's mental model as closely as possible. Highlighting and automatically creating these semantic constructs would remove a great deal of effort from the maintainer, specifically they

would require less time spent on scanning the source code. The field of software visualisation is thus considered an important area of research within the general field of program comprehension.

Roman and Cox [Roman92] highlight a number of arguments often used in advocating the need for software visualisation. These arguments include: the importance of imagery in human communication; the speed with which humans can track and identify visual patterns; the expressive potential of a visual vocabulary that exploits multiple dimensions; and the inherent power of abstraction within visual representations. The credibility of such arguments is often underlined with reference to the success of graphical representations in areas such as human computer interfaces, scientific data visualisation and information visualisation.

2.3.2 Defining software visualisation

Despite being an established field of research, software visualisation is still not a well-defined term and lacks any concrete definition or meaning. Many researchers have proposed definitions, though the majority of these definitions contradict one another or offer varying levels of detail, scope and abstraction. At present the term 'software visualisation' is generally used to encompass almost all forms of visualisation concerned with representing any aspect of a software system. A number of more specialised areas have evolved under the guise of software visualisation, areas such as *algorithm animation*, *data visualisation* and *code visualisation*. These areas are also subject to varying definitions and scope of field. To compound matters, the actual term *software visualisation* is often used as being synonymous to *program visualisation*, *computation visualisation* and *code visualisation*.

A vast number of definitions and descriptions related to software visualisation are given in the literature, a large selection of these are presented below. This list is intentionally long, to illustrate the variety of definitions and interpretations offered.

- *Program visualisation* uses graphics to illustrate some aspect of the program or its run-time execution, where the program is specified in a conventional, textual manner. [Myers90]

- *Software visualisation* is the use of the crafts of typography, graphic design, animation and cinematography with modern human-computer interaction technology to facilitate both the human understanding and effective use of computer software. [Price93]

- *Program visualisation* deals with graphical presentation, monitoring and exploration of programs expressed in textual form. [Roman92]

- *Program visualisation* is a mapping, or transformation, of a program to a graphical representation. [Roman93]

- *Software visualisation* describes systems that use visual (and other) media to enhance one programmer's understanding of another's work (or his own). [Domingue92]

- *Program visualisation* is a sub-set of the area known as *software visualisation* - the use of graphics and animation to visually describe and illustrate software and its function. [Jerding94]

- *Visualisation* is the process of creating and manipulating a visual image which allows a mental picture of some situation or phenomenon to be formed. These images can either be static or dynamic and need not necessarily be graphical in nature, this allows the possibility of images comprising of symbolic constructs. [Watson95]

- *Software visualisation* is the use of visualisation and animation techniques to help people understand the characteristics and executions of computer programs. [Muthukumarasamy95]

- *Program visualisation* is the representation of source code, and other documents, in a graphical manner. [Kings95]

- *Software visualisation* illustrates computer processes and data in addition to regular programs, while *program visualisation* illustrates data structures, program state and program code. *Computation visualisation* encompasses both software and hardware views. [Stasko92a]

It can be seen from the above definitions and descriptions that the field of software visualisation is diverse, with research interests in many different directions. One commonality evident in the majority of definitions is the emphasis on visual representations. The term software visualisation or program visualisation is generally read as incorporating exclusively graphic representations. This, however, is misleading and overly restrictive. Software visualisation should allow for representations other than purely graphical. It is not unreasonable to foresee visualisation tools which make use of other available human senses, in particular hearing and touch. Price *et al* redefine their use of the term visualisation to include such flexibility, as "the power or process of forming a mental picture or vision of something not actually present to the sight" [Price93].

One of the major differences between various definitions such as those given above, is their specificity. Almost all of the definitions and descriptions are too specific in one aspect or another. Some definitions imply that visualisation must incorporate solely graphics or a particular media, whereas others state the use of specific techniques or systems. Furthermore, many of the definitions omit the *purpose* of software visualisation - what the intended goal is. The definition of the term software visualisation as used throughout this thesis will be:

> *"Enhancing or aiding the understanding of a software system using visual representations and other media."*

This provides a universal definition lacking any specificity for implementation techniques or media used. Also avoided is any mention of how that understanding is gained or how it is aided, thus allowing for a variety of comprehension models to be addressed. Finally, this definition encapsulates the general intention and meaning of the majority of definitions that have already been proposed.

2.3.3 The position of software visualisation

Identifying the position of software visualisation with respect to other fields and more specific sub-fields has been suggested by a number of authors [Price93, Stasko92a, Stasko92b, Myers90]. Price *et al* [Price93] view software visualisation as a field which encompasses the two distinct areas of *program visualisation* (including *visual programming*) and *algorithm animation*. Stasko [Stasko92a] proposes the field of *computation visualisation* which includes both software and hardware views. Fields encompassed by the topic of software visualisation are identified by Price *et al*, resulting in the structure shown in Figure 2.1. This structure does, however, pose some questions.

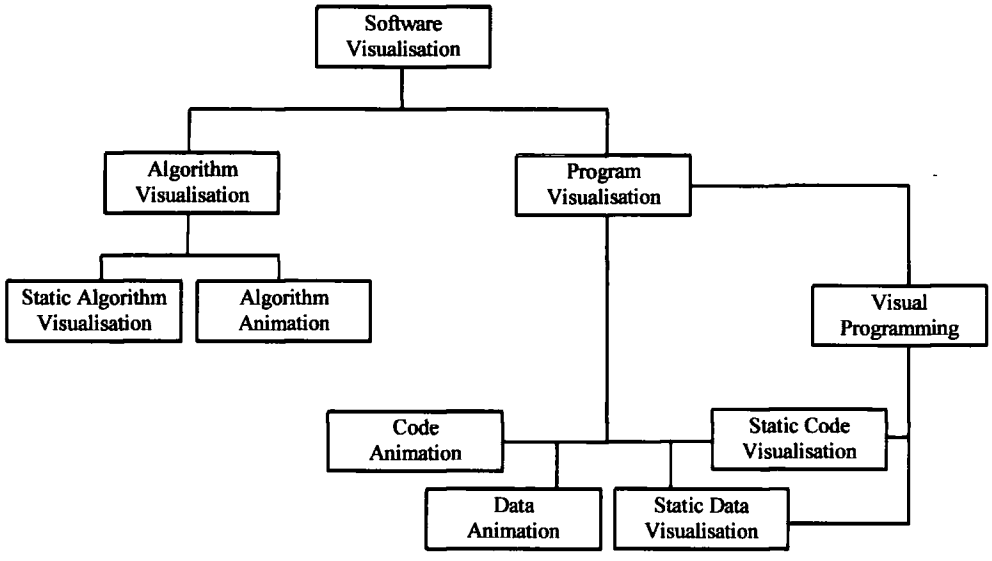


Figure 2.1. Scope and inclusion of software visualisation terms.

Equivalent to Figure 1 in [Price93].

Visual programming, as defined by Roman and Cox, is "*concerned with graphical specification of computer programs*" [Roman92] and does not apply to the current onslaught of so-called visual languages such as Visual C++, Visual Basic, VisualAge and Visual Objects. These systems provide little more than a unified GUI (graphical user interface) to conventional program development tools [Najork95]. Price *et al* identify the term *program visualisation* as including the field of *visual programming*. This notion is in contradiction to other views that consider program visualisation and visual programming as disparate fields [Roman93, Myers90]. Myers even states, "Program Visualisation is an entirely different concept from visual programming" [Myers90]. Conversely, it is not unreasonable to argue that, in order to create a visual programming environment, you must first be able to visualise programs, hence program visualisation. This argument is open to interpretation. One discrepancy, however, appears in the classification by Price *et al* in which the field of algorithm animation is not included as part of program visualisation. It seems unreasonable to make such a distinction in one case and not in the other.

Further, Price *et al* describe program visualisation as being composed of both static and dynamic code and data visualisations, whereas visual programming is concerned only with static visualisations. Again, this point is open to debate. While visual programming is concerned with the creation or visual implementation of programs, this process should not be restricted purely to static visualisations. One simple argument is that debugging and tracing of code is considered an integral part of programming, and in a visual sense the only way in which the execution or tracing can be presented is through animated visualisations. Another argument is that we cannot constrain the visual programming paradigm to purely static or even graphic visualisations. A visual programming-by-example paradigm could be imagined in which the programmer specifies operations by manipulating the graphical interface in a dynamic fashion; by providing an example. For example, copying or moving data within memory, this being achieved by dragging or moving appropriate areas within the visualisation.

The arguments posed above offer constructive criticism to the classification of definitions within software visualisation. The validity of these arguments does however hinge upon the reader's interpretation of the terms and their scope. Figure 2.1 above illustrates the scope of these terms as described by Price *et al*. Similarly, Figure 2.2 below shows a revised classification of these terms which incorporates the arguments mentioned above. In addition to the field of software visualisation, the term *computation visualisation* is introduced by Stasko [Stasko92a] as including both software and hardware views. This is incorporated in Figure 2.2.

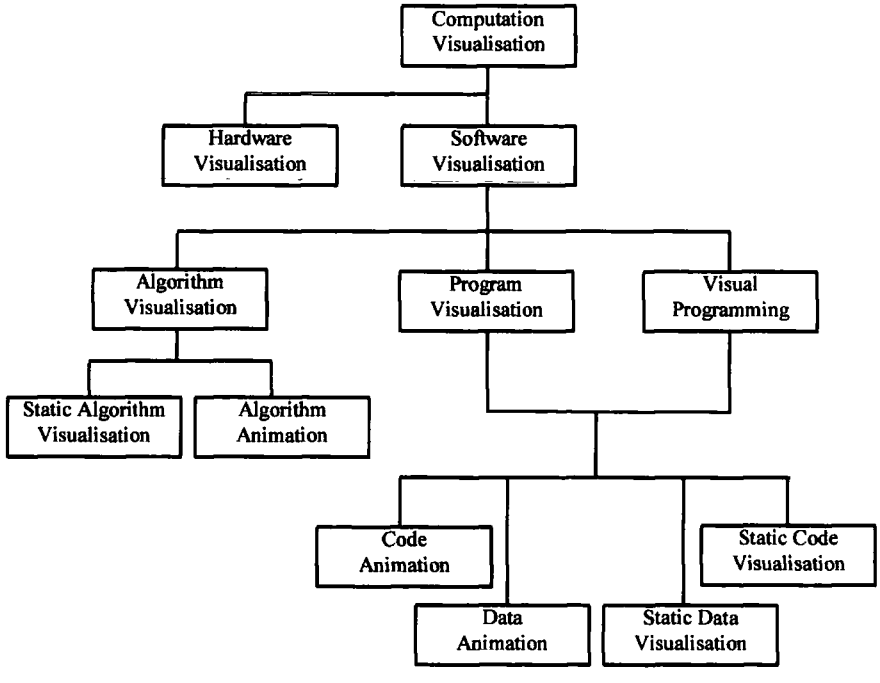


Figure 2.2. Revised scope and inclusion of software visualisation terms.

A brief description of each of these fields is now given.

- **Computation visualisation**

  Computation visualisation is the use of computer graphics to explain, illustrate and show how computer hardware and software function. [Stasko92a]

- **Software visualisation**

  Software visualisation describes systems that use visual (and other) media to enhance or aid comprehension of software systems.

- **Program visualisation**

  Program visualisation is the representation of source code and other documents to enhance understanding of an existing software system.

- **Visual programming**

  Visual programming is the use of graphical representations to specify and implement computer software.

- **Algorithm visualisation (Static and Dynamic)**

  Algorithm visualisation concentrates on the representation and animation of the abstract operation of software algorithms.

- **Static and dynamic code visualisation**

  Concentrates on illustrating the actual program code, either by enhancing its textual appearance [Hill93] or by representing it in a more abstract form.

- **Static and dynamic data visualisation**

  Concentrates on visualising the data defined and manipulated within the software.

2.3.4 Taxonomies of software visualisation

Several taxonomies have been applied to software visualisation systems. These taxonomies vary to the same extent as the definitions in that several possible classification systems are presented, but no one method can be identified as being most appropriate. There are two taxonomies predominant in the field, an early taxonomy by Myers [Myers90] and a more recent taxonomy by Price *et al* [Price93]. Myers' taxonomy classifies systems along two main axes: the data they present, such as code, data or algorithm visualisations; and whether the visualisations are dynamic or static. While this taxonomy covers some of the important characteristics of a software visualisation system, it does not provide any scope for evaluating the effectiveness, usability, applicability or quality of systems. Price *et al* address this to a greater extent in their taxonomy.

Price *et al* categorise systems in a more systematic manner and attempt to structure the taxonomy to allow flexibility for future expansion and revision. Their taxonomy is based upon a hierarchical structure of classifications, which allows simple addition of new categories. The top level of this

hierarchy is based on a very general model of software containing six categories: *scope*, *content*, *form*, *method*, *interaction* and *effectiveness*.

Other taxonomies exist, again using a different set of classifications. One such taxonomy is by Stasko and Patterson [Stasko92b] and introduces scaled dimensions into a four-category taxonomy. These categories cover *aspect*, *abstractness*, *animation* and *automation*. Another such taxonomy by Roman and Cox [Roman92] initially identified systems using the categories of *scope*, *abstraction*, *specification method* and *technique*. This initial taxonomy was refined further [Roman93] with the category '*technique*' being replaced with the two categories of *interface* and *presentation*. Roman and Cox view software visualisation as a mapping from programs to graphical representations, with the categories of the taxonomy representing aspects of this mapping [Roman92]. The *scope* covers what aspects of the program are visualised. *Abstraction* and *specification method* cover the actual visualisation process: what information is conveyed and how it is constructed. Finally, *interface* and *presentation* (*technique* in earlier work) cover the actual graphical representation: how it is used to convey the information.

This section serves to highlight the lack of any consensus between researchers as to what aspects characterise a software visualisation system. This reinforces the view that software visualisation, and particularly 3D software visualisation, remains a relatively new area of research.

2.3.5 The current state of research

Current research into software visualisation is diverse, with activity in all areas of the subject. Initially the main research thrust was in developing algorithm visualisation frameworks and systems. These systems were used predominantly for teaching algorithm design and analysis and were not typically used as general-purpose tools. Another favourite topic within software visualisation is its application to program debugging and execution tracing, particularly within imperative languages. More recently, an increase in research has been directed towards structural or macroscopic visualisations of code. These visualisations concentrate on a large-scale view of the code and typical examples are function call graphs, control flow graphs and file dependencies. In addition, there is a very active research interest in the visualisation of parallel programs, though generally concentrating on their execution behaviour [PVMTrace99, Koike99, Miller93, Kraemer93].

Algorithm visualisation research concentrates on producing meaningful and insightful visualisations and animations to aid understanding of algorithm operation. Generally these visualisations are not produced automatically and require an animator to design a suitable representation for each algorithm. The majority of algorithm animation frameworks created rely on the animator to instrument the algorithm implementation at critical points with additional calls to the animation environment or to a log file generator. This information is then used either during execution or post-mortem, respectively, to drive the animation. Many systems have been developed for algorithm animation, a large number of which are described in various taxonomies [Price93, Myers90, Roman93]. Such systems include BALSA; Balsa-II; Zeus; TANGO; Polka-3D; and ANIM.

Visual debugging is not generally accepted as a separate topic of program visualisation though there are a large number of systems that use visualisation as a primary aid to debugging. Many of these systems are confined to tracing the execution of imperative languages, however, various other languages and paradigms are also covered. The majority of these systems operate automatically; i.e. they do not need manual intervention to obtain the program information. This is generally obtained using one of two methods. One method provides an interpreter base in which the target program is 'executed' thus allowing the visualiser access to all information regarding the program and its execution. Alternatively the visualiser will automatically instrument the target source code with calls to the visualiser, or with statements which output information to a log file for post-mortem analysis. Again, these systems are described and classified within the various taxonomies. Such systems include Pascal Genie; UWPI; TPM; Pavane; LogoMedia; CentreLine ObjectCentre (Saber-C++); ZStep 94 [Lieberman95, Fry95]; and PVMTrace.

Structural visualisations concentrate mainly on large scale or macroscopic views of the source code. These views typically consist of visualising dependencies between the various components that make up a software system. Components such as variables, classes, functions, methods, files and modules are displayed to produce the stereotypical views of call-graphs, control-flow graphs, module dependencies and data dependencies. The key factor of these views being their inherent graph structure. These graphs are often very large and difficult to display effectively. Such systems are not covered in current taxonomies, though they could be easily incorporated by the various taxonomic criteria. Examples of these systems include SMARTsystem; McCabe; Logiscope [Meekel88]; SNiFF; Rigi [Storey95]; Narcissus [Hendley95b]; GraphLog [Consens92]; SemNet [Fairchild88]; and GraphVisualiser3D [GV3D99].

The majority of software visualisation systems to date have concentrated on producing two dimensional representations and animations of various aspects of a software system. In addition, a growing area of research is investigating the application of 3D graphics and virtual environment technology to software visualisation. Research into 3D software visualisation attempts to address the problem in terms of what data to represent, how to represent it, and the user's ability to navigate this data. The former aspects stem from a growing need for more expressive powers to represent the inherently multidimensional data, facts and relationships which constitute a software system. The latter plays upon the naturally developed capabilities of humans for navigating and interacting within three-dimensional environments.

Software systems, and in particular the large, highly inter-related information bases that they constitute, are inherently multidimensional and thus difficult to visualise [Brooks87]. Software visualisation systems typically isolate one aspect or view of this information for display, effectively hiding all remaining information from view. This isolated view generally consists of a large and complex directed graph which is displayed on a two dimensional canvas using one of the many layout algorithms developed within the field of graph layout. Regardless of the quality of the layout algorithm or viewing technique used (such as fisheye), the user is typically presented with an incomprehensible birds-nest graph or must reduce the amount of information displayed thus removing potentially important

information from view [Fairchild88]. While the use of 3D or virtual environment technology cannot immediately solve this problem, it does offer a number of advantages. Advantages such as: greater working volume for information presentation; focus, context and point of interest are managed dynamically by the user's viewpoint on the information; and greater flexibility with representation, organisation and presentation of the information. These arguments do not in themselves justify 3D as superior to 2D representations, however they do highlight possibilities for advances in software visualisation using 3D graphics.

2.3.6 Summary and conclusions

This section has described the current state of software visualisation research and provided a brief introduction to the rapidly developing field of 3D software visualisation (discussed further in Chapter 3). Particular attention has been paid to the definition of software visualisation and related topics; its application to software maintenance; and to the current state and direction of research effort. There are a number of conclusions that can be made:

- **The field of software visualisation is still not clearly defined.**

As yet, there is still no common consensus on the definition of software visualisation and other related terms. Worse still, many of the terms are used within different context and are often interchanged, such as software visualisation and program visualisation. A firm definition of the terms within the field of software visualisation is needed to enable meaningful and unambiguous discussion. A definition of the various terms as interpreted within this thesis has been provided.

- **The scope and range of current software visualisation systems is too small.**

At present, software visualisation systems concentrate only on small, 'toy' programs, with few exceptions. For software visualisation to become a viable tool, widely accepted within the software maintenance community, then it must address the needs of industrial applications. The majority of techniques available in the software visualisation arsenal tend to fail when applied to large, industrial strength applications. It may not be possible to devise generic techniques that would apply to both small and large programs. In such a case, more research is needed to investigate the development and in particular, the integration of separate techniques. There is a need for industrial tools that are capable of being applied to real-world problems.

- **There is a current lack of, and need for, empirical evaluations of software visualisation systems.**

A large number of software visualisation systems have already been produced, the majority of which are research prototypes. At present there is no empirical method for evaluating the effectiveness or worth of particular systems. Similarly, other than the existing taxonomies for classifying systems, there is no empirical method for comparison between systems. There is little point devising new

visualisations without the ability to determine if they provide any improvement, or even without knowledge of what makes a good or bad visualisation.

- **An improved level of automation and integration is required.**

Many of the software visualisation systems developed rely heavily on user interaction and aid in generating the visualisations. This aid can be provided at various levels, from instrumenting or pre-parsing source code to defining and creating the actual visualisation itself. There is a definite need for a greater level of automation to increase user acceptance of such tools. Similarly, integration of visualisation tools within the software development environment is an important goal. Only when these two areas have been addressed more thoroughly will we begin to see visualisations become more common place. The effort required in visualising software (source code) should not greatly exceed that required to compile and execute it.

- **Research must address the need of users of the visualisations.**

Software visualisation research needs to take a more detailed look at what the user actually wants to see, and how they would like to see it. At present, many systems produce wonderful, interesting pictures or animations that, while looking very impressive, provide little information of value to the user. Alternatively they present information which could often be obtained through more conventional approaches, imparting a lower overhead cost in computing resources or time. Similarly, the information they present may be clearer or more immediately identified when obtained from other sources. Research needs to concentrate on how the technology can aid the user, rather than creating merely 'pretty pictures'.

Regardless of the above comments, it seems safe to assume that research and interest into software visualisation will continue to grow. As hardware technology improves and prices continue to fall, the overheads of creating visualisations will become less of a concern. Software visualisation has great potential to provide useful, time-saving support to the software maintenance (and possibly development) process. Whether or not this potential is realised remains uncertain, though the deciding factor will undoubtedly be its acceptance by industry.

2.4 Conclusions

This chapter has presented a literature review of two distinct research areas, program comprehension and software visualisation. The findings of this chapter can be summarised as follows:

- There are many theories and models for program comprehension though little consensus as to which theory is most representative of the program comprehension process.

- Most program comprehension theories subscribe to the notion of a cognitive model which is composed of a knowledge base, mental model and assimilation process.

- Several researchers suggest an integrated program comprehension process in which the maintainer selects a particular strategy on an as-needed basis. The research presented within this thesis is based upon the notion of this integrated approach to program comprehension.

- Software visualisation is a large and well-established research field, though the more specific area of 3D software visualisation is relatively new.

- There is much variety in the terminology used and little consensus as to the definition of software visualisation. A definition of software visualisation is provided, as viewed within this thesis.

- There are many unresolved issues within software visualisation, providing a broad scope for future research.

- Software visualisation is currently limited to a small set of views upon a software system. There is a need to explore new techniques and new views for visualising software systems.

The following chapter focuses in detail on various aspects of 3D visualisation. This includes an introduction to virtual reality and a comprehensive review of 3D information visualisation research, concluding with a brief overview of 3D software visualisation research.

Chapter 3.

3D visualisation

3.1 Introduction

This chapter presents a review of 3D visualisation systems and provides an introduction and explanation of the various aspects involved in a virtual reality system. The major part of this chapter describes 3D information visualisation, both the techniques developed in this area and the large variety of prototype systems created. Information visualisation is viewed here as encompassing 3D software visualisation, the main topic of this thesis. Information visualisation attempts to provide a means of identifying trends, generalities and anomalies within large information stores. The goal is also to provide a unique and efficient means of navigating, querying and working *within* these large information stores. Similarly with software visualisation, a software system merely represents a specific form of information collection. Software visualisation attempts to convey information about a software system with the goal of imparting some knowledge of the construction, structure and workings of that system to a software maintainer. The parallels between information visualisation and software visualisation are so great that both types of system are described side by side in Section 3.3.2.

This chapter is concerned with 3D visualisation, both information visualisation and software visualisation, and to a lesser extent the enabling technology, namely virtual reality. Virtual reality is a term used widely to describe almost any form of 3D graphical system. While the use of the term VR may often be inappropriate, it is sufficiently broad to encompass many of the systems described here and is directly relevant to the focus of this thesis. Section 3.2 provides an explanation of the term virtual reality and defines what software and equipment constitute a virtual reality system. Factors important to a VR system are also described, along with a brief look at the large variety of different types of VR system.

Section 3.3 introduces the field of 3D information visualisation and describes a variety of visualisation techniques and research systems that have been developed to aid human comprehension of large information systems. Also incorporated within this review of systems are a number of 3D software visualisation prototypes. It is believed that software visualisation is essentially a more specific form of information visualisation, the systems described here using many of the same techniques and with similar goals to that of pure information visualisation systems. Finally, Section 3.4 describes the more specific area of 3D software visualisation with reference to the systems described in Section 3.3.2.

The goal of this chapter is to provide an introduction to virtual reality and 3D visualisation. Section 3.2 aims to provide an unambiguous view of virtual reality and to clarify any preconceptions of the term or the technology. Section 3.3 highlights a closely related field which has received considerable research attention, and which promises to provide usable techniques that may also be applicable to the subject of this thesis, 3D software visualisation. Overall, this chapter provides a broad look at a variety of visualisation techniques and prototypes, highlighting the possibilities afforded by 3D visualisation and attempting to disprove any preconception that the use of 3D graphics is merely a gimmick.

3.2 Virtual environments and VR systems

The term *Virtual Reality* (VR) has been used (or misused) widely to refer to almost any form of computer graphics and interaction. There are numerous definitions of the term virtual reality and contention as to what a VR system actually is. Many people view VR as an attempt at modelling the real world as believably as possible, others see it merely as an advanced form of human-computer interface. Certain views suggest that a true VR system must include paraphernalia such as head-mounted displays (HMDs), tactile feedback gloves, and full body suits. This suggests that the interface to the world is what characterises a VR system, whereas other views deem the actual 3D environment, level of interaction and the degree of presence to be of more importance. Regardless of these views, everyone generally accepts VR for what it is; a system for providing an interactive exploration of a three dimensional virtual environment. A large number of professionals in the field have opted for the term *Virtual Environment* as it carries less of the hype and expectations which VR has been associated with.

In the early days of VR research the technology was rare and expensive meaning that only the dedicated researchers or companies with a realistic practical application of the technology were developing systems. Things have improved greatly over recent years with computer hardware becoming increasingly more powerful while steadily becoming more affordable. Systems capable of running a VR application are now commonly available within people's homes and in the everyday workplace. Increasingly, companies are investigating what VR has to offer them and how they can put the technology into practice. Research into VR systems and applications of VR is now widespread.

3.2.1 Virtual reality systems

Computer graphics is now a well-established field and is used in countless applications and situations. The research presented in this thesis concentrates on 3D computer graphics, a more specialised yet still extremely widespread area. Computer aided design (CAD) systems originally led the way with 3D graphics. These systems allowed 3D models of real-world objects to be constructed as geometric descriptions of their structure. By supplying other information such as the position, size, colour and orientation of these objects it is possible to render realistic images of a product before any real-world prototype has been built. The implications of this, from an industrial point of view, are incredible savings during product development as ideas and designs can be constructed and evaluated in a fraction

of the time and cost of building an actual prototype. CAD is now a vital tool within a large number of research and engineering applications.

Virtual environments are based on similar principals to CAD systems. A database is used to store the geometric representation of the objects within the VE. A graphics engine constructs an image of the scene, which can be viewed from any angle or position, and displays the image in a 2D form as if looking through a camera. The term used for creating this image is *rendering*. Rendering a 3D scene is a complex and time consuming process. In order to produce a moderately detailed scene it was often necessary to wait for several minutes or even hours for a single image. Hardware and software have progressed a great deal and it is now possible to produce realistic images in a fraction of a second. This is what made VR possible, being able to generate several images per second means that a user can control the viewpoint or camera and explore the virtual world in real-time.

There will always be a trade-off between realism and interactivity. The more realistic a scene must appear then the longer it will take to render and the slower the virtual environment will update. This again produces two different opinions on what a virtual reality system's goals should be. Some people believe that a virtual reality system must look real, requiring the most detailed images possible. Other people believe that it is the fluidity and responsiveness of the 3D environment that provides the key to a VR system. Both these approaches are valid, but in terms of computer graphics they are mutually exclusive. Incredibly detailed images will make the VE appear more realistic, but movement through the environment will be slow and cumbersome, detracting from the experience. On the other hand, less detailed scenes will appear false and artificial, but movement through the environment will be smooth and responsive giving a heightened sense of immersion and presence.

3.2.2 Virtual reality software

There are a vast number of different software packages allowing users to either experience virtual worlds, or even to create and edit them. The majority of professional VR packages offer the same basic functionality, allowing a world to be created from any number of 3D objects which can be arbitrarily defined using either graphic primitives or specific face sets (collections of 2D facets which comprise a 3D object). These packages also offer total freedom in viewing the virtual world from any conceivable position and orientation. Different systems merely offer additional features, perform certain operations better, give improved performance or image quality, etc. A more interesting development of 3D graphics engines and immersive environments has occurred in the computer games industry. Regardless of the stigma of computer games within the serious research community, it is undeniable that they offer some of the most immersive, usable and engrossing virtual environments - how would they be so successful and popular otherwise?

The majority of VR systems produced are commercial applications designed to allow rapid creation and exploration of a virtual world. These systems are typically extremely flexible and allow a great deal of control and detail to be designed into a virtual world. Such systems are designed to cater for the widest

possible range of uses and are thus extremely generic in nature. On the opposite scale of development within the range of virtual reality systems are computer games.

Computer games have developed greatly over the past decade with a strong focus towards incorporating 3D graphics in almost every modern title. Computer games are often dismissed by the VR community due to their focus on entertainment and lack of any apparent research value. This has changed recently with the advent of more powerful, flexible and imaginative game engines. Such engines are beginning to encroach on areas previously dominated by dedicated VR systems. Research has already explored the possibilities afforded by a game engine for information visualisation [Knight98].

3.2.3 Important factors in VR systems

There are many factors attributing to a realistic and believable virtual environment, three of the more important factors are described briefly here. These three factors are visual realism, latency and presence. Other factors important to a VR system include image resolution, frame rate, interaction, believability and immersion.

I. Visual realism

The level of realism in a scene aids considerably in making a believable environment. Ray tracer and professional animation systems produce incredibly realistic images, such as those used in special effects for movie production. Some of the best applications of this technology result in the viewer not noticing any transition or discrepancies between real footage and computer generated effects. Providing this level of detail and sophistication is extremely complex and requires a great deal of rendering time. More and more advanced features are slowly appearing in virtual environments though it will still be a long time before they reach the same quality which current computer animation can provide.

Visual realism can be increased or affected in numerous ways. One of the most obvious and common methods for increasing visual realism is to increase the polygon count within a virtual environment. Objects within the VE are created from a discrete number of polygons. The majority of VR systems use polygonal facets to create all objects within the VE, even curved surfaces such as a sphere or cylinder. By increasing the number of polygons used to represent an object, the level of detail is increased and objects begin to appear less geometrical. This has a similar effect to increasing the resolution of a 2D graphical image, detail is increased and the granularity of the image is reduced.

Another popular and extremely effective technique is to map detailed and realistic textures onto the facets of an object. The greater the number, resolution and variety of textures used then the more realistic an object will appear. Obviously this implies correct application and use of texture. Poor and thoughtless use of texture can also be detrimental to visual realism. Finally, there are numerous algorithms and techniques that address areas such as shading, lighting and shadows. Applying such algorithms to a virtual world may increase visual realism greatly. Unfortunately, while there are many techniques that can be applied to increase visual realism, almost all of these come at the cost of

increased rendering time. Increasing visual realism will invariably increase rendering time and have adverse effects on other aspects of the virtual environment. The only solution here is to increase available computing power.

II. Latency

Latency is probably one of the most important aspects of a virtual reality system which must be addressed to make the environment not only more realistic, but simply tolerable. Latency, frequently referred to as *lag*, is the delay induced by the various components of a VR system between a user's inputs and the corresponding response from the system in the form of a change in the display. As latency increases, a user's senses become increasingly confused as their actions become more and more delayed. Chronic cases can even result in simulator sickness, a recognised medical problem associated with virtual environments [Vince95]. Latency must be kept to a minimum in order to create a usable VR system.

Although latency is undeniably a major problem for virtual environments, humans are often very adept at coping with its affects and adapting their use of the environment to compensate. For this reason, small amounts of latency are acceptable. There are two main types of latency: fixed and variable. Fixed latency refers to a consistent delay between user action and virtual world reaction. This delay is often due to physical delays in the virtual reality system and will remain reasonably steady throughout. This form of latency can be most easily compensated for by the user, even up to relatively high levels. Variable latency is more problematic and refers to unpredictable bursts of lag, varying amounts of latency and even sudden pauses in the virtual reality system. Variable latency most often occurs in networked virtual environments due to the unpredictable nature of the network performance. Another cause of variable latency is often scene complexity. When moving the viewpoint from a scene of relatively low complexity to a high complexity scene, there is a dramatically increased load on the graphics system. This load will result in variations in frame rate possibly leading to perceived latency (or actual latency in extreme cases). However, the increased load on the graphics system will reduce the amount of computing power available for other important systems such as input sampling or the simulation engine, this can result in actual and large increases in latency.

Due to the physical limits of computing equipment VR systems will always experience some degree of latency, minimising the amount and type of latency is the key to a successful virtual environment. Keeping fixed latency as low as practical and reducing the effects of variable latency is an important goal and a critical design issue for VR systems and worlds.

III. Presence

The overall goal of optimising and balancing each of the properties of a VR system is to provide the user with a heightened sense of presence within the virtual world. The terms presence, immersion and believability are often mentioned when describing virtual environments and the sensations associated with their use. Presence is a sense of having a position, orientation and representation within a virtual

world. It is the sense of being involved in a world and belonging as part of that world. Presence refers to the user's virtual representation and not to the user themselves. Immersion, on the other hand, is a feeling that the user himself or herself is immersed bodily within a virtual world. Finally believability is a sense that what the user sees constitutes a believable environment. To some extent it is the ability to believe that the virtual environment is representative of a real environment. The dependencies and relationships between these terms are often argued or confused by researchers, their definitions and interpretations being very subjective.

It is sometimes argued that presence and immersion are in fact the same, or that to create a sense of presence or immersion then the environment must be believable. Believability is also argued to be dependant on visual realism and also to the extent at which a virtual world mimics the real world. Fantastic settings used within a virtual world are often deemed to defeat believability. In this thesis it is thought that the terms presence, believability and immersion are each discrete measures of the experience a user has when interacting with a virtual environment. While there is undeniably some relationship and dependency between these terms, it is believed there are no prerequisites that one implies the other or that one requires another. These relationships are also very subjective, with each user having a very different experience of their interaction with a virtual world. Some people find it easier to suspend belief than others do. Similarly some users are capable of mentally sustaining a sense of presence within two locations simultaneously – the virtual world and real life. Those that cannot, may limit their feeling of presence to the real world or to the virtual world, giving two extremes of immersion.

It is the opinion of this research that the level of presence is by far the most important of these three attributes. Immersion can be improved greatly by the use of novel interface technologies. Believability is so highly subjective that it becomes excessively hard to cater for in the design of a virtual world. The level of presence, however, is the basic necessity for navigating and interacting with a virtual environment. A feeling of presence is what puts the user *into* the virtual world. Presence can be accentuated in various ways though the most important provision is that of a body or avatar for the user. In the same way that we relate to our real world environment, which is fashioned about our stature and nature, the user will relate to his or her virtual environment with similar expectations.

3.2.4 Types of VR systems

Virtual reality systems can be classified into three main groups depending on their primary medium. These are graphical, textual and non-visual [Dieberger94]. This research is concerned only with graphical virtual environments and for the purpose of this research all reference to the term virtual reality or VR in subsequent chapters will imply a graphical virtual environment.

Graphical virtual environments, as the name implies, incorporate some form of visual display to produce a graphical image of the user's view from within the virtual world. These systems require little imagination on the part of the user because a concrete and recognisable image of the virtual world is

displayed. The only imagination necessary is to be able to suspend a feeling of disbelief and to put aside the knowledge that the environment is false. This involves recognising objects for what they represent and not merely how they appear. This is necessary because most objects in a virtual environment will only be roughly similar to real world objects, and often will appear far from life-like. This is, however, only a limitation of current technology.

There are a number of different classifications of graphical VR system, based mainly on the interface methods used. A number of these are described in detail here. The purpose of highlighting these different forms of VR system is to illustrate that VR is not dependent on any one form of technology or any particular input and output devices. Virtual reality is a hardware independent concept, contrary to the popular public view which considers the term virtual reality to be synonymous with head-mounted display goggles and novel interaction techniques.

I. Window on World (WoW) or Desktop VR

One of the most common and accessible forms of virtual reality is Desktop VR or Window on World (WoW) systems. These systems do not rely on any specialist input or output devices in order to use them. Typically a standard computer mouse and monitor is sufficient. The user interacts with the virtual world by use of a controller such as a mouse, keyboard, joystick or other specialised device. Visual feedback is provided by the computer monitor which acts as a window looking into the virtual world. Computer games are a prime example of a common and successful application of desktop VR.

A quote by Ivan Sutherland in his 1965 research paper highlights the goals of this approach. "*One must look at a display screen as a window through which one beholds a virtual world. The challenge to computer graphics is to make the picture in the window look real, sound real and the objects act real.*" [Sutherland65]

II. Immersive VR

The goal of most virtual reality systems is to completely immerse the user within a synthetic environment, to make them feel a part of that environment. In order to exploit this to maximum effect the user must be effectively cut off from the real world and instead have a presence and viewpoint within the virtual world. Such a VR system is referred to as immersive VR. Current technology usually involves a head-mounted display (HMD) which provides visual and auditory feedback. It is possible to have varying degrees of immersion within a virtual world, this being achieved by concentrating on simulating only certain senses or actions. For example a HMD may provide only visual stimuli, leaving the user audibly aware of their real-world surroundings.

A 'cave' environment is another example of an immersive system, but one that does not use a HMD. A cave environment refers to a room which uses multiple, large projectors to display the appropriate viewpoints on each wall of the room. The user, or users, enter this room and navigate the virtual world using various control systems provided. They are able to look freely around the virtual world by simply

looking in the appropriate direction within the room. One important difference between using a HMD and using a cave system is that the latter can be a directly shared experience. Several users can be present within the cave room and interact directly with each other, though having only a single representation within the virtual environment. If using a HMD system each user would only be aware of other users via a graphical representation of them, such as an avatar. This has obvious implications for group working and communication within virtual environments.

III. Augmented reality

Augmented or mixed reality systems provide a middle ground between a non-immersive and fully immersive VR system. AR systems overlay computer generated information over the user's view of the real world, without completely occluding it. This allows the user to operate within the real world while receiving computer generated information, often relevant to their current task, location or activity within the real world.

The most common example of an AR system is the head-up display (HUD) used widely in modern military aircraft. This superimposes flight data such as altitude, air speed, attitude or targeting information upon the pilot's field of view. There are many possible applications of AR systems with the potential to be extremely useful. In addition, such systems will probably be more acceptable and desirable than fully immersive or desktop VR systems, due to their integration with the real world. AR has a great potential in the rapidly developing strive for creating a portable office.

IV. Fish tank VR

This phrase is used to describe a hybrid system incorporating a standard desktop VR system with a stereoscopic viewing and head tracking mechanism. The system uses LCD shutter glasses to provide stereoscopic images and a head tracker that monitors the user's point of view on the computer screen. As the user moves his or her head, the screen display updates to show the new perspective. This allows the user to look either side of objects and to move their head in order to gain additional depth cues. This system provides a far superior viewing experience than normal desktop VR. The stereoscopic images provide heightened depth awareness and an improved three-dimensional view. More importantly providing head tracking allows the effects of motion parallax as the user moves his or her head, greatly increasing comprehension of three dimensional scenes [Ware94].

3.2.5 Summary

This section has provided an introduction to the field of virtual reality and detailed some of the important aspects of a VR system. The goal of this section was to clarify the meaning of the term *virtual reality* and to dispel any preconceptions of the technology required to create a virtual environment.

To summarise, virtual reality is not dependent on any particular technology and may even be independent of computer graphics. Textual and non-visual environments may also be described as providing a virtual reality, yet these are far from the public conception of a virtual reality system, which typically includes novel interface technology and realistic imagery. The research work presented in this thesis concentrates on the use of graphical virtual environments in a typical desktop configuration. The goal is not to investigate the technology of virtual reality, but to see how the concept of virtual reality can be used to aid software visualisation.

3.3 Information visualisation

This section describes a variety of 3D information visualisation techniques and research systems that have been developed to aid the human comprehension of large information systems. This section is split into two distinct parts. The first part (3.3.1) describes the techniques currently available for displaying various forms of information using 3D graphics. The second part (3.3.2) goes on to describe a number of prototype or research visualisation systems and provide a brief summary of the abilities of each. Finally, the section draws to a close with some conclusions on the systems and techniques described and an explanation of the possible application of such technology to software visualisation. Throughout this section comment will be made as to the suitability or potential for any techniques or aspects of a visualisation to be applied to 3D software visualisation.

Also mentioned below are a number of 3D software visualisation systems. It is the author's opinion that software visualisation can be considered a more specialised area within the general scope of information visualisation. Software systems are essentially information artefacts, they consist entirely of information objects with relationships and dependencies between these objects. The goal of software visualisation is to impart some knowledge of the structure, operation and meaning of this information artefact. These are goals almost identical to that of information visualisation. It is therefore sensible to include such systems alongside more general information visualisation systems. Section 3.4 provides a general overview of the software visualisation systems described here.

3.3.1 Visualisation techniques

This section describes a variety of information visualisation techniques that are used in many 3D information visualisation systems. These techniques range from the familiar data presentation of surface plots and 3D bar charts through to the creation of abstract data spaces and the behaviour of objects within them.

The techniques described here can be classified as belonging roughly to one of three groups: mappings; presentation techniques; and dynamic visualisation techniques. Each of the techniques listed below is described within this chapter.

Surface plots, cityscapes, Benediktine space and spatial arrangement can all be classified as *mappings* from the data domain to the visualisation space. These techniques all use some aspect, property or value of the data items to produce a mapping to objects within the visualisation.

Perspective walls, cone trees, cam trees and rooms may be classed as information *presentation* techniques. These visualisations concentrate on the appearance, accessibility and usability of the data and aim to provide a user friendly and intuitive interface.

Finally, fish-eye views, emotional icons and self organising graphs may all be described as *dynamic* information visualisation techniques. These techniques endow the visualisations with behaviour and dynamic properties, allowing the visualisations to respond automatically to changes in the data or to the actions of the user.

This classification system only serves to provide some gross collation of the various techniques described. There is some degree of overlap between these categories and the classification of a technique is somewhat subjective.

The following sections describe in detail each of the techniques mentioned above.

I. Surface plots

One of the most familiar extensions from standard 2D graphs is the 3D surface plot. Surface plots are constructed by plotting data triples onto the three co-ordinate axes X, Y and Z. Typically the data will consist of two standard sets which have a regular structure, e.g. days of the week and time of day, and one actual data value for example wind strength. The two regular sets are normally plotted on the horizontal axes X and Z with the variable data being plotted as height in the Y axis. The set of points thus formed are netted into a mesh or surface which is often colour coded to indicate height variations. The resulting visualisation resembles a landscape that can be easily interpreted to identify features such as patterns or irregularities.

An example of a 3D surface plot is shown in Figure 3.1 below. The data set describes the number of users connecting to a particular server over the course of 24 hours, split into 2-hour time bands. An additional dimension is added by describing the same data over seven days. This data is then plotted onto the 3D surface plot shown in Figure 3.1. Immediately it allows the viewer to spot any particular trends or abnormalities in the data, requiring considerably less mental effort than interpreting the data values in the table. Trends such as the small peak in lunchtime use midweek, the much higher server use on weekday evenings, and the steadily higher server use on weekends become readily apparent, even on this very small data set.

	12am 2am	2am 4am	4am 6am	6am 8am	8am 10am	10am 12pm	12pm 2pm	2pm 4pm	4pm 6pm	6pm 8pm	8pm 10pm	10pm 12am
Monday	20	13	0	0	10	25	50	10	12	243	302	289
Tuesday	60	2	0	1	7	20	49	15	15	230	271	266
Wednesday	63	3	1	0	11	27	57	11	11	233	280	290
Thursday	60	1	0	0	8	18	44	13	25	280	330	300
Friday	55	2	0	0	12	30	69	23	30	293	297	275
Saturday	88	20	5	2	4	40	120	143	220	413	449	353
Sunday	103	25	12	3	3	32	104	154	254	402	411	301

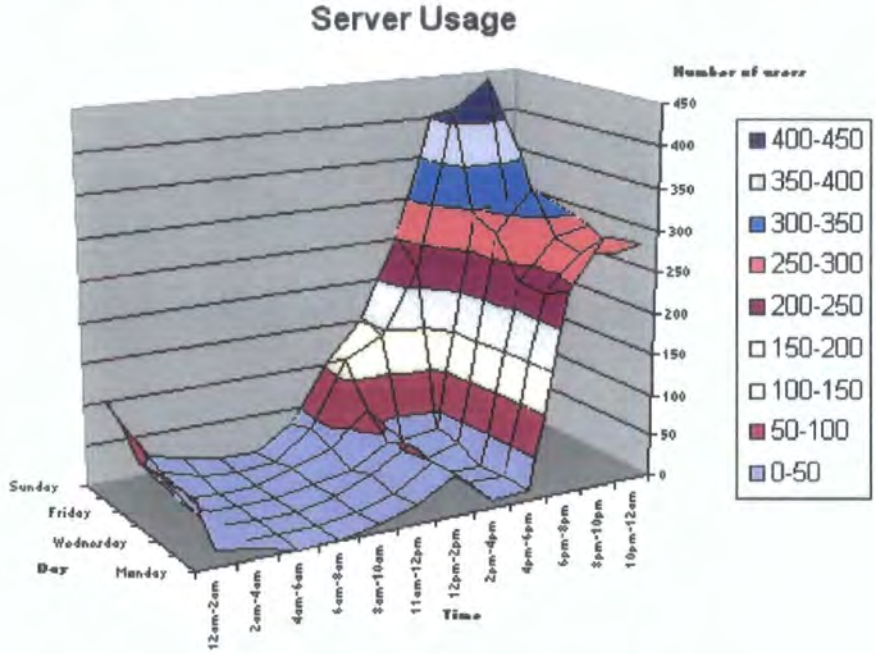


Figure 3.1. 3D Surface plot with corresponding data.

Surface plots are specifically designed for the visualisation of quantitative, typically numerical information. Software systems contain a substantial amount of such information, and in particular the analysis of a software system will often result in the generation of a large amount and variety of metric information. Surface plots provide an efficient and understandable technique for visualising such information.

II. Cityscapes

The term Cityscape is used within this thesis to describe two different visualisation techniques, though both sharing a common theme. The actual term is used in both cases to describe a visualisation with an appearance resembling the skyline, shape and form of a real-world city. The characteristics being regular sized towers of varying height, placed within close proximity of one another and often in a very regular pattern.

The term 'Cityscape' is used here to describe what is effectively an extension to 3D bar charts and a variation on surface plots. Cityscapes are created in a similar fashion to the surface plots by mapping scalar data values onto the height of 3D vertical bars or blocks, the blocks being placed on a uniform

2D horizontal plane. The resulting visualisation can be thought of as a more granular representation of the surface plot. The cityscape demonstration [Walker93] developed at British Telecommunications research labs includes additional features to provide further information on the data set and allow simple comparison of results. One feature projects the minimum, average and maximum values for each row and column in the 2D plane (i.e. X and Z axes) onto the end walls of the cityscape. Another feature allows the addition of a variable transparency 'sheet' to be placed at a specified height in the cityscape. The transparent sheet highlights blocks or data values that exceed the desired height, allowing instant identification and comparison between these data values.

An example of such a cityscape visualisation is shown in Figure 3.2. This particular example uses the same data set as shown in the 3D surface plot example of Figure 3.1. The maximum, minimum and average values for each row and column of the data are projected onto the side walls. A transparent selection field is also shown, highlighting any data with a value equal to or greater than 50. Due to the transparent nature of this selection plane, data items not fitting this criterion are still visible, providing a context for the results. Finally, a selected data item is shown with the average values for its row and column being extended to provide some context with neighbouring data.

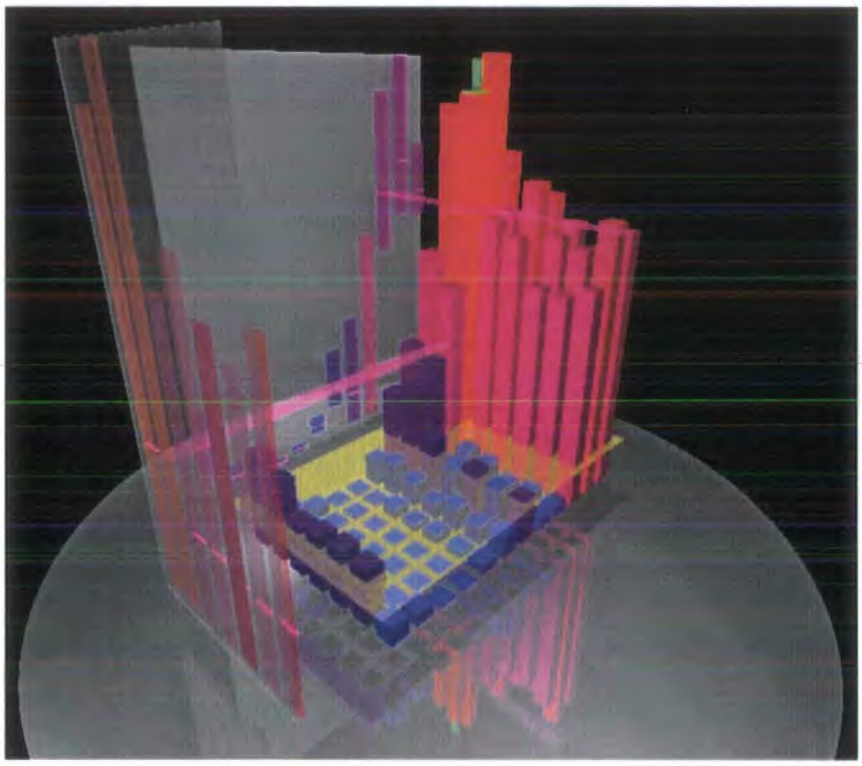


Figure 3.2. A Cityscape visualisation showing a variety of information.

As with surface plots, cityscape visualisations are designed to present predominantly numerical information. Such a visualisation would provide a suitable and intuitive display of numerical software information, for example software metrics.

III. Fish-eye views

The name given to this particular technique is taken from the similar effect produced by a very wide-angle 'fish-eye' lens. The fish-eye lens distorts the view so that objects close to the centre of the lens are magnified greatly, this magnification drops rapidly the further the objects are from the centre of the lens. This view results in objects that are at the centre of attention being shown in the greatest detail and with maximum emphasis, whereas objects on the periphery are shown with less emphasis. This allows objects of interest to be studied in detail, while still maintaining a notion of context or position with respect to other objects.

Fisheye views were originally investigated by Furnas [Furnas86], but have since received more attention and widespread use. The original views have been extended to give more control over the layout and to take into account the overall information structure [Sakar92].

The use of fish-eye techniques could prove to be useful in visualising large graphs. The technique will allow nodes of interest to be brought 'closer' to the viewer thus displaying greater detail while other nodes are moved further from the viewer and displayed in lesser detail. This provides the ability for the viewer to concentrate on the nodes of interest while still maintaining a view of these nodes' position within the whole graph structure.

Software systems are often visualised using a graph-based technique. Such graphs rapidly become very large and complex as the size of the visualised system increases, often resulting in a tightly cluttered graph structure. The use of fish-eye views can aid exploration of such graph structures by distorting the space around the viewer's focus, effectively spacing out the graph to provide a clearer view of the structure. The rest of the graph structure remains visible at the periphery of the users view, thus providing some context for their focus of interest. An example of the use of fish-eye views is demonstrated by the Rigi reverse engineering tool [Storey95].

IV. Benediktine space

Benediktine space is a term that arose from Michael Benedikt's research into the structure of cyberspace [Benedikt91]. Benedikt put forward the notion that attributes of an object may be mapped onto intrinsic and extrinsic spatial dimensions. Extrinsic dimensions specify a point within space, for example a set of Cartesian co-ordinates. Intrinsic dimensions specify object attributes such as size, shape, colour, texture, etc. An example of a Benediktine space could be to map an attribute such as student names onto an extrinsic dimension such as the x-axis and their exam marks to another extrinsic dimension, such as the y-axis. The class of degree that a student received could then be mapped onto an intrinsic dimension such as shape. Other intrinsic dimensions could then be used to encode other relevant information. By doing so we create an information space in which the very structure and position of objects provides insight into the data.

Benedikt also proposed two rules for cyberspace, the principles of *exclusion* and *maximal exclusion.* These rules attempt to clarify the positioning of data and in particular to avoid 'crowding' of data items. The principle of exclusion essentially ensures that no two data items can exist in the same location within space, i.e. their extrinsic dimensions must be different. This is stated as:

"Two non-identical objects having the same extrinsic dimensions and dimension values, whether at the same time, or including time as an extrinsic dimension from the outset, is forbidden, no matter what other comparisons may be made between their intrinsic dimensions and values."

The Principle of Maximal Exclusion expands upon the Principle of Exclusion by ensuring different data items are separated as much as possible, thus avoiding confusion produced by cluttering of objects. This is stated as:

"Given any N-dimensional state of a phenomenon, and all the values - actual and possible - on those N-dimensions, choose as extrinsic dimensions - as "space and time" - that set of (two, three, or four) dimensions that will minimise the number of violations of the Principle of Exclusion."

While this notion of intrinsic dimensions, extrinsic dimensions, exclusion and maximal exclusion may seem straightforward and common sense, it is still very convenient to be able to refer to objects within 3D space as having intrinsic and extrinsic dimensions. This work by Benedikt provides some clarity and a common frame of reference for describing object properties, within any form of 3D visualisation.

V. Perspective walls

Perspective walls [Mackinlay91] are a technique for viewing and navigating large, linearly structured information, allowing the viewer to focus on a particular area while still maintaining some notion of location or context. Perspective walls have some similarity to the fisheye views in that they allow a particular area of information to be viewed in detail while information close to this is still visible in lesser detail thus giving an idea of position and orientation within the data.

The two strategies previously used to display large volumes of information were the *space strategy* and the *time strategy.* The space strategy used layout techniques and graphical design to maximise a single display area, presenting as much information as possible. This technique suffered from information overload in that so much information was presented that extracting any detail became harder. The time strategy took the principle of breaking the information structure into a number of separate views, only one of which could be displayed at any one time. This allows the viewer to switch between the views and focus on particular information as needed. The problem suffered by this view was that it was easy to become lost within the data as no overall view or cues to the current location are provided.

The perspective wall addresses the above problems by effectively extending the time strategy with contextual cues. The perspective wall folds the linear structure of the information in 3D space, for example forming a cylindrical shell with the data mapped onto the interior or exterior surface (other configurations are possible). A single section of the data may be viewed in detail at any one time, with

adjacent sections being folded back on either side of the view to give cues to the position of the current section. On moving between sections the wall would rotate smoothly to bring the next section to the centre of the view.

An example of a perspective wall is shown in Figure 3.3. The linear information structure, in this case a syntax highlighted source code file, is displayed on the exterior surface of a horizontal cylinder. Controls are available to spin the cylinder to bring any 'page' to the front. Contextual clues to the position within the linear structure are also available as a vertical slide bar and also as a colour coded strip running along the left edge of the cylinder. The currently selected page, or facet of the cylinder, would be in clear view to the user. The neighbouring pages are also within view to a lesser extent, thus providing some additional context to the location of the current page.

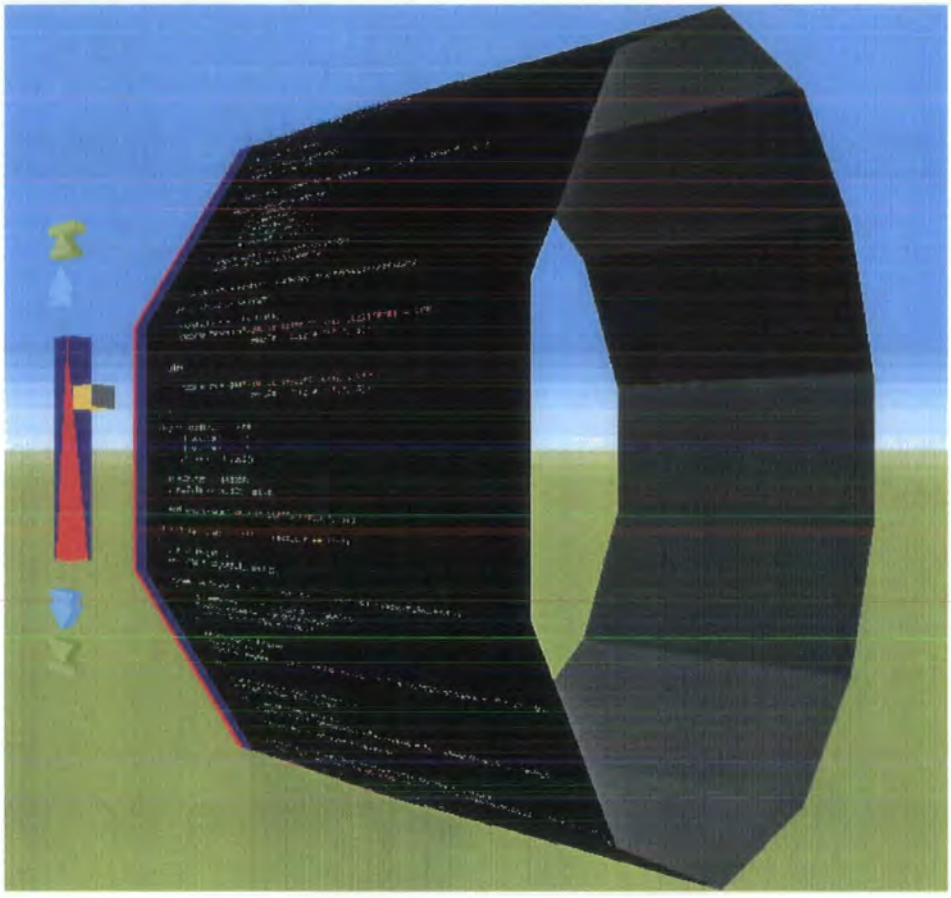


Figure 3.3. FileWall: An example of a perspective wall.

The perspective wall, cone trees, cam trees and 3D-Rooms (described below) are the result of research into an integrated *information visualizer* [Card91, Clarkson91] at the Xerox Palo Alto Research Centre (Xerox PARC).

Within software systems there is a large variety of information which can often only be displayed within a two-dimensional view. It is inevitable that for some aspects of a software system it will be inappropriate to radically change the way in which the information is presented. One such example is the presentation of the program source code. This is such a fundamental aspect of the software that any

form of display, other than the standard 2D textual view, would inevitably inhibit comprehension performance or present an excessive learning step. Perspective walls allow the presentation of 2D information seamlessly within a 3D environment, this would be particularly suitable for the display of source code files and other inherently 2D information within a 3D software visualisation.

VI. Cone trees and cam trees

Cone trees [Robertson91] are a three-dimensional extension to the more familiar 2D hierarchical tree structures. Cam trees are identical to cone trees except that they grow horizontally as opposed to vertically. The aim of cone trees is to allow a greater amount of information to be navigated and displayed in an intuitive manner and also to shift some of the cognitive load of comprehending the structure to the human perceptual system.

Cone trees are constructed by placing the root node of the tree at the apex of a translucent cone, positioned near the top of the display. All child nodes are then distributed at equal distances along the perimeter of the cone. This process is repeated for every node in the hierarchy. The base diameter of the cones is reduced at each level as the hierarchy descends in order to ensure sufficient space to accommodate all further leaf nodes.

The original cone and cam tree visualisations produced at Xerox PARC enabled the tree to be rotated smoothly to bring any particular node into focus. The smooth animation was found to be critical in maintaining the viewer's cognitive model of the structure. Sudden changes in orientation of the tree would require a significant amount of time to reorient the user's cognitive model.

Figure 3.4 shows a large example of a cone tree structure. Figure 3.16, appearing on page 58, illustrates a cam tree visualisation in which the tree grows horizontally from left to right. This often provides a much more intuitive and useful structure. Cam trees have the added benefit that textual labels flow nicely along the length of the tree, providing a much more compact and readable structure than cone trees.

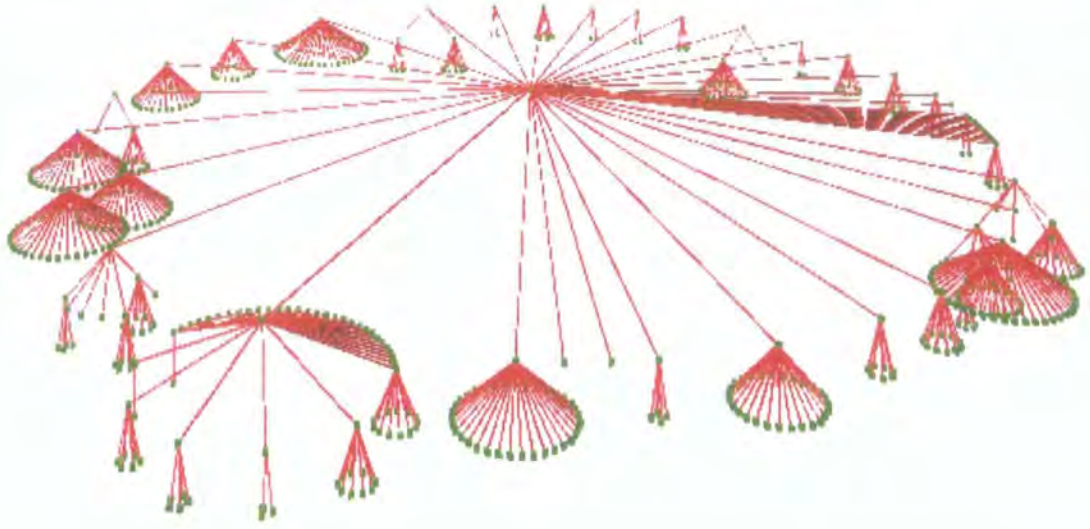


Figure 3.4. Example visualisation of a cone tree representing a directory structure.

Image courtesy of Dave Snowdon, Nottingham University.

Software systems contain a high degree of relationships between various components, for example function calls, file inclusion, class hierarchies and data dependencies. These relationships can be displayed using a graph-based visualisation, depicting each component and the relationships between them. Often these graphs will be predominantly hierarchical and the use of cone or cam trees would provide an intuitive depiction of their structure within a 3D environment.

VII. Sphere visualisation

The sphere visualisation is described by Fairchild *et al* [Fairchild93] as a 3D version of the predominantly two-dimensional perspective wall. The sphere visualisation is used within the VizNet visualisation system to view associative relationships between multimedia objects and a selected *object of interest* (OOI). Objects are mapped onto the surface of a sphere with highly related objects placed close to the OOI. Unrelated objects are displayed further from the OOI and thus become less visible as they move round to the opposite side of the sphere. This provides a natural fisheye view which emphasises objects of interest and de-emphasises less relevant objects.

The information is presented on the surface of a number of nested spheres. This provides a mechanism for representing different levels of information. The OOI is displayed on the outermost sphere with objects directly related to it fanning out around the surface of the sphere. Objects indirectly related to the OOI are considered lower level objects and are displayed on spheres nested within the outer sphere. The colour of the spheres becomes darker with increasing depth of nesting to give the user visual cues to their current location within the visualisation. Navigation through the visualisation is facilitated by rotating the sphere to bring nodes of interest into view and by traversing links to lower level spheres.

Figure 3.5. Sphere visualisation produced using VizNet.

Image courtesy of Kim Fairchild, Institute of Systems Science, National University of Singapore.

Software systems are comprised of a large variety of objects with various types of relationship between them [Chan98]. The sphere visualisation might provide a suitable form of display to highlight the various relationships with respect to a particular object of interest to the user. The visualisation would allow the user to traverse this network of relationships, moving their interest from one software object to another.

VIII. Rooms

The '3D-Rooms' metaphor is a three-dimensional counterpart to the desktop metaphor commonly encountered in computing today. 3D-Rooms were developed at Xerox PARC as part of their integrated *information workspace* and are a 3D extension to the original concept of 2D Rooms. 2D Rooms [Card87, Henderson86] built upon the notion of a multiple desktop workspace by adding features such as the ability to share the same objects between different workspaces, to overview the workspaces and also to load and save workspaces. Rooms allow another way for users to structure and organise their work by allocating certain tasks to certain rooms and moving between different rooms as needed. Within each room there will be a variety of information sources depending on the type of task allocated to a particular room. 3D objects may be present which represent, for example, documents or applications. The walls of the room may also contain information in a more conventional 2D manner, effectively using the walls as 2D displays.

The various rooms in an information complex such as this are connected via a number of doors. Doors lead from one room directly to another and, leaving a room by the back door, will return you to the previous room you were in. A necessary navigational aid for these structures is the provision of an

overview or floor plan of the rooms, including some indication to the content or tasks of the individual rooms. This allows the user to quickly move between unrelated rooms or tasks without having to go via a number of other rooms. One further useful addition to the rooms metaphor is the notion of *pockets* which allow the user to 'carry' information or objects between rooms or keep important items close at hand.

The 3D rooms metaphor does not provide any direct technique for the visualisation of software systems. It is predominantly an organisational metaphor, providing a unified interface and an understandable structure to a variety of other visualisations. Software visualisation is a complex problem and it is extremely unlikely that a single visualisation could cater for the needs of all its users. Software visualisation can not, and should not, be restricted solely to one view upon a software system. The use of the 3D rooms metaphor could provide a suitable way of bringing together the various specific software visualisations required by the user, amalgamating them into a single system with a unified interface.

IX. Emotional icons

Emotional icons [Walker95], taken within the context of a 3D data world, are objects that adopt varying behaviours in response to the presence of a user or possibly other icons. The aim of emotional icons is to make the use of the data world a more interactive, dynamic and individual experience. Emotional icons may respond to the presence or proximity of a user within the environment, their behaviour possibly being dependent on the profile of the user or their current activities and interests. Icons could possibly advance towards or retreat from the user, grow, shrink, animate or change their appearance all dependent on the relevance or importance of the data they represent to the current user. Icons could also respond to the proximity of other icons, those representing information of a similar nature may move together whereas dissimilar icons may move further apart. Emotional icons could provide a big step towards creating a 'living' data environment, one that responds to individual users in order to maximise the benefit of their time within the environment. Figure 3.6 provides an example of a 3D visualisation which is inhabited by emotional icons. Unfortunately emotional icons are predominantly interactive and rely heavily on animation, both of which cannot be represented within a static image.

The concept behind emotional icons and the notion of user profiles would prove a powerful and useful technique within software visualisation. Software maintainers will always have varying interests which are dependant on a number of issues such as the specific task they are trying to perform and the nature of the software system they are working on. By specifying these interests when creating a personal or task-oriented user profile then the visualisation can adapt to meet the specific needs of the maintainer. A software visualisation could be created as a dynamic environment which reacts to the presence and interests of individual users. Objects which are unrelated to a particular maintenance task could move away from the user, thus providing clarity and increased focus on objects which are of possible importance. There is a great deal of scope for the inclusion of such techniques, to some degree, into almost any form of 3D software visualisation.

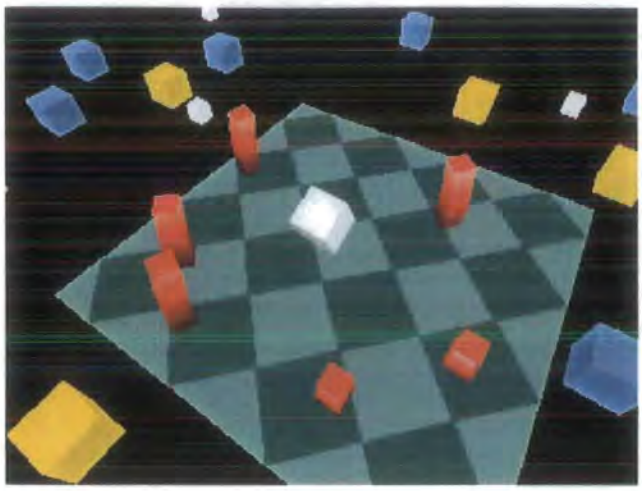


Figure 3.6. Emotional icons inhabiting a data space.

Image courtesy of Graham Walker, British Telecommunications Plc.

X. Self-organising graphs

Self-organising graphs typically refer to a particular technique used in automatically laying out graphs. Conventional layout techniques involve a function or routine that attempts to perform a suitable layout on a given graph while aiming to satisfy a number of aesthetic criteria or heuristics. Self-organising graphs allow the graph itself to perform the layout by modelling it as an initially unstable physical system and allowing the system to settle into a stable equilibrium. There is no need for the graph behaviour to accurately model an actual physical system, allowing the implementation of models geared towards different criteria such as efficiency, speed, accuracy and aesthetics.

The first application of this method to graph layout was by Eades [Eades84] and was named the *spring-embedder*. Eades' work evolved from a VLSI (Very large-scale integrated circuit) layout technique termed *force-directed placement* (FDP) [Quinn79] and has since seen a large number of improvements, modifications and similar algorithms. Graph layouts produced using such methods are often very satisfactory and aesthetically pleasing. Examples of the application of a force-directed placement algorithm to a large number of graphs can be found in work by Fruchterman and Reingold [Fruchterman91].

Eades' system was modelled around a network of rings and springs. The vertices of the graph represented the rings, and the edges represented the springs. The springs modelled did not obey Hooke's law, contrary to what you would expect from a physical spring. When extended past their natural length they exert an attractive force proportional to the length of extension. When compressed past their natural length they exert a repulsive force, again proportional to the length of the compression. This network begins in an unstable or high-energy state and, over a number of iterations, the forces within the system will attempt to reach an equilibrium or minimum energy level. Once this more stable state is achieved the layout is complete.

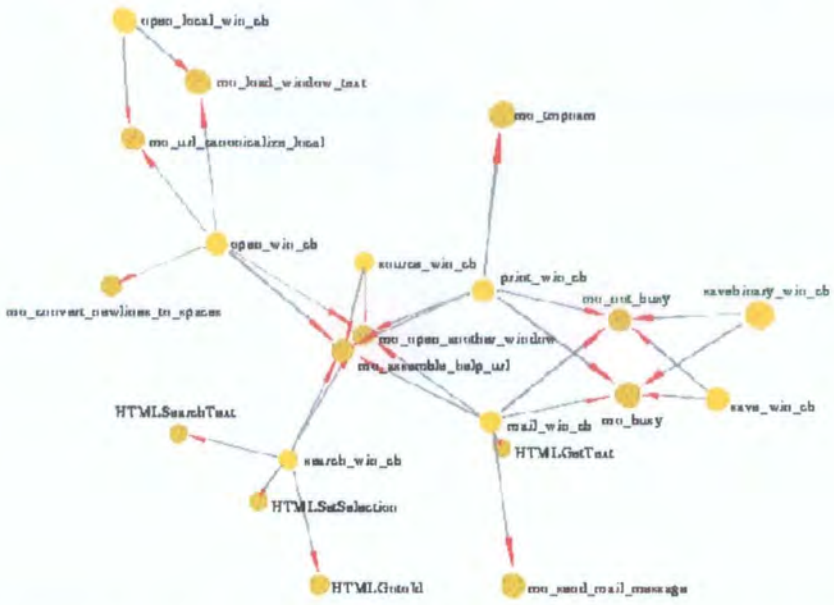


Figure 3.7. 3D graph produced using force directed placement algorithm.

This technique is often used in the display of large hyper-structures, which are highly connected graphs of interacting components. Examples of such structures are program call graphs, control-flow graphs and hypertext document relationships. The FDP technique is easily expanded into three-dimensional space producing very interesting and often optimal graph structures. The provision of an additional dimension allows much greater freedom for the simulated physical system to find a minimum energy state. Examples of the practical application of such algorithms within 3D information visualisation can be found in a number of research papers [Chalmers95, Hendley95a, Kings95]. Figure 3.7 shows a simple 3D graph, the layout being determined using an FDP algorithm. Another example of such a layout can be seen in Figure 3.21 on page 65. Figure 3.21 shows a large graph, the layout of which is the result of applying a force-directed placement algorithm, similar to the Eades' technique described above.

The possible application to software visualisation is in the visualisation of relationships between software components such as files, functions, classes and data objects.

XI. Spatial arrangement of data

By mapping data attributes to various intrinsic and extrinsic dimensions of 3D objects we begin to define the nature of the 3D environment as a whole. The shape, form and population of that environment will be described by the information we are visualising, in effect the information *becomes* the 3D environment. When discussing such visualisations, mention is often made of the *information terrain* or the *information landscape*. These terms refer to the data-dependent nature of the environment, as it is the information itself that defines the terrain of this environment.

A key problem faced when creating an information terrain is in defining a useful mapping from the data itself to a corresponding representation and location within the virtual environment (intrinsic and extrinsic dimensions). The requirement of this process is to create a spatial configuration from which the properties of data items within the information terrain and the relationships between them can be readily interpreted, simply from their position and presentation. Within the research of Populated Information Terrains (PITs), the following four approaches were investigated [Colebourne94, Benford94a, Benford94b].

A. Benediktine cyberspace

As mentioned in Section IV, Benediktine cyberspace is created by mapping data attributes onto the intrinsic and extrinsic dimensions of the corresponding 3D object. This method would involve generating a schema for the database to be visualised in order to determine which data attributes are mapped to the intrinsic and extrinsic dimensions of the visual representation. In this case the issue of spatial arrangement is of little concern as this will be defined by the appropriate data values. However, certain steps must still be taken to ensure best use of the space available and to provide the most desirable information terrain. A desirable information terrain, in this case, refers to an understandable and easily navigable environment. The principles of exclusion and maximum exclusion, mentioned previously, attempt to provide some support for this.

An example application of a Benediktine space is Q-PIT, which is described further in Section 3.3.2.

B. Statistical clustering and proximity measures

Statistical methods can be used to analyse database contents in an attempt to group items according to some measure of their semantic proximity. For example the contents of a document store could be grouped corresponding to matching keywords. Analysis performed on these information stores typically result in a number of 'scores' for documents, which can then be used to create a suitable mapping into Benediktine space. The closer together objects are semantically, then the closer they will be within the information terrain. Systems adopting this approach include VIBE [Olsen93] and BEAD [Chalmers92], though the original idea of VIBE has been developed further and extended into three dimensions to produce VR-VIBE [Benford95]. Clustering is often a very subjective approach due to the number of possible interpretations of 'close', both semantically within the information and also graphically within the visualisation. For example, two objects within a visualisation could be described as being close if they were positioned only a short distance apart. The same two objects could also be described as being close if they were positioned far apart, but had a very similar appearance.

C. Hyper-structures

Hyper-structures are created from information stores consisting of a variety of data objects, with any number or arrangement of explicit relationships between them. An example of a hyper-structure could be formed from the document structure within hypertext systems such as the World Wide Web. Another

example is the dependency graph formed from components within a software system, such as a function call-graph. These structures are typically viewed using 3D extensions to standard graph drawing algorithms, or by more inherently three-dimensional visualisations such as fish-eye views, cone trees or perspective walls. An example of a 3D graph depicting a hyper-structure is shown in Figure 3.21, on page 65.

D. Human centred approaches

This approach is typically not suitable for automatic generation of visualisations and often relies on the user or system designer to create appropriate representations of the data to be visualised. The technique lends itself to the notion of abstract real-world metaphors such as cities, buildings and rooms within which the user navigates the data through more familiar surroundings. The problem with this approach is that the generation of such metaphors is time-consuming and difficult. Creating an abstract environment such as this, which matches both the user's conception of these real-world surroundings *and* appropriate structure and representation of the data being visualised, is not an easy task.

While such an approach has obvious benefits, in particular the familiarity of the environment to the user (through the use of real-world metaphors), encoding information into a strong metaphor-based environment is particularly difficult. This is especially so if attempting to avoid breaking any metaphors used. The benefit of using metaphors is almost completely negated if these metaphors can occasionally be broken within the environment, thus destroying the user's expectations. Ideally, a visualisation should aim to strike a careful balance between the use of metaphor and the use of more abstract constructs. Metaphors are useful for encouraging comprehension of the environment and the information contained within it, whereas abstract constructs are useful for actually encoding information and offering maximum flexibility for the display of that information. Visualisations that make use of very strong metaphors are often very simplistic, as visualisations, providing nothing more than an alternative access to information which could be more efficiently retrieved using more standard interfaces.

An example of a system that makes use of very strong real-world metaphors is VIRGILIO [Levialdi95]. VIRGILIO uses the notion of corridors, elevators, rooms, filing cabinets, books, and various other appropriate real-world objects in order to display information retrieved from a database query.

XII. The information cube

The information cube is a technique developed by Rekimoto and Green [Rekimoto93] to visualise hierarchical information using nested translucent cubes. The information cube is loosely based upon the 2D tree-map visualisations of Johnson and Shneiderman [Johnson91, Shneiderman91]. Tree maps visualise hierarchical information in a rectangular 2D display using a space-filling approach to maximise display usage. The available display space is partitioned into a number of rectangular bounding boxes, which represent the tree structure. Parent items are subdivided into boxes that represent their children, and so on. Facilities are provided for the user to interact with the visualisation

and specify the presentation of both structural and content information, for example varying the 'depth' of information or it's colour. Information within the display can be given a degree of interest or measure of importance, resulting in it being allocated a greater proportion of available display space.

The information cube technique effectively extends 2D tree maps into 3D. Hierarchical information is presented as nested translucent cubes, with the level of transparency varied to control the 'depth' of the visualisation and hence the amount of information presented. Transparency and shading is the main technique used to control the information content of the cube visualisations. This transparency allows the user to view the content of the cubes and their children, while gradually hiding inner information as it becomes less relevant (to the current focus). Without this reduction in presented information the visualisation would become too complex to understand. A title is attached to the surface of each cube. Leaf nodes, which contain no further nested cubes, are represented as 2D tiles with the title presented on the surface. Cubes may contain arbitrary information and are not restricted to containing only further cubes. Other 3D visualisations or information may also be presented within the cube structure.

Figure 3.8. An information cube visualisation.

Image from Jun Rekimoto's web page, Sony Computer Science Laboratory Inc.

Software systems are typically comprised from a hierarchy of objects such as files, classes, methods and data objects. The relationships within this hierarchy can often be described as a 'contains' relation; files contain classes, classes contain methods, and classes also contain data objects. The use of transparency and similar techniques to the information cube could be used to highlight these relationships within software systems.

3.3.2 Research visualisation systems

The previous section described various techniques used within 3D information visualisation. This section describes a number of information visualisation systems that make use of these techniques. The majority of these systems are prototypes or concept demonstrators and, as such, their merits or operation can only be assessed from their associated publications. This section aims to serve as a

synopsis of the publications describing these systems, and to provide insight into the possibilities afforded by 3D visualisation. The wide variety of visualisation systems presented in this section illustrates the scope of the field and also highlights the potential for applying similar visualisation techniques to 3D software visualisation.

This section provides a varied sample of 3D information visualisation systems. While there are many other systems which have not been included here [Walker93, Walker95, Kings95], this section aims to provide a sample of the diversity of visualisation techniques used and the various approaches adopted by researchers.

I. Database Visualisations

A. AMAZE

AMAZE [Boyle93, Benford94a] is a three-dimensional visual query language, created at Aberdeen University specifically for the complex analysis of protein structural data. AMAZE was developed as the next logical step in a series of advances to the existing database interface. AMAZE allows users to visually construct a query within the 3D environment then explore the resulting data sets in the same environment. The user constructs a query by interacting with a three dimensional representation of the schema. The completed query is passed to the remote database for evaluation and the resulting data is displayed as a set of 3D objects which the user can investigate further for additional information.

The AMAZE interface uses a mixture of 2D and 3D to provide an extension rather than a replacement for the more typical, and often more appropriate, 2D windowing interface. 3D environments used in the system are supported within separate windows and navigated with a conventional mouse.

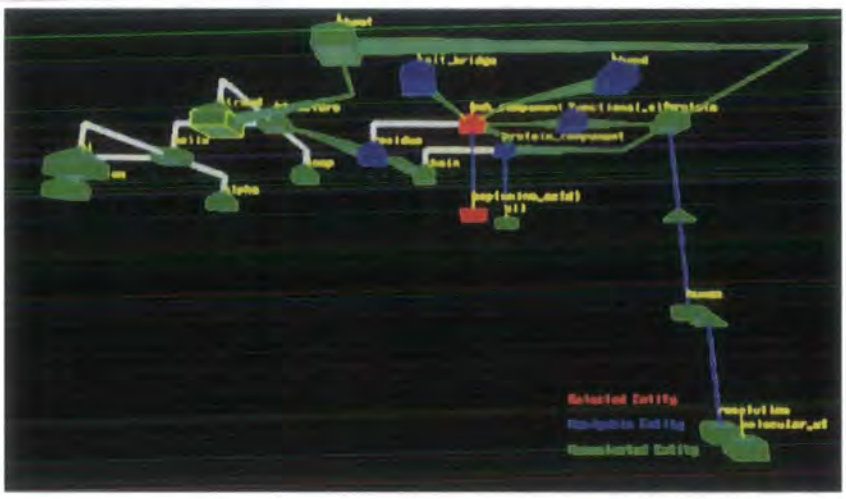


Figure 3.9. Formation of an AMAZE query.

Image courtesy of John Boyle, Aberdeen University.

A session using the AMAZE interface would begin with the user navigating the 3D schema then attaching queries to relevant entity classes and adding constraints by way of a dialogue box. The user

can build up a more complex query by navigating to a number of interesting entity classes and placing queries on each. An example of an AMAZE query is shown in Figure 3.9. Results from the query are represented within the 'Result Maze', which the user can explore through the 3D interface. The result maze is constructed by grouping resulting instances of the same entity class into sets then colour coding these sets. Each instance within a set is represented as a single 3D cube, except for the case in which there are a very large number of instances - these being represented as an abstract group object. Finally, to view a particular instance, the user selects that instance causing an appropriate external viewer to display the data.

B. VIRGILIO

The Virgilio project [Levialdi95] addresses the issues of automatic metaphor generation for use in visualising database query results using a virtual environment. The project concentrates mainly on the display of database results containing multimedia information. The project members define a theoretical framework for determining whether a possible metaphor is suitable for displaying the results of a given query, and more specifically, if that metaphor is structurally sound. By adopting a virtual environment and 3D visualisation for display of the results, Virgilio hopes to allow users to easily explore deeply structured chunks of information using a familiar and intuitive suite of appropriate metaphors.

The Virgilio system aims to provide a general-purpose interface for information visualisation by using dynamic metaphor generation. One of the most important aspects of this approach is that the system must choose the best and most suitable metaphor from the many possibilities. In order to decide what is a good metaphor for any particular user of the system, the notion of a user profile is necessary.

Figure 3.10. Virgilio corridor metaphor for selecting band names.

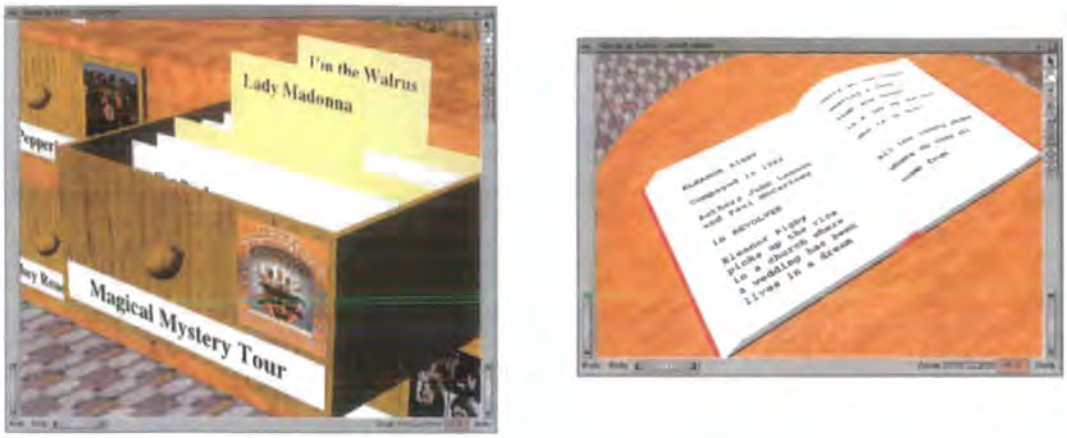
The Virgilio prototype system uses the Virtual Reality Modelling Language, VRML, to produce 3D scenarios representing the query results. Users can then navigate and explore this environment and the objects contained within it, and by doing so they are navigating and exploring the query result data. Information is embedded in the many visual details provided by the virtual world metaphor, meaning that no specific learning is required to interact with the visualisation.

Figure 3.11. Virgilio room metaphor, showing information on a selected band.

Figure 3.10, Figure 3.11, and Figure 3.12 illustrate a portion of a Virgilio visualisation and exploration scenario. The scenario is based upon a music database containing information such as songs, lyrics, albums and artists. The visualisation generated has selected a 'rooms' metaphor and has structured the information into a virtual building. Initially the user is presented with the external view of the building. On entering the building they can progress through the foyer, which contains decorative features pertaining to the nature of the database, and enter the elevator. Each floor in the building represents a style of music such as classic, rock, pop and jazz. The user indicates their interest by selecting the corresponding button in the elevator.

On exiting the elevator (Figure 3.10) the user is presented with a corridor, which corresponds to various bands that have produced music in this particular category. Information on the category of music selected is shown on a floorboard just outside the elevator, and a banner indicating the category is displayed. Each door along the corridor leads to a room pertaining to one artist or group, with the room label being clearly visible from the corridor.

On entering a particular room (Figure 3.11) the user is presented with a variety of information on this particular artist or group. Within the room is a set of cabinets containing drawers, each of which correspond to a CD or Album produced by the group. Again, opening these drawers (Figure 3.12a) leads to further, more detailed information on each of the tracks of that album.



(a) (b)

Figure 3.12. Viewing albums, songs and lyrics in Virgilio.

The Virgilio system also attempts to reduce the cognitive load of users as they refine or expand queries. The system will attempt to accommodate new information in a seamless way, using redundant visual objects. For example, if the user refines the query to display the lyrics of a particular song, the system displays this information by presenting a book on a previously empty table (Figure 3.12b). The aim here is to minimise the amount of change within the environment, for a corresponding change within the query, thus minimising disorientation and the need to relearn the visualisation structure.

II. Populated information terrains (PITs)

The concept of a Populated Information Terrain (PIT) aims to provide a useful database or information visualisation by taking key ideas from the fields of computer supported co-operative work (CSCW), virtual reality and database technology. The definition of a PIT is taken as a virtual data space that may be inhabited by multiple users [Colebourne94, Benford94b]. The philosophy behind PITs is that they should support people to work co-operatively *within* data as opposed to merely *with* data. Possible applications of PITs can apply to any form of information store, for example: library catalogues; large hypertext systems such as the World Wide Web; or even large software systems through an appropriate set of visual abstractions.

PITs were developed in response to a lack of sufficient support for co-operative work and information browsing within current information stores. The lack of support for information browsing is addressed by the use of virtual reality to create a rich data environment in which the user can navigate and explore freely. VR provides the capabilities for the user to become immersed within the data and to explore and interact with it while making full use of his or her perceptual and intuitive skills. Co-operative work is supported in PITs by populating the virtual data environment with users. This allows users of the PIT to be aware of the presence and actions of other users within the same PIT and the ability to communicate with them, irrespective of their physical location. This awareness of other users enables a true sharing

of information. A user interested in a particular subject area may question other users browsing the same information or even ask directions to relevant data.

A. Q-PIT

Q-PIT [Benford94b] is a prototype implementation of a populated information terrain, following the Benediktine approach. Q-PIT is implemented using the World ToolKit virtual reality library and is capable of processing and displaying a simple database of named tuples. Q-PIT allows the display, querying and manipulation of the database within the three dimensional environment, in addition to allowing this environment to be shared by multiple users.

Within Q-PIT, attributes of the data to be visualised are mapped onto the extrinsic and intrinsic dimensions of the 3D representations. The extrinsic dimensions used are the X, Y and Z co-ordinate axes, while intrinsic dimensions supported are shape, height and angular velocity. Users within Q-PIT are embodied as monoliths - tall, thin cuboids each allocated a different colour to represent their identity. Users are aware of the presence of other users within the same PIT and can obtain information on them or open a communication channel with them.

Manipulation of the underlying data in Q-PIT is provided via a number of methods. The user may either select a single data item via its 3D representation or they may issue a query, resulting in all matching data items being highlighted. The actual data in the tuple can be modified by selecting an object then using a textual interface to modify the fields. Once altered, changes to either intrinsic or extrinsic attributes are reflected immediately in the visualisation, the latter of which results in a smooth transition between positions.

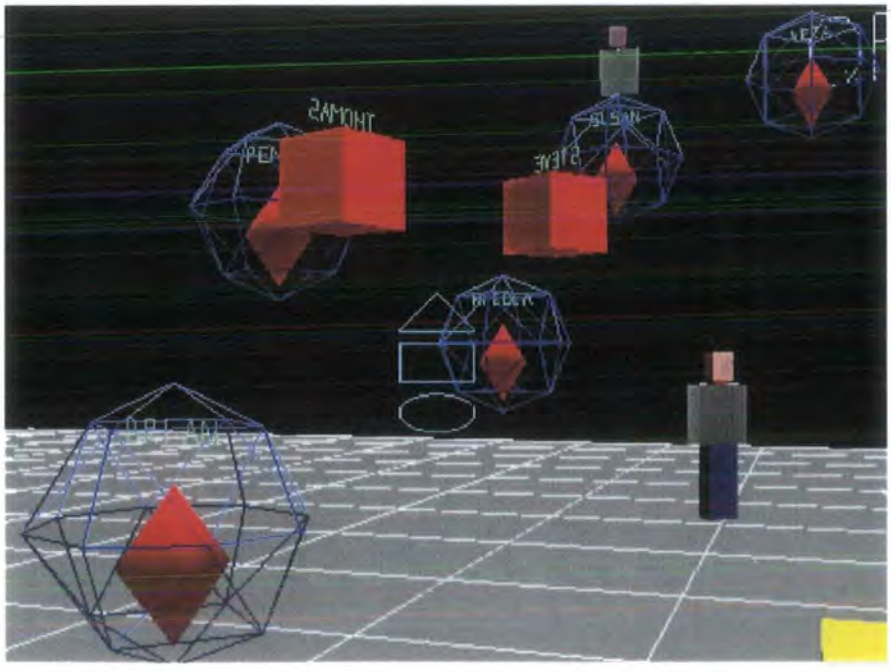


Figure 3.13. Q-PIT Visualisation showing multiple users.

Figure 3.13 shows a Q-PIT visualisation containing several users and a number of data items. In this version of Q-PIT, the representation of the users has been refined to provide a more human appearance, though more importantly it provides a better impression of the orientation and focus of interest of each user. Data items are also visible with textual labels appended. Several of the data items are shown to be selected, possibly by a user or the result of a query. The spherical mesh surrounding a data item indicates this selection.

B. BEAD

BEAD [Chalmers92] is a 3D document visualisation system that makes use of the DIVE toolkit and Visualiser to produce a multiparticipant environment. BEAD utilises a variant of the force-directed placement algorithm to arrange document representations within 3D space, the end result forming an information terrain. The layout algorithm is driven with a document similarity metric and strives to produce a terrain in which closely related or similar documents are grouped physically close together. This metric is based on word co-occurrence, i.e. a measure of the number of words occurring in both documents. Similar documents are attracted towards each other, whereas dissimilar documents will repel each other if they are too close together.

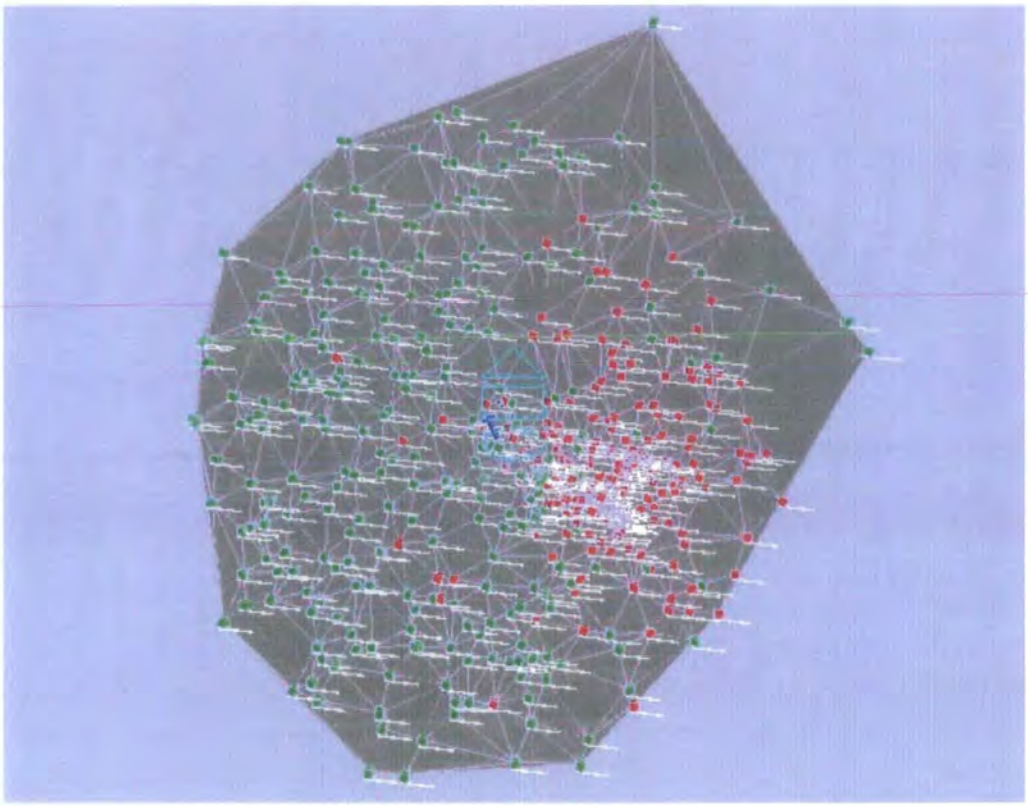


Figure 3.14. Screenshot of BEAD showing an overview (looking down) of a data landscape constructed from over 500 bibliographic references.

Image courtesy of Mathew Chalmers, Union Bank of Switzerland.

An important point of note in the development of BEAD is that the original system attempted to match the physical distances between documents as closely as possible to the similarity metric. This was facilitated by allowing the document nodes to move freely within three dimensions, thus creating a ball or cloud-like shape. It was found that the visual complexity of such structures and the inherent problems of navigation within them were too great. Development of BEAD then took on a more two-dimensional aspect, with the document nodes being forced to form a flatter terrain, though still with some degree of vertical movement. The document nodes are then meshed together by polygonal shapes, the resulting visualisation resembling an island with an undulating terrain.

Multiple users may populate the virtual environment or data landscape produced by BEAD. Users are represented as T-shaped figures constructed of two cuboids, with a further two cuboids to represent eyes. This allows other users to see exactly how a user is orientated and in which direction they are looking. Users can interact with the data by either selecting specific objects or by performing searches, resulting in matching documents being highlighted. By browsing the overall data terrain certain features may become apparent such as dense areas and rough terrain, again providing additional information on the similarity between documents. If a document is highlighted in a search then it is possible that documents close to it, but not highlighted, may also be of some interest due to their high similarity. Inherent in the layout algorithm is a tendency for documents, which are only slightly related to the main theme, to be pushed towards the shores of the island. Geographical features such as the shoreline, densely clustered areas, and rough areas allow the users to build up a notion of how to navigate the information and where to find information of most value.

C. VR-VIBE

VR-VIBE is a virtual reality extension to the original two-dimensional VIBE system and enables the 3D visualisation of large document collections. The original VIBE system [Olsen93] was developed at the University of Pittsburgh and provided users with a 2D visualisation of a collection of documents. This visualisation gives users an overview of the documents they are interested in with respect to the whole collection. The visualisations in VIBE are constructed by defining a set of 'Points of Interest' (POIs) containing keywords which will be used as a basis for the query. A full-text search is performed on the document collection and each document is scored for relevance to each POI. The visualisation is constructed by distributing the POIs evenly on a 2D plane then placing the document icons between the POIs with respect to their relevance to each POI. For example, with two POIs A and B, a document may have a relevance score of four to POI A and a score of three to POI B. This document could then be placed at a ratio of 4/7th of the distance along the line joining A and B, thus showing the proportion of relevance attributed to each POI.

VR-VIBE exploits 3D visualisation techniques to tackle the problems of display crowding and also allow the possibility of larger numbers of POIs to be used. The document positioning technique used in VIBE is simply extended into three dimensions with VR-VIBE and is essentially the same. VR-VIBE offers two slightly different layout methods. One method is very similar to the VIBE layout and

constrains the nodes to a 2D plane when organising their position between POIs. A vertical displacement is then introduced to each object to represent their degree of relevance. The second method (Figure 3.15) allows the objects to be positioned at any point in 3D space, within the confines of the POIs. Intrinsic dimensions of each object, for example colour, are exploited to indicate selection and also to emphasise relevance. The colour of a document object serves as an indication as to the level of relevance to POIs. This allows distinction between two documents, which are both equally relevant to two POIs, however the actual scores may be different. For example a document with scores of three and three would be coloured differently to one with scores eight and eight. Object shape is also used to distinguish between the documents, POIs and the users. These are represented respectively as cuboids, octahedrons and 'blockies' (similar to Q-PIT users).

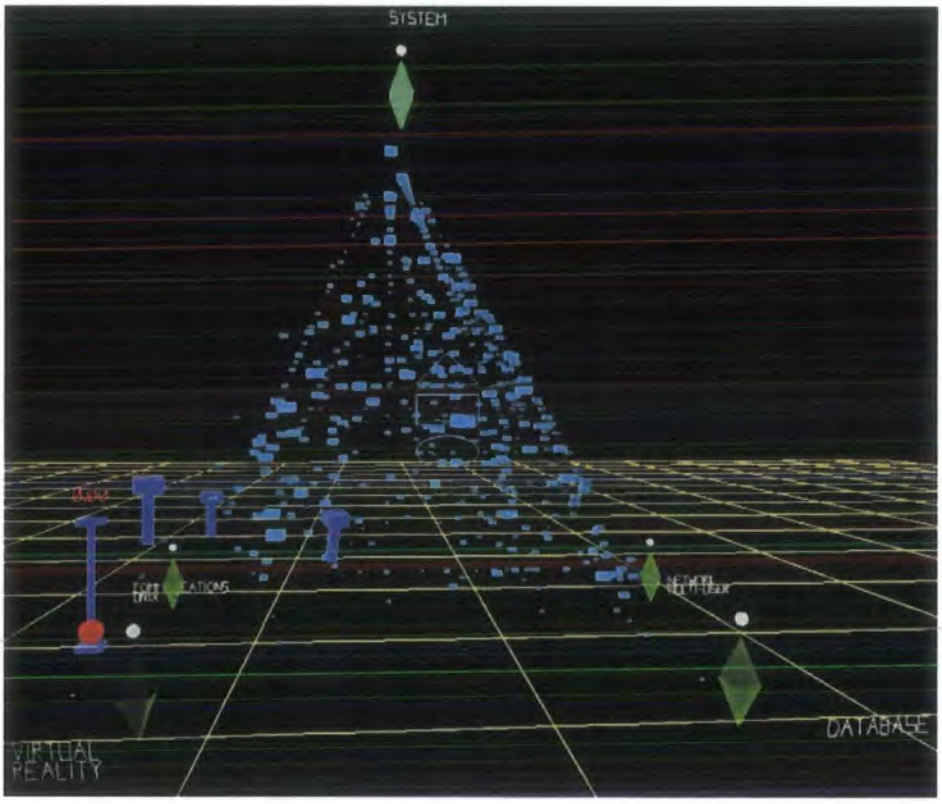


Figure 3.15. Screenshot of VR-VIBE showing a PIT containing 5 POIs and a number of users.

Image courtesy of Dave Snowdon, Nottingham University.

VR-VIBE allows users to navigate freely through the information structure, select documents, perform queries, apply filtering or request additional information. A high degree of interaction is supported through the 3D interface and users are able to add POIs, switch POIs on or off to try different configurations, and even move POIs to see which documents are pulled along after them. Users may also define a number of viewpoints within the visualisation and switch between them at will. This allows comparison of different views on the information and also aids in navigation by providing a set of reference points that can be quickly returned to. Finally, as with Q-PIT, VR-VIBE supports multiple users within the same visualisation, each aware of the presence and actions of each other.

D. LyberWorld

LyberWorld [Hemmje93, Hemmje94] is a prototype information retrieval user interface which aims to make full use of the advantages afforded by 3D graphics. LyberWorld concentrates on the visualisation of an abstract information space, *Fulltext*, and providing a user interface for the retrieval system *INQUERY*. The visualisations used in LyberWorld aim to provide an intuitive and natural interface to enable efficient searching and browsing within this abstract information space. The goal of LyberWorld is to provide a tight coupling between the graphical model of the information space (the visualisations) and the cognitive spatial model maintained by the user. If such a coupling exists then the visualisations should aid the user in exploring and querying the database, navigating and orientating themselves within the data, and judging the relevance and context of query results.

The information within LyberWorld is modelled as a network of documents and terms. The relationships in this network are formed by a measure of the relevance between these document and term nodes. Two main visualisations are used within LyberWorld, the *NavigationCone* and the *RelevanceSphere*. NavigationCones provide a visualisation of the retrieval history, i.e. the extent to which the data space has been explored and the query paths travelled through the data. The RelevanceSphere provides a display of the retrieved documents and their relevance to each of the search terms in the query. Additionally, the RelevanceSphere provides a visual indication of document clustering, highlighting documents that possibly belong to a closely related subject area.

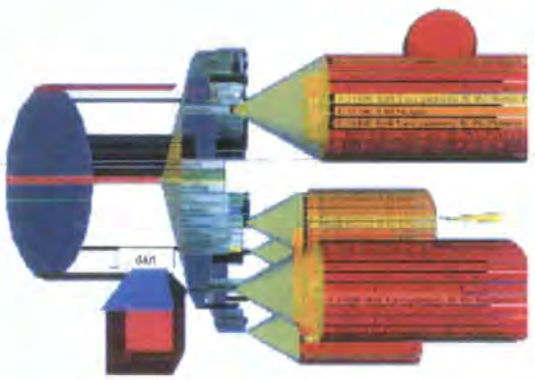


Figure 3.16. Screenshot of LyberWorld showing the NavigationCones visualisation.

Image courtesy of Matthias Hemmje,
German National Research Centre for Computer Science (GMD).

LyberWorld incorporates the RelevanceSphere visualisation to aid in judging the relevance of items within the result space. The RelevanceSphere attempts to tackle the ambiguity problem associated with visualisations such as VIBE and VR-VIBE. This ambiguity occurs because a particular point in space within a VIBE visualisation can be attributed to a number of relevance ratios to different points of interest. This problem is described by Hemmje *et al* [Hemmje94] and also by Benford *et al* [Benford95]. The RelevanceSphere addresses this problem by transforming the network into a spatial

visualisation that displays the relevance path lengths while still maintaining the numerical proportions. This approach allows path lengths to be easily compared with one another.

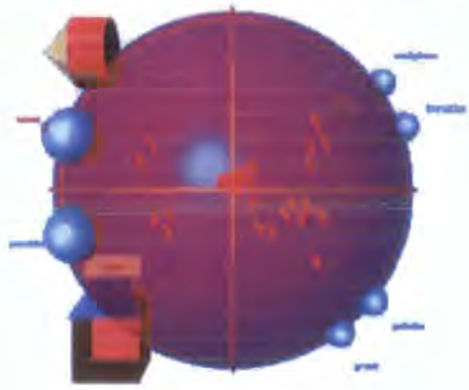


Figure 3.17. Screenshot of LyberWorld showing the RelevanceSphere visualisation.

Image courtesy of Matthias Hemmje,
German National Research Center for Computer Science (GMD).

The use of three-dimensional space to contain the document and term nodes leads to both advantages and disadvantages in the case of the RelevanceSphere. The main advantage is that the added display volume and thus increased spatial freedom reduces the probability of unclear positioning of document nodes, although it is still possible for ambiguity to arise. One problem encountered, typical in many 3D visualisations, is the reliance on the user's perception of depth. When viewing a static 2D image of a 3D structure it is often hard to judge depth, size and positioning. The RelevanceSphere deals with this problem by allowing the user to rotate the sphere freely and examine the visualisation from any angle. Additional features allowing the user to interact with the RelevanceSphere are provided. These facilities provide methods of investigating and manipulating the result space. Users may alter parameters such as the *document density*, *term attraction* and *scaling* of the visualisation [Hemmje94]. This interaction is an important contributor to increasing the user's understanding of the query results and the structure of the visualisation.

E. Vineta

Vineta is a prototype information visualisation system developed by Uwe Krohn [Krohn96a, Krohn96b]. Vineta allows the visualisation, browsing and querying of large bibliographic data without resorting to typing and revising keyword based queries. Similar to VR-VIBE, LyberWorld, and BEAD visualisations, Vineta presents documents and terms as graphical objects within a three-dimensional space, the '*navigation space*' [Krohn96b]. The positioning of these objects within that space encodes the semantic relevance between documents, terms and the user's interests. Vineta differs greatly from other visualisations in one distinct area, that is, the number of dimensions or terms it can present. Systems such as VR-VIBE are limited by the number of terms or points of interest (POIs) they can present simultaneously. Vineta applies techniques from multivariate analysis and numerical linear algebra for mapping documents and terms into the three-dimensional navigation space. This allows the

visualisation of much larger term sets or POIs by providing a mapping from the multidimensional document space into the 3D navigation space.

Vineta is built upon the premise that navigation through an information space can be an effective means of retrieving information of interest. Krohn states that informational navigation is strongly connected with the human intuitive comprehension of abstract facts by means of analogies with familiar concepts such as location or motion. Vineta uses spatial proximity to represent semantic similarity between objects (i.e. documents). As with previously described systems, the notion of similar documents is emphasised by placing them physically close together within the information space, whereas dissimilar documents are placed further apart.

Vineta has employed two main metaphors for producing visualisations, the galaxy metaphor and the landscape metaphor. The landscape metaphor has superseded the galaxy metaphor in the final implementation of Vineta, proving more intuitive and easier to comprehend.

The galaxy metaphor represents the navigation space as a galaxy of stars. Documents are presented as fixed stars while terms are presented as 'shooting stars'. Semantic similarity is encoded in the galaxy metaphor by the proximity of the stars, i.e. similar documents and terms are grouped closely together.

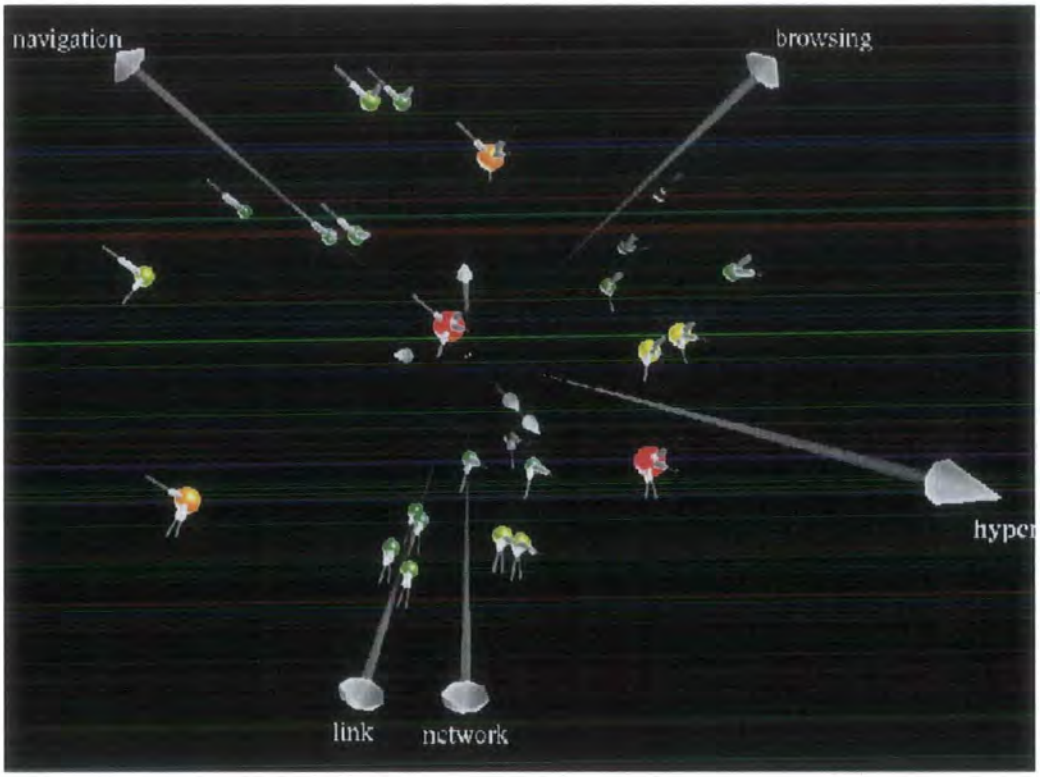


Figure 3.18. Screenshot of Vineta, showing the galaxy visualisation.

Image courtesy of Uwe Krohn, University of Durham.

The landscape metaphor represents the navigation space as a flat, textured surface containing flowers with stems and petals. The textured surface of the landscape helps to give the viewer an indication of

the depth or distance of the documents (flowers). The inclusion of this ground plane was encouraged by the study of ecological optics which emphasises that perception of objects should never be considered apart from a textured ground surface [Krohn96b]. Flowers nearer to the user's viewpoint are of more relevance to the initial query than flowers further from the viewpoint. The stems of the flowers aid the user by extrapolating the flower's position onto the ground plane. The direction and colour of petals on the flowers represent the search terms and their relevance to each document.

Figure 3.18 shows an image of the Vineta galaxy display. Terms are represented as arrows while documents are presented as spheres. Figure 3.19 shows an image of the Vineta prototype, demonstrating the landscape metaphor. It can be seen from this image that the introduction of the ground plane helps greatly in determining the relative positions of the documents (i.e. the flowers). In both visualisations, documents closer to the user's viewpoint are of more relevance to their overall interests.

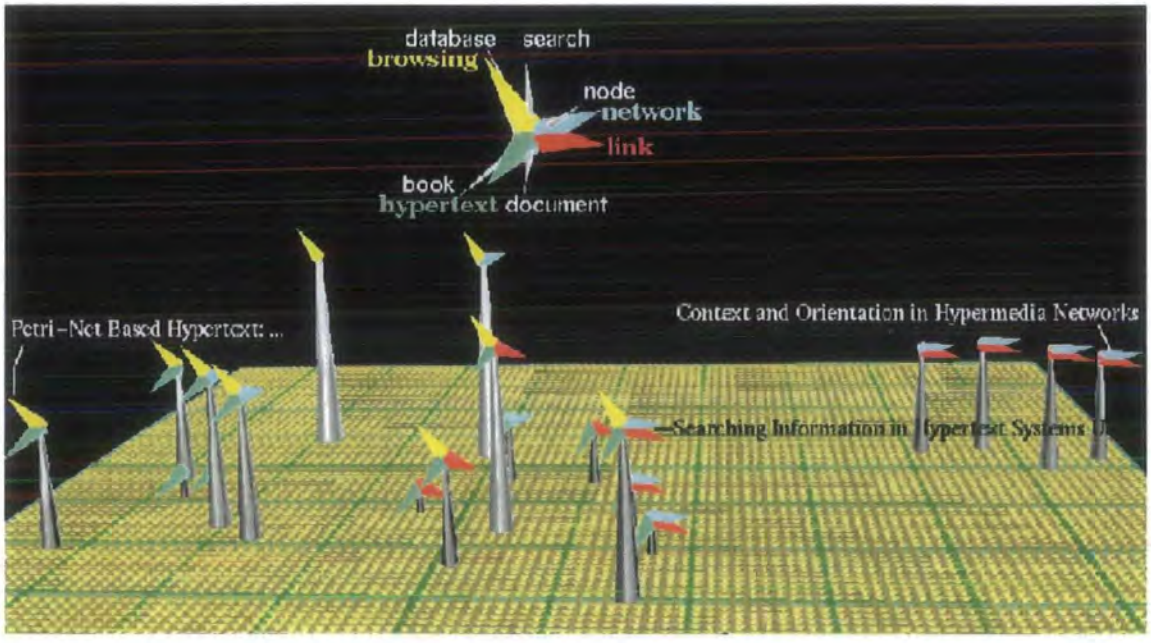


Figure 3.19. Screenshot of Vineta showing the landscape visualisation.

Image courtesy of Uwe Krohn, University of Durham.

III. Legibility enhancement

Research into legibility enhancement attempts to aid navigation through large virtual information spaces by applying principles from the more physical area of city planning. The legibility of an urban environment is judged by the ease with which inhabitants can develop a cognitive model of their surroundings over time, thus aiding their navigation through the environment. It has been argued that the legibility of urban environments can be improved greatly by careful design of certain key features. This section will concentrate on research performed at Nottingham University into the application of legibility features to 3D information visualisations [Ingram95a, Ingram95b].

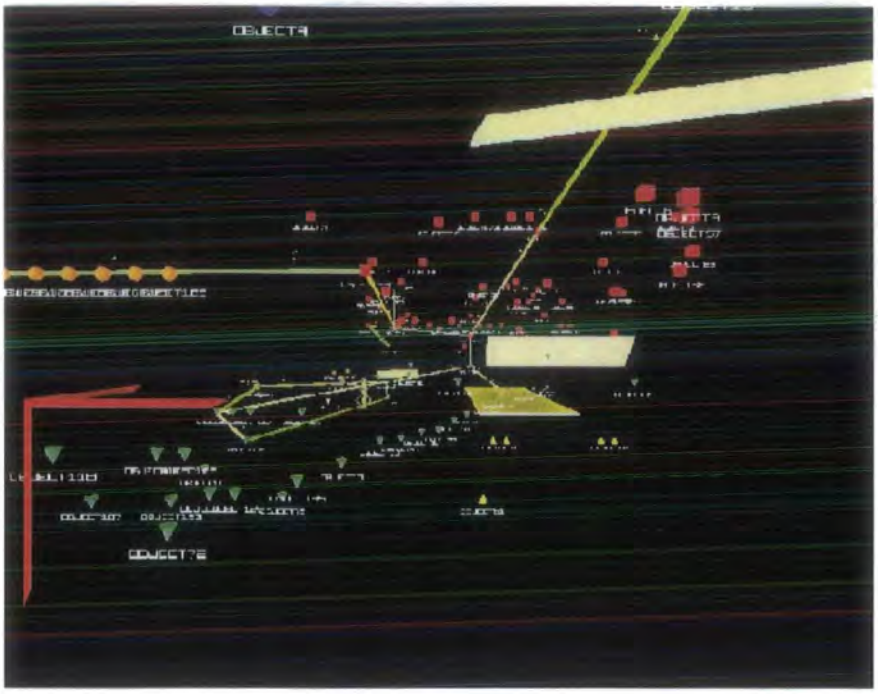


Figure 3.20. Screenshot of a Q-PIT visualisation after application of LEADS.

Image courtesy of Rob Ingram, Nottingham University.

The legibility of an environment relies on the ease with which an inhabitant can construct a cognitive model or internal representation of the environment. These cognitive models can be classified into two categories, linear and spatial. Linear models are based upon movement through the environment and observation of features enroute. Spatial models represent an overall model of some area of the environment which can be mentally manipulated and updated with information gained from linear models.

In 'The Image of the City' [Lynch60], Lynch identifies five major elements which are used in constructing cognitive models of an urban environment. These elements were identified as a result of an experiment in which inhabitants of various cities were questioned on the city they resided in. Information was gained using various techniques such as interviews, written descriptions of routes through the city or by drawing maps. These five features are described here in addition to how they are applied to information visualisations by Ingram and Benford [Ingram95a]. The system developed at Nottingham is named LEADS (**LE**gibility for **A**bstract **D**ata **S**paces) and implements a number of algorithms to structure data according to these key features. Figure 3.20 shows a Q-PIT visualisation after the application of LEADS enhancement features. Prior to application, the environment consisted of simple geometrical objects representing data items. The LEADS system preserves the extrinsic dimensions of the data items, but modifies intrinsic attributes such as colour and shape to create districts. Other LEADS features such as paths, edges and landmarks are clearly visible and impart a greatly increased sense of structure to the visualisation.

A. Districts

Districts are distinct areas within an environment, which can be abstracted into a single entity. Such areas must usually contain some form of commonality or character in order to be considered as a whole. Districts can be identifiable in a real world sense, for example by the nature of the buildings within them. An example of two districts could be a residential district and a commercial district; both are distinct abstractions of the buildings within each district.

Identifying districts within an abstract data space is a process of determining similarity and dissimilarity between data items. This process is fairly complicated when attempting to create a general method applicable to a wide range of data spaces. The LEADS system employs cluster analysis techniques to identify areas of similarity, then groups all data items belonging to a particular area. The actual display of districts within the virtual environment is fairly arbitrary, the requirements being that the data items within a district appear similar and that districts as a whole should appear sufficiently different. One possible representation would be to colour all data items within a district with a particular colour, each district then having a different colour. A similar approach could be taken using the shape of data representations.

B. Edges

Edges are features of an environment which provide distinctive borders to districts, or act as linear divisions. Examples of edges are rivers and motorways, though the latter may serve a dual purpose, as an edge to pedestrians or as a path to motorists. Because edges serve as boundary features it makes sense to position them at intersections between districts within the abstract data space. Research performed while developing the LEADS system identified three possible methods for defining appropriate edges. These will be briefly described here.

The first method, also the selected method for LEADS due to complexity and efficiency reasons, is to identify the two nearest data items between districts. An edge can then be defined between these two points and orientated accordingly. This method is extremely primitive but also allows very fast identification of edges. The second method is to identify the hull or bounding region which completely encloses a district. This hull can then be rendered as the edge. Finally, the third method involves identifying the hull surrounding two districts then creating an edge by interpolating the points along adjoining edges of the districts. Unfortunately, the latter two methods are computationally expensive.

C. Landmarks

Landmarks are static and easily recognisable features of the environment, which can be used to give a sense of location and bearing. Within urban environments, examples of landmarks could include distinctive buildings or structures. One pre-requisite of landmarks, both within the urban environment and the data space, is that they remain relatively static in position. This is not a simple task when

considering a possibly changing data environment, so it is reasonable to assume that the position of landmarks may vary slightly from time to time.

Again, three methods were evaluated for inclusion in LEADS and will be described briefly. The first method defines the landmarks at the centroid or geometric centre of districts. While simple to implement, this method was rejected, as it does not cater for the size or density of districts. Landmarks could be easily obscured by dense regions of data, and landmarks may possibly be more meaningful if placed external to districts.

The second method places landmarks at intersections between three or more districts by simply identifying the midpoint of the closest data items between adjacent districts. These landmarks would then give an indication to clusters of districts and would probably serve as a better navigation aid. Finally, the third method is a slight variant of the second in that the landmarks are placed by triangulation between the centroids of any groups of three adjacent districts. This method produces roughly the same number of landmarks as the second method, though it should result in a more stable positioning as it relies on the average position of data within a district and not just any one data item.

D. Nodes and paths

Paths represent well-travelled routes through an environment, the key factor for the creation and maintenance of a path being that people actually use it. It is proposed by Ingram and Benford that paths should be composed of links between nodes, which represent individual objects within the visualisation. The construction of these nodes and paths will rely on two stages. The first stage will assume no initial access to the data space and will attempt to define nodes and paths as data items which are likely to be accessed frequently. This will work on the basic assumption that more frequently accessed data will possibly be items at intersections between districts and also at the centre of districts. The former assumes that these items will be seen as the most similar items between two districts. The latter assumes that the centre items will be most typical of the information content of a district.

The second stage relies on the recording of usage information within the data space, such as which nodes are accessed and in which order they are accessed. This information can then be used to gradually construct new nodes and paths, with old and disused nodes and paths possibly fading over time. Metrics such as the frequency of access of a data item could be used to identify nodes, whereas the frequency with which two data items are accessed successively could be used to define paths. Finally, the aim is to produce two forms of path, major and minor, which will indicate the popularity of various routes.

IV. Hyper-structure visualisation

As mentioned previously, hyper-structures are formed by typically large information stores with an arbitrary number and configuration of explicit relationships or links between data items. Examples of such structures are extremely common within software visualisation, in particular dependency graphs. A more immediate and recognisable example is the graph formed from document stores in hypermedia

systems where the documents and the links between them form the hyper-structure. The inherent complexity and arbitrary arrangement of such structures make visualisation an extremely difficult task. Hyper-structures representing even relatively simple data sets can quickly clutter the visualisation and rapidly deteriorate usefulness.

Several visualisation systems have been developed to offer some form of visualisation of these complex structures, the majority of which opting to use self-organising graph layout techniques as described earlier. Several of these systems are discussed here.

A. Narcissus and Hyperspace

Narcissus [Hendley95b] is an information visualisation system, developed at the University of Birmingham, for creating 3D visualisations of hyper-structures with a particular application towards software visualisation. One of the proposed applications of Narcissus is in visualising software structure by displaying the dependencies between various components within a system. Hyperspace [Wood95] is an extension of the research into the Narcissus information visualisation system and concentrates on the visualisation of hypertext document stores, specifically HTML WWW pages readable using NCSA Mosaic. The system makes use of the Mosaic WWW browser API for data collection, i.e. fetching pages from remote locations. The Mosaic browser is also used to display pages selected from within the visualisation.

Figure 3.21. Example of a hyper-structure.

Image courtesy of Dave Snowdon, Nottingham University.

The visualisations in both Hyperspace and Narcissus are created as wireframe 3D graphs with the pages represented as spherical nodes and the links as directed edges. Layout of the graph is performed using a self-organising technique, with new nodes introduced to the structure being placed randomly. Problems

with this form of visualisation are that the structure and appearance of the model produced by the graph may change greatly when a new node or link is introduced. This is compounded by the fact that visualisations of the same structure may be inconsistent between runs. This effectively destroys the user's cognitive model of the structure thus requiring frequent re-orientation as the graph is updated.

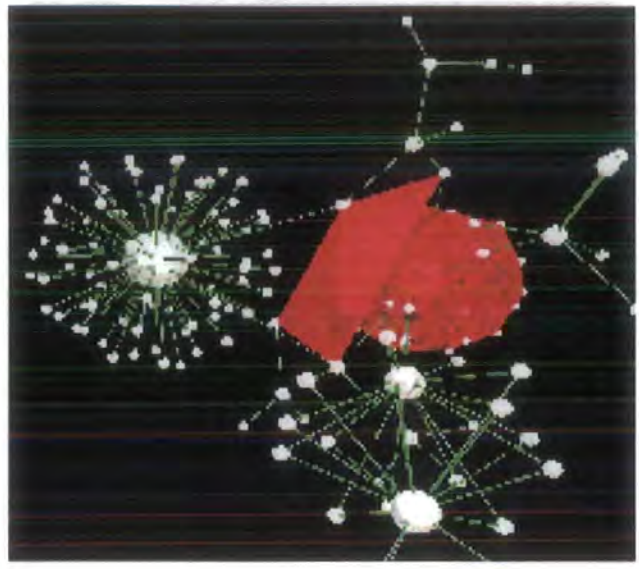


Figure 3.22. Screenshot of ray-traced output from Narcissus.

Image obtained from Birmingham University WWW presentation.

B. SHriMP Views

SHriMP (Simple Hierarchical Multi-Perspective views) [Storey95] is a tool for visualising large graphs and hyper-structures and is geared towards such graphs within software systems. SHriMP is incorporated within the Rigi reverse engineering system and aims to provide an overview of software structure allowing both detailed views while still maintaining a notion of positioning or context within the graph structure. SHriMP employs both nested graph and fisheye techniques to provide both abstraction and focusing of detail as necessary.

Nested graphs [Harel88] are a technique used to abstract information from complex graphs in an attempt to clarify their structure yet also enable the detail to be viewed when necessary. Nested graphs are based on the notion that nodes and arcs within the graph may be either atomic or composite. Composite nodes are a single node abstraction of any number of child nodes. Similarly, composite arcs may represent any number of child arcs. The use of such a technique allows graph structures, particularly hierarchies to be simplified greatly, enabling the user to select additional detail on areas of interest.

C. SemNet

SemNet [Fairchild88] is a research prototype developed to advance understanding of the complex relationships and structures of large, arbitrary knowledge bases (Figure 3.23). The aim of SemNet was

not to provide a solution for this general problem, but rather to explore the application of 3D visualisation techniques and to investigate the specifics of the problem. The main problem addressed by SemNet is how to present large knowledge bases to enable improved or more efficient comprehension. Fairchild and Poltrock hypothesise that for comprehension of a knowledge base, a user must recognise:

- the identity and meaning of individual elements;
- the relative position of elements within a hierarchical context;
- the explicit relationships between elements.

Knowledge bases represent information about relationships between objects or records. Fairchild and Poltrock believe that graphical representations offer an effective way of communicating such relationships. This, coupled with the high level of visual processing and spatial reasoning capable in the human brain, makes 3D representations a good candidate for such a task.

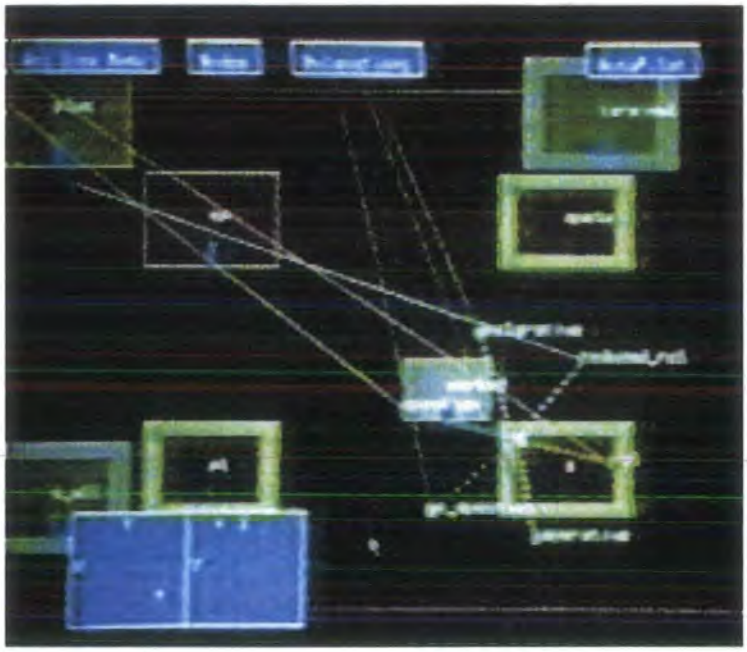


Figure 3.23. SemNet visualisation.

Image courtesy of Kim Fairchild, Institute of Systems Science, National University of Singapore.

Fairchild and Poltrock are quick to point out that simply using a graphical representation does not immediately solve the problems inherent with exploring, manipulating, understanding and modifying large knowledge bases. Very large knowledge bases simply become very large directed graphs! This highlights an important need for reducing the visual complexity of the graphical representations, a need that is also addressed in SemNet.

Graph layout and information hiding within SemNet is debated by Fairchild and Poltrock. A number of methods are described with their advantages and disadvantages noted. The authors agree that the

positioning of knowledge elements within the visualisation can greatly affect comprehension of the knowledge base structure. Layout of the visualisation in SemNet can be performed using one of two methods:

- The application which is providing the knowledge base can assign positions to the elements using semantic information which is not available to SemNet;
- Alternatively, SemNet can perform layout using a number of heuristics designed to bring highly related elements closer together while moving unrelated elements further apart. As the semantic meaning of the elements are not available to SemNet, the layout algorithm uses the number of interconnections between nodes as a measure of their relatedness. Thus, highly interconnected nodes are placed much closer together than unconnected nodes.

Information hiding in SemNet is provided both by the inherent perspective based reduction in detail due to distancing in the 3D display, and also by application of additional fish-eye techniques. The former method is apparent if proximity-based positioning has been used. Objects that are unrelated to the current user interest will be further from the viewpoint, hence reduced in detail and size. This provides a fish-eye view based purely on three-dimensional perspective. This was combined with a cluster-based fish-eye view that attempts to reduce the complexity of the graph while still maintaining the context of the overall structure. This technique assigns elements within similar areas of 3D space to clusters, with neighbouring clusters being assigned to higher level clusters. The technique is similar to the nested graphs of Harel [Harel88]. Using these clusters, only knowledge elements close to the viewpoint are displayed individually, at further distances only the clusters are visible.

D. GraphVisualizer3D

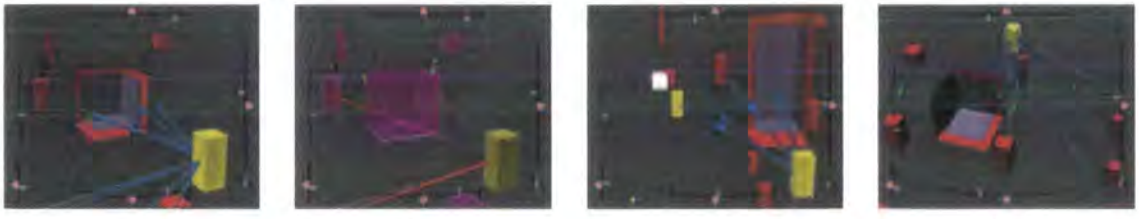
GraphVisualizer3D (GV3D) [Ware93, GV3D99] is a prototype visualisation system developed at the University of New Brunswick and shown in Figure 3.24. The system has recently been developed into a commercial application by Nvision Software Systems Inc. and renamed to NestedVision3D. The name GraphVisualiser3D or GV3D will be used to describe the system here. The aim of GV3D is to utilise 3D visualisations and virtual environment technology to provide a more readable and comprehensible graph visualisation. The project is aimed at displaying predominantly directed graphs, though the focus of the development has been to visualise software structures. The structures in this case are component relationships within object oriented C++ programs. Examples cited are software module dependencies, software usage, and inheritance relations. Ware *et al* [Ware93] describe the design issues and architecture of GV3D and include details of empirical evaluations performed on the system, which include an investigation into various interfaces and interaction techniques.

Ware *et al* investigate the use of a number of different interaction techniques both for inclusion in the GV3D project [Ware93] and also as a separate study [Ware94]. The study evaluated the occurrence of errors in a graph comprehension task in which the subjects were restricted to using a number of different interaction techniques. The results of the study highlight an important decrease in errors, thus

an increase in graph cognition, when a combination of viewpoint manipulation techniques and a stereoscopic display is used. The study also confirmed previous research indicating that the use of viewpoint manipulation alone was more effective than a stereoscopic display alone. These results highlight the importance of the virtual environment interface in producing useable and useful visualisations.

GV3D draws on a number of ideas from SemNet and is hinged upon the same concepts, however, GV3D aims to improve upon SemNet in a number of ways. One area of divergence is a shift from the predominantly automatic node placement of SemNet to the more manual oriented placement of GV3D. Ware *et al* believe that the most important heuristic for graph layout is semantic clustering: grouping nodes with respect to their semantic meaning as opposed to their position within the graph structure. The authors state that it is unreasonable to expect current layout algorithms to achieve more than a first approximation to the layout of software entities in space. The reason being that good layout is based greatly upon the high level semantics of the nodes, this cannot be formalised or automated at present. The graph layout used in GV3D is based upon elementary layout algorithms but supplemented by extensive manipulation and interaction techniques.

Similar to SemNet, GV3D also includes the notion of composite nodes and arcs [Harel88]. This technique allows reduction of the visual complexity of a graph structure by representing similar nodes and arcs as a single composite node or arc (the measure of similarity is arbitrary). These composite nodes can then be collapsed or expanded (opened or closed) as needed, depending on the focus of the viewer. GV3D copes with composite nodes by performing an identical layout on the sub-graph enclosed in the composite node. The composite node is then scaled to completely enclose the sub-graph, for this reason the node size cannot be used to display attributes. Opening or expanding a node results in the contents being displayed.



(a) **(b)** **(c)** **(d)**

Figure 3.24 (a-d). Various views of a network using GraphVisualiser3D.

Images courtesy of Glenn Franck, University of New Brunswick.

V. Information workspace

The *information workspace* or *information visualiser* is a concept demonstrator designed at the Xerox Palo Alto Research Centre (PARC) and is aimed at providing a more cost efficient workplace with respect to information access. The information visualiser was targeted to be the successor to the desktop metaphor, taking full advantage of 3D graphics and modern computing power to produce an intuitive

and effective interface. Research at PARC has investigated the cost structure of information retrieval [Card91] within the workplace and oriented the information visualiser towards maximising effective organisation of information and minimising retrieval cost.

Research into the information visualiser has lead to the development of a number of new representations and visual metaphors for information display. These new visualisations include cone trees, cam trees, perspective walls and 3D rooms, all of which were described previously in Section 3.3.1. The information visualiser is based primarily around the 3D rooms metaphor with rooms containing visualisations appropriate to the data contained within them. The information visualiser incorporates all of the features of rooms as described earlier, including the ability to display an overview of all rooms in use. The user is also permitted to manipulate the contents of the rooms from the overview, though in considerably less detail.

Navigation through the rooms is provided by the users' viewpoint being embodied in a virtual human, allowing independently controlled motion and viewpoint orientation, similar to a human walking while looking around. Interaction with objects within rooms is facilitated by either selecting an object to examine, or by giving a gesture indicating the operation to perform. Objects or the viewpoint may be moved towards one another by selecting an object of interest and moving towards it, or moving it towards you. The speed of movement is logarithmic with separation, ensuring maximal control over positioning and also that the object and viewpoint do not collide. Many items within the information visualiser respond to gestures. Gestures are performed using rapid movement of the mouse in distinct paths, for example flicking the mouse in one direction or tracing a check mark or cross. Gestures allow an easy method of performing simple commands without releasing the mouse or requiring banks of function icons.

VI. Software visualisation systems

This section describes a number of 3D visualisation systems that are all targeted predominantly at software visualisations.

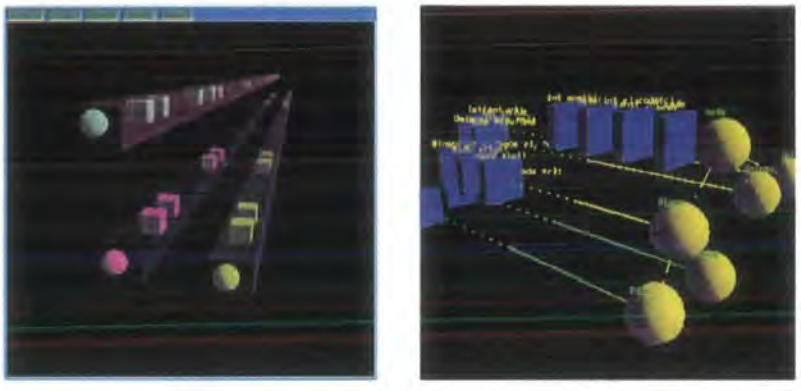
A. VOGUE

VOGUE is a visualisation system developed at Koike Labs, which provides a framework for creating 3D software visualisations. VOGUE concentrates on the concept of integrating a number of 2D views to create a more powerful 3D visualisation. This system has been applied in a number of different visualisation scenarios:

- as a software visualisation of parallel Linda programs;
- as a version control and module management visualisation;
- as a C++ class library browser.

VOGUE is used in creating a 3D visualisation that integrates version control and module management, an example of which is shown in Figure 3.25(a). The visualisation creates a two dimensional tree from the module dependencies at each revision. This set of 2D trees are then placed in series along the Z axis of the display space, with the most recent revision placed at the front. Relationships between source code modules are visible both between different versions of the same module through time, and as compilation dependencies between modules for each revision.

The 3D class library browser implemented using VOGUE, shown in Figure 3.25(b), integrates two sets of information concerned with class inheritance hierarchies. The first set concerns the actual inheritance hierarchy and depicts which classes are derived from other classes. The second set of information shows the methods belonging to particular classes. These two sets are integrated into a single 3D visualisation to provide immediate indication of which method is actually executed when called from any particular class. By shifting viewpoint orientation the programmer can obtain a display of either data set, or view both sets and any dependencies between them. This clarifies the problem by reducing the need to mentally integrate both sets of information.



(a) (b)

Figure 3.25 (a, b). VOGUE version control management and C++ class browser.

Images courtesy of Hideki Koike, Koike Labs, University of Electro-Communications, Tokyo.

B. VisuaLinda

VisuaLinda is a program visualisation system developed at Koike Labs which uses 3D visualisations to clarify the operation of parallel programs [Koike99]. VisuaLinda is built upon the VOGUE visualisation system.

VisuaLinda utilises 3D graphics to visualise the execution of parallel processes, an example of which is shown in Figure 3.26. This allows the programmers to view both the relationships between processes and a time flow of the processes simultaneously. Messages between individual processes can also be seen over time.

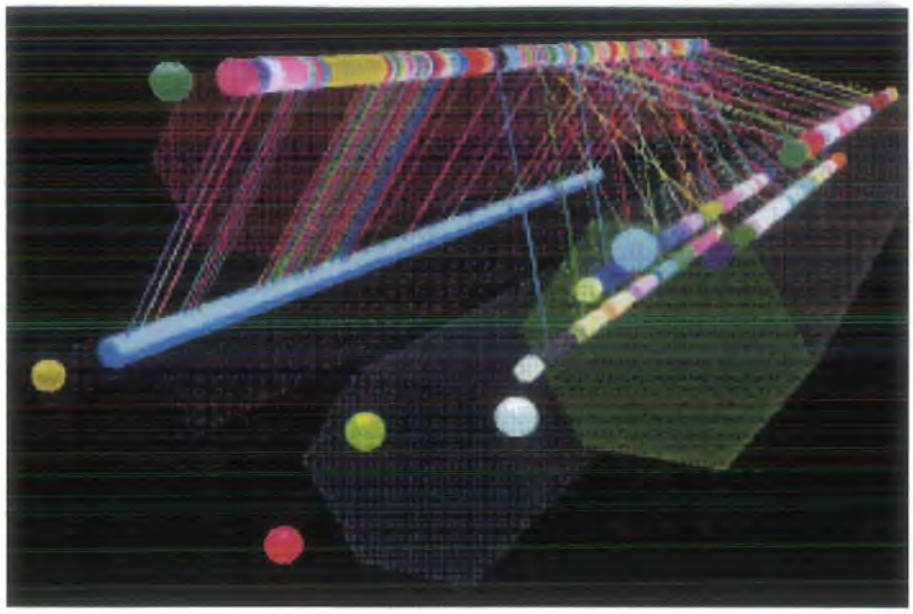


Figure 3.26. VisuaLinda implemented using VOGUE.

Image courtesy of Hideki Koike, Koike Labs, University of Electro-Communications, Tokyo.

C. Zeus

Zeus forms the latest in a series of algorithm animation systems developed at Brown University and the Digital Equipment Corporation. BALSA and BALSA-II were considered as the first major interactive software visualisation systems. This early system (BALSA) was used widely as an aid to teaching algorithm design and analysis and allows students to interactively view animations prepared by a lecturer. Functions such as stop, play, pause, speed control and direction control are provided. Zeus offers a number of enhancements over these earlier systems, in particular the use of colour, sound and, more interestingly, 3D graphics.

Brown and Najork [Brown93] describe a variety of 3D graphics techniques for algorithm animation. These techniques are highlighted by examples created using Zeus. Constructing animations within Zeus relies on the animator identifying critical areas within the algorithm code, such as data manipulation, conditional evaluation or procedure calls. These areas are then instrumented with calls to the Zeus 'interesting event' manager. During execution of the algorithm these interesting events will be reported to Zeus, which will then forward them on to any interested views. The views must be predefined and designed to respond appropriately to events in the code. During the visualisation the user is still able to alter various aspects of a view by changing its associated control panel. This allows both generic parameters, such as lighting level, and view-specific parameters to be changed.

D. PVMTrace

PVMTrace [PVMTrace99] is a visualisation system developed at the University of New Brunswick, concerned with the visualisation of parallel processes with an emphasis on debugging (Figure 3.27).

Understanding and debugging complex sequential programs is often very difficult, this problem grows drastically when trying to understand the execution of communicating parallel processes. There is a need for good tools to aid the understanding of such programs, and visualisation promises to provide an effective communication medium between the execution data and the programmer.

PVMTrace concentrates on visualising parallel programs that are running on PVM. PVM is a library which simulates a parallel computer by running processes on multiple workstations in a network. The project investigates different visualisations and representations for presenting the execution of parallel processes and messages between them. Particular attention is given to the application of three-dimensional animated visualisations.

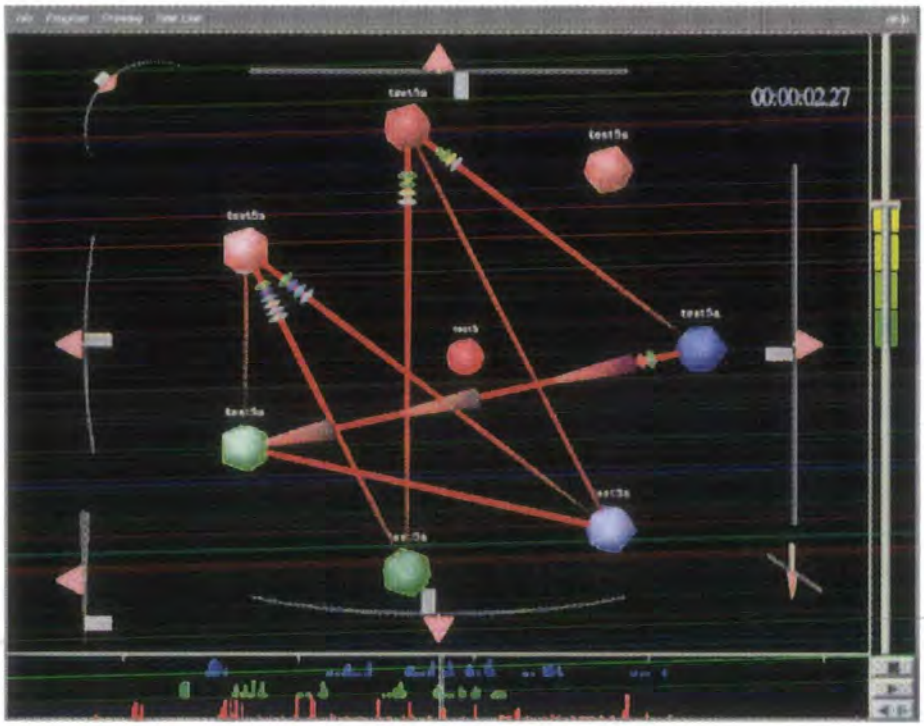


Figure 3.27. Execution of parallel processes, shown using PVMTrace.

Image courtesy of Glenn Frank, University of New Brunswick.

VII. Other relevant systems

This section describes two visualisation systems which do not fall into any of the categories of systems described above but which are deemed relevant and have provided inspiration during the course of this research.

A. FSN

FSN, pronounced 'Fusion', is a file system navigator developed by Silicon Graphics as a fun and freely available utility to demonstrate their 3D graphics hardware. One claim to fame is that it starred in the blockbuster film *Jurassic Park*. FSN makes use of the landscape metaphor to visualise the structure of a

UNIX file store and was originally intended as an investigation into information landscape navigation. The essentially hierarchical information is presented as a tree of nodes and paths on the surface of the landscape or ground plane. The root of the tree begins closest to the user with the branches receding into the distance, away from the user's viewpoint.

Each directory within the tree is represented as a pedestal. The height of this pedestal represents the combined size of the files contained within that directory. These directory pedestals are connected by paths or links, which can be traversed to manoeuvre within the file structure. Files within each directory are represented by boxes, placed in a grid, on top of the directory pedestal. Each box is adorned with a graphical image mapped onto its top surface, which is an iconic representation of the file type. The height of these file boxes represents their size, whereas the colour of the boxes represents the age of the file. The use of this information encoding coupled with the landscape metaphor means it is easy to identify prominent features such as large files or directories. These features can then act as landmarks or points of reference within the visualisation.

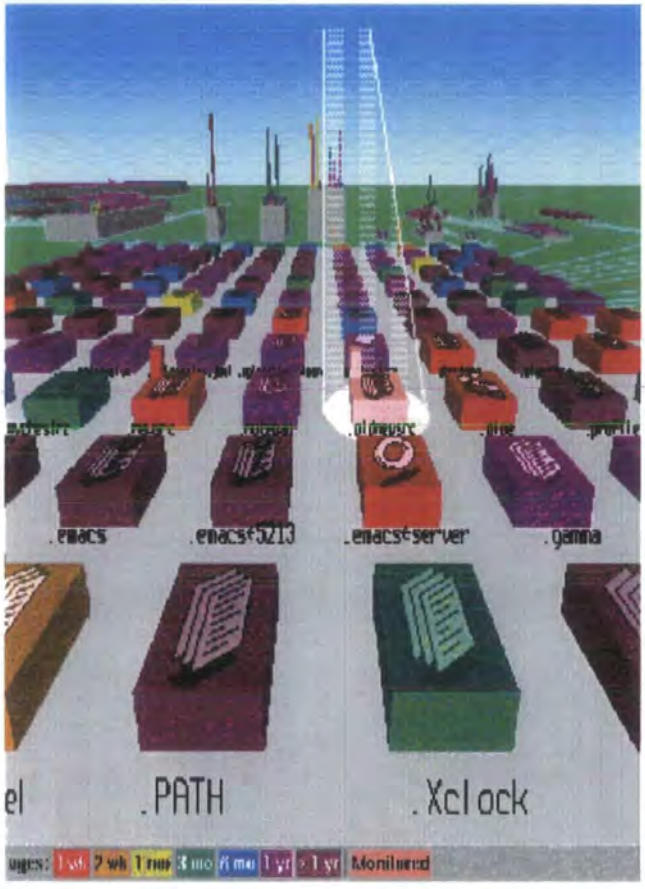


Figure 3.28. FSN visualisation of a UNIX file store.

Image courtesy of Joel Tesler, Silicon Graphics.

FSN is a fully-fledged file manager in that it not only allows the visualisation of the file structure, but also allows manipulation of the contents. Various operations may be performed on the files such as

copying or moving them. Additionally, certain files may be edited or viewed using appropriate external applications depending on the file type. Selection of files is performed by an effective 'spot-light' facility, which highlights the selected file, as shown in Figure 3.28.

B. SeeSoft and SeeDiff

SeeSoft and SeeDiff [Ball96, Burkwald98] are both entirely 2D visualisations and as such do not technically belong alongside the other visualisations described within this chapter. Their inclusion is warranted as an example of how an extremely effective and powerful visualisation can be created from the most simple of concepts. The nature of these visualisations also lends itself to some of the concepts described further in this thesis, necessitating their inclusion here. The two visualisations differ slightly in that one (SeeSoft) is purely a software visualisation system, whereas the other (SeeDiff) has more wide-ranging applications.

The basic concept of SeeSoft is to present the source code of a very large software system in its entirety allowing identification of interesting areas followed by detailed inspection. The key factors used in this visualisation are text formatting, colour coding using some metric, and proficient use of magnification and scaling. The visualisation begins in a bottom-up fashion by presenting a standard text editor showing a small section of the program source code. The indentation and formatting of the code are all preserved. Each line of the code is colour coded with some metric, in this case it is the age of each particular line. Recently modified lines are given warmer colours such as red, whereas an older unmodified section of code would be given a cooler colour such as blue.

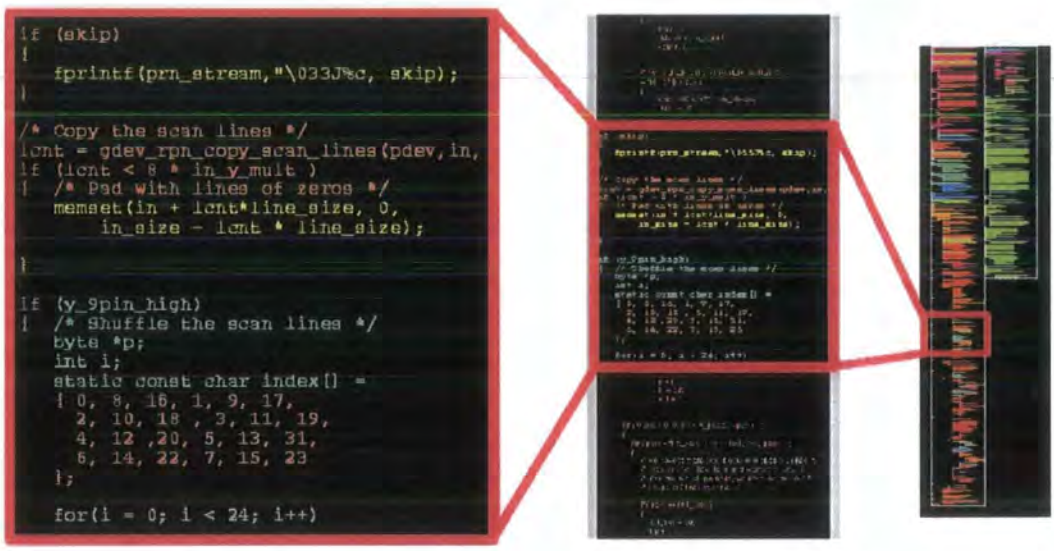


Figure 3.29. The SeeSoft concept showing decreasing magnification.

Individual images courtesy of Thomas Ball, Bell Labs.

This view of the code is then shrunk by effectively 'zooming out' from the source code while preserving both colour information and text formatting (Figure 3.29). More of the source file becomes visible, and it is possible to gain some context of the surroundings of the original section of code. By

reducing the code further so that each character of the text is represented by a single coloured pixel, we are able to see the entire source file. This final view enables us to see the structure and indentation of the code, thus identifying common sections or nesting. It also allows the immediate identification of 'hotspots' within the code which are receiving a considerable amount of attention and revision. In contrast, it is possible to identify more stable areas of code which are receiving little change during the current phase of development.

The SeeSoft application takes this final view of code and tiles it across the application window, showing all the files within the system on one screen. Figure 3.30 shows a SeeSoft visualisation of a 15,000 line software module which is comprised of 52 files. Using this overview, it is possible to immediately identify patterns within the development.

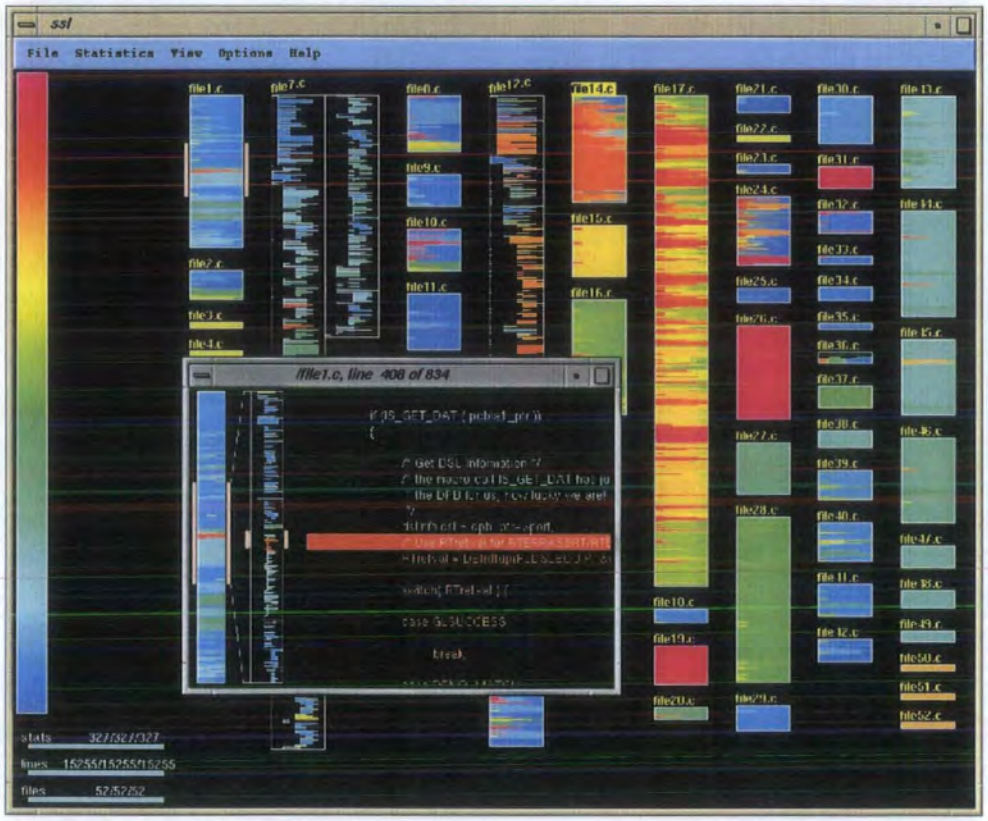


Figure 3.30. SeeSoft visualisation of a large software system.

Image courtesy of Thomas Ball, Bell Labs.

SeeDiff is a different application based on the same concept. Diff is a Unix-developed tool for comparing the differences between two similar files. Its primary use within a programming context is to identify the changes made between subsequent versions of a software system. SeeDiff uses a similar concept to SeeSoft, by providing an overview of large text files. Colour coding in this case is used to identify lines which have been added, removed, modified or which remain unchanged.

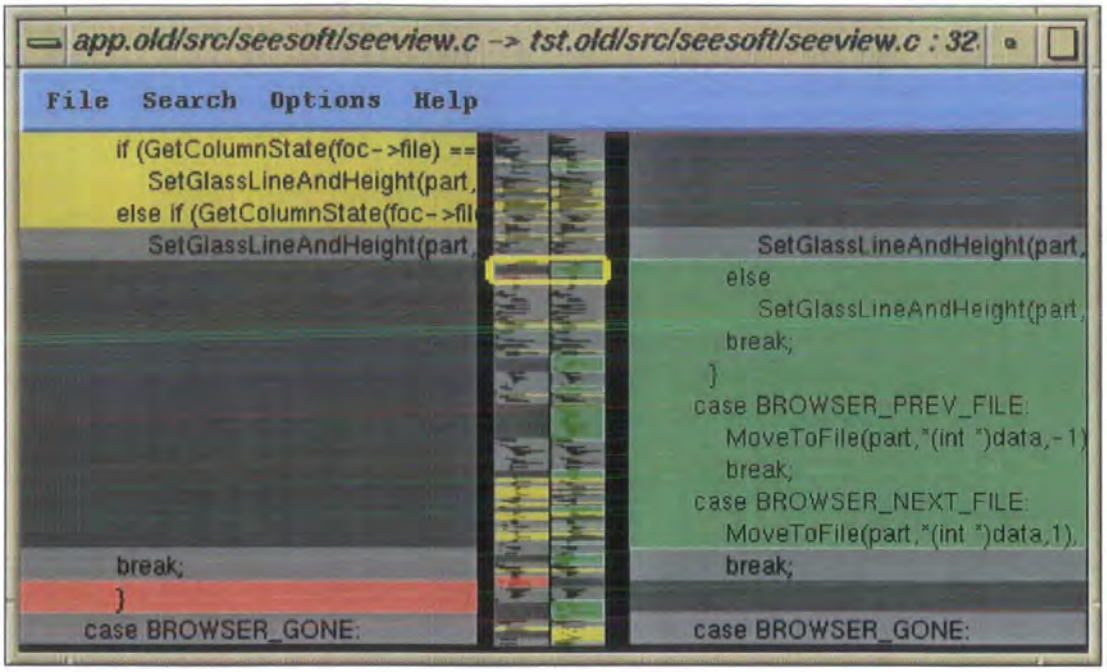


Figure 3.31. SeeDiff comparison between two source files.

Image courtesy of Thomas Ball, Bell Labs.

Figure 3.31 shows the SeeDiff application comparing two versions of a source code file. Both the detailed text of a section of the file and an overview of the entire file are available. By surveying the overview image, it is immediately apparent where the majority of changes have been made and what type of changes these are. This overview can then be used as an index for specifying which sections of code to examine in detail. In Figure 3.31, deleted lines are shown in red, inserted lines are shown in green, modified lines are shown in yellow, and any unchanged lines are shown in light grey. In order to maintain synchronisation between the files, dark grey lines are inserted to match either new or deleted lines in the opposite file.

3.3.3 Summary

This section has presented a number of information visualisation techniques and research systems. It is clear from the variety of these techniques that general information visualisation is particularly difficult. One problem of considerable importance is the generation of abstract representations which both hide the underlying complexity while offering a simple and recognisable graphical representation, i.e. a meaningful abstraction. This problem will prove a great hurdle in the acceptance and widespread use of information visualisation systems. This may be circumvented, to some extent, by producing more specialised visualisations, meaningful primarily to their intended information domain. This helps by limiting the variety of information to be visualised, though the problem still exists of creating suitable abstractions for this information.

Information visualisation techniques should play a useful role in the field of software visualisation. Software systems are not physical entities as such, they have no physical form and exist only as

information; they are information artefacts. This highlights the problem mentioned above, that of creating meaningful visual representations or abstractions. Creating abstractions for software components and their dependencies is not an easy task. The sheer size and complexity of the abstract structures created within software places them beyond the cognitive ability of a single human. The goal of software visualisation will thus be to allow users to browse and navigate these software structures freely and to identify and extract the information they require quickly. Co-operative work and interaction should also provide a major benefit to such systems, allowing teams of engineers to work on a particular software system and be aware of the actions of their colleagues.

In any case, it is clear that research into information visualisation will continue to be of great interest in the future. The need for techniques and systems developing from this research will increase as the amount of available on-line information grows increasingly rapidly. It is already hard enough to find files of relevance in one's own disk space, buried within the myriad of directories, old documents and temporary files. Without regular cleaning and restructuring this problem will worsen over time, unfortunately globally available information does not often undergo such maintenance. Sometimes when it does, this maintenance itself causes problems due to users of the information not being notified of changes. The situation will inevitably deteriorate, as the task of maintaining or regulating this information is immense and possibly infeasible. Research into techniques to help cope with this problem rather than cure it will thus be of great relevance, information visualisation is one such technique.

3.4 3D Software visualisation

The previous section provided detailed descriptions of a large number of 3D information visualisation techniques and prototype systems. Among these were a number of software visualisation systems such as Narcissus, SemNet, Rigi, GV3D, Vogue, VisuaLinda, Zeus, and PVMTrace. These systems all utilise 3D graphics in an attempt to provide an understandable picture of some aspect of a software system or computer program. In general these systems do not make full use of the 3D graphics available or advance software visualisation to any great extent. The majority simply extend and adapt already established two dimensional visualisation techniques into 3D [Brown93, Reiss94]. An example of this is highlighted by Brown and Najork [Brown93] who describe a variety of 3D graphics techniques for algorithm animation. They describe three fundamental uses of 3D:

- for providing additional information or attributes about objects which are intrinsically two dimensional;
- for uniting multiple views;
- and for providing an execution history.

These techniques take little advantage of the additional dimension provided by VR and merely offer 'window dressing' on existing techniques. They do, however, provide useful additional information

such as the execution history or extended attributes, though these could be facilitated in other ways such as animation or annotations respectively.

Similarly, 3D structural visualisation systems such as Rigi [Storey95], Narcissus and GV3D merely offer simple extensions of 2D graph techniques into 3D. While the use of 3D in these circumstances does provide a new view on the software structure, the complexity and ease of comprehension remains relatively unaffected and in some cases is worsened [Young97]. For visualisations to effectively make use of the advantages afforded by 3D graphics and virtual reality, they must be designed specifically for this environment. In some circumstances it makes sense to incorporate well established 2D techniques or to adapt and extend these techniques, however, in most cases, completely new techniques for viewing a software system within a VR environment must be developed. Until recently there has been very little research dedicated towards exploring software systems within a virtual reality environment [Knight98, Knight99, Young97, Young98a, Young98b]. This is possibly because the required hardware and software have only recently become widely available, but may also be due to the difficult nature of the problem and the lack of any research foundation to base further research on. This thesis attempts to tackle this problem from the latter point, by providing both a theoretical and practical foundation upon which further research work may be based.

There are numerous different types of software visualisation each of which concentrate on providing information or aiding with one particular area of program comprehension. While some software visualisation systems may concentrate on the dynamics of program execution, others may provide insight into the structure of a software system or the dependencies within the code. A number of different types of software visualisation are mentioned here:

- **Source code visualisations.**

  Such visualisations produce a picture from the source code text itself. By displaying indentation and line structure, and providing an overview of the source code, it is possible for experienced programmers to identify interesting sections. If this picture is overlaid with colour coding presenting some other aspect, for example changes to code or the age of code, then the display becomes even more powerful. Systems such as SeeSoft and SeeDiff provide a good example of this approach.

- **Software dynamics visualisation.**

  These systems concentrate on visualising the dynamics of program execution, for example execution traces, memory manipulation and evaluation of conditional statements. Such visualisations are useful in explaining or understanding algorithms such as sort routines. Being able to visualise the process of the algorithm, the conditional execution and the way in which the data is manipulated in memory provides an excellent insight into its operation. Systems such as Balsa, Zeus and Tango [Price93] have already provided experimental evidence of the worth of such visualisations, particularly within a teaching environment.

- **Software structure visualisation.**

  Software structure visualisations concentrate mainly on the gross features of a software system, often with a view to understanding the structure of large-scale software systems. These will concentrate on objects such as the source code files, system components, software objects, functional sections of code, etc. The aim of such a system is to produce an overall picture of the software at a high level of abstraction allowing a software engineer to browse areas of interest or identify any interesting features. Such visualisations will incorporate various information about the software system depending on the intended use of the visualisation and the needs of the engineer. One such application provides a large-scale view of a multimillion-line software system, highlighting any date fields with a view to estimating Y2K impact [Burkwald98].

- **Parallel execution modelling.**

  Systems such as Pavane and VisuaLinda provide a visualisation of message passing between parallel threads within a concurrent program. Such systems are notoriously difficult to understand and debug due to their non-sequential and often distributed nature. By providing a visualisation of the message passing it is hoped that a clearer understanding of the interaction and dependencies between these processes can be formed.

These generalisations represent only a small subset of the wide range of possible applications of software visualisation. It is obvious from these sample applications that many visualisations will be diverse and difficult to integrate into a single, general purpose visualisation of a software system. As such, it proves sensible to identify the aim and purpose of a particular visualisation prior to beginning research or development.

3.4.1 Summary

This section has presented a brief description of 3D software visualisation. A number of 3D software visualisation systems have been described earlier in Section 3.3.2.

Four types of software visualisation were described here. These were source code visualisations, software dynamics visualisations, software structure visualisations and parallel execution modelling. While far from exhaustive, these four types of software visualisation provide an indication of the variety of possible aspects of a software system that may be visualised.

The main point presented in Section 3.4 is that there is currently very little research work investigating the application of 3D graphics and VR technology to software visualisation. The majority of software visualisation systems which do utilise 3D graphics often simply extend existing 2D visualisation techniques. While such an approach does offer some benefits, there is a lack of investigation into the full possibilities afforded by virtual reality. In particular, there are currently no techniques or frameworks available to aid either the design or evaluation of 3D software visualisations. As previously stated, there is little point devising new visualisations without the ability to determine if they provide

any improvement over existing visualisations, or even without knowledge of what makes a good or bad visualisation.

3.5 Conclusions

This chapter has provided an introduction to virtual reality systems and an authoritative and comprehensive review of 3D information visualisation. 3D software visualisation has also been presented as a more specific subset of information visualisation. It is apparent that research into the application of virtual environments for exploring large information systems is receiving a great deal of interest. The current trend towards distributed collaborative working has also been addressed by many research projects investigating the issues involved for collaborative work within virtual environments. A marriage of these two research threads promises great potential for progress and for the increased acceptance of virtual reality systems within the workplace.

While there has been a great deal of research investigating the application of 3D graphics to information visualisation, software visualisation has received considerably less attention. This is most probably due to the more specialist nature of software visualisation, essentially a form of information visualisation, but is also attributable to the extremely complex visualisation problems posed by most software systems. The majority of software visualisation systems developed have concentrated on well-established 2D techniques such as directed graphs and textual metrics information [Burd96]. Some software visualisation systems have explored the use of 3D graphics but most have simply used an extension of existing 2D techniques within the 3D environment.

The main issues raised in this chapter are:

- Virtual reality is a concept that is not dependent on any particular technology or implementation.
- 3D information visualisation is a large and growing area of research which addresses very similar issues to those of 3D software visualisation.
- 3D software visualisation is currently a small and relatively new field with little existing research work upon which to build. Most systems developed simply extend existing 2D techniques into 3D, taking little advantage of the full power and flexibility of virtual reality.
- Techniques derived from the field of 3D information visualisation may be of direct use within software visualisation or may simply offer inspiration and guidance in the creation of 3D software visualisations.

Chapter 2 and Chapter 3 constitute a comprehensive review of the literature within the key research fields relevant to the work presented in this thesis. The remainder of this thesis focuses on the visualisation of software systems within a virtual reality environment, presenting the original research work of this thesis.

Chapter 4.

Visualising software in cyberspace

4.1 Introduction

Chapters 2 and 3 provided a review of the literature and research fields relevant to this research. The following chapters present the theory, results and evaluation of the subsequent research work performed.

There are a number of conclusions that can be drawn from the previous chapters. Chapter 2 highlighted the large variety of program comprehension models proposed and the lack of any consensus as to which model is most accurate. A number of researchers propose an integrated approach to program comprehension where the maintainer will utilise various program comprehension models, including top-down and bottom-up, in an as-needed manner [Chan98, Mayrhauser95]. It is clear that providing support for this integrated comprehension model will prove most productive, enabling the maintainer to utilise whichever comprehension strategy they prefer or the task demands. Chapter 2 also highlighted the fact that the majority of software visualisation systems currently utilise only a small and limited set of visualisation techniques and metaphors. Exploration of the possibilities afforded by now well-established technologies, such as 3D graphics and virtual reality, are particularly lacking.

Chapter 3 provided an authoritative review of 3D information visualisation systems, a research area which is keen to exploit the possible benefits of new technology. Information visualisation draws many parallels with software visualisation, both of which are concerned with the display of abstract information entities. Many of the techniques developed within each field should prove transferable. While 3D information visualisation has received considerable attention, work on 3D software visualisation has been slow to develop. One stumbling block is the lack of any guidelines for the design and evaluation of 3D software visualisation systems, and the absence of any research to provide insight and foundation for this new field. The research presented in this thesis attempts to address this deficiency and provide a significant first-step towards effective 3D software visualisation.

This chapter begins by identifying seven key areas of 3D software visualisation. These seven areas must each be considered when creating such a visualisation and provide a basis for describing the various qualities of a visualisation. Section 4.3 introduces the concept of visualisations and representations.

This provides a framework for discussing the various properties of a visualisation by providing a conceptual division of entities within it. This is followed by a description of the desirable properties of visualisations and representations. These properties act as both design guidelines and also a framework for evaluation. Finally, Section 4.4 presents an example of some of the concepts involved. This is provided with reference to a prototype software visualisation in which two differing representations are used. A description of how each representation addresses the desirable properties is made, providing an illustration of some of the issues and pitfalls involved.

4.2 3D Software visualisation

The field of 3D software visualisation is still very much in its infancy. Whilst software visualisation in general is reasonably well developed, it offers very little support for the issues involved in creating a software visualisation within a virtual environment. At present there are no guidelines for creating a 3D software visualisation, or even what aspects should be considered. 3D information visualisation and database visualisation has received more attention in this respect with several papers identifying some of the issues involved [Kennedy96, Wiss98a, Wiss98b, Wiss98c]. The first step of this research is to identify the main areas of 3D software visualisation which must be considered when creating such a system.

This research identifies seven key areas that must be considered when designing 3D software visualisations. These seven areas were derived from attempting to describe the aspects of 3D information visualisation systems and software visualisation systems while considering the needs and goals of software visualisation. There may be additional key areas which are identified for more specific aspects of software visualisation, though the seven areas identified here are believed to provide a core framework which is applicable to any 3D software visualisation.

The seven areas described here, are general in scope, but are of fundamental importance within 3D software visualisation. Application of these areas could also be made to the more general field of 3D information visualisation for which limited guidelines and methods of evaluation have already been defined. The seven key areas identified here are: representation; abstraction; navigation; correlation; automation; interaction and scaling. Careful consideration must be made as to how a new visualisation will address each of these points in order to create a well-balanced and usable visualisation.

The research presented in this thesis is focused primarily upon single-user software visualisations yet still provides insight and concepts for use within a multi-user environment. Multi-user software visualisations would be a natural extension of this work. Software maintainers rarely work alone, more often they work as a team. Being able to support a team-orientated approach within these visualisations would be of great value, but is beyond the scope of this research. If the visualisation was to be a shared, multi-user virtual environment then several other key areas must also be introduced in addition to the seven presented here. As this research is concerned primarily with single user visualisations these

additional areas have not been investigated fully. Such areas would include issues such as communication, co-operation, impact and accessibility.

The seven areas described here are intended as a purely qualitative framework for identifying aspects which should be considered during the design of a 3D software visualisation. While no attempt has been made to provide any form of quantitative measurement of these areas, they could possibly provide the basis for such a framework through future research. The seven areas are intended here to provide some overall structure to the design process and to direct the thoughts of the designer to areas which are believed to be important for 3D software visualisation.

The seven key areas of 3D software visualisation are now described individually.

4.2.1 Representation

This is concerned with how the various components of the software system are shown graphically and also how information about these components can be encoded into the graphical representation. This is arguably one of the most important aspects of a visualisation, and careful decisions must be made. The graphical representations used will determine the overall structure and feel of a visualisation and could make the difference between an intuitive and highly useful visualisation or a confusing and off-putting one. Design factors here will impact on all the other key areas and vice versa.

The issues of visualisations and representations (Section 4.3) are of great relevance here. Finding a suitable balance of properties within an appropriate visual representation is a particularly hard task. Section 4.3 describes the desirable properties of both visualisations and representations, illustrating the difficult compromises which must be made in order to achieve an effective visualisation.

4.2.2 Abstraction

One of the main aims of software visualisation is to abstract information away from the low-level detail, for example the source code, and present it as a more useful higher level representation. In deciding the level of abstraction to be used, we must determine what information will be presented and also in how much detail to present it. The level of abstraction implemented will greatly affect how, and for what tasks the visualisation tool itself will be used.

A visualisation that displays information at a very high level, for example showing the distribution of files within a system, will be of little use for debugging a localised problem within a particular section of code. Similarly, a visualisation displaying detailed information on a single function would be unsuitable for learning the overall structure of the software system and dependencies within it. Ideally a visualisation should support a dynamic change in the level of abstraction and amount of detail presented, dependent on the user's interests and needs.

4.2.3 Navigation

Information sources such as document collections and particularly software systems are typically very large. It follows that the visualisation of such information will also tend to be large. Additionally, virtual environments enable us to submerge the user *within* the visualisation effectively creating an information terrain or information environment around them. It is important that users can easily navigate their way through the visualisation without becoming lost or disorientated. Features such as signposts, landmarks, paths and districts all aid in creating a legible environment.

Navigation is a wide ranging term which includes the legibility of the environment but is also concerned with tools which may aid navigation such as maps, bookmarks and teleports. These are perhaps more suited to the category of *interaction.*

4.2.4 Correlation

Visualisations represent another view of a software system or information store. They present additional information on the system and, more specifically, they do not replace the information already available. It is important to be able to link the visualisations in with other forms of information, for example the source code or documentation in the case of software systems. Providing a readily understandable correlation between the visualisation and the underlying information is vital if the visualisation is to be of any use. The actual changes made to software occur at the code level so relationships between the objects in the visualisation and actual points in the source code must be clearly visible or easily accessible.

4.2.5 Automation

An important point to consider when designing information visualisations is to what extent the creation of the visualisation is automated. It is necessary to determine how much of the visualisation is generated automatically and how much control the user has over the process or the result. The goal of software visualisation is to aid the understanding of software systems. Often this understanding is best obtained by practical exercises. Allowing the user to 'build' the visualisation while investigating areas of the software system may prove more intellectually profitable than fully automating the process with the user then gaining their understanding from the completed visualisation. A similar analogy can be made to the process of making notes on the content of a technical paper or journal article. A greater understanding of the content of the paper would result from creating the notes or annotations oneself rather than being given a pre-written set of notes. Conversely, the pre-written notes are of benefit as they reduce the amount of time and effort necessary to understand the salient points of the paper.

There are many other automation issues such as the use of layout algorithms or heuristics, legibility and resilience to change. The latter of these is particularly important when considering automatically generated visualisations. When changes are made to the underlying software system, it is imperative that the new visualisation changes as little as necessary in order to reflect this. This resilience to change

is necessary in order to maintain the user's mental model of both the software and the visualisation. If the structure and layout of the visualisation were to change drastically then the user would need to relearn and explore the environment each time. This is clearly counterproductive.

4.2.6 Interaction

Creating a suitable visualisation is only one part of the problem. Complex visualisations will inevitably require some form of interaction with the user. This interaction may be no more complex than navigating the visualisation or, in contrast, performing some complex data mining using visual query techniques [Boyle93]. Consideration must be made as to how the user will interact with the visualisation and how they can manipulate its contents or build upon it with higher-level semantic information.

The subject of interaction also covers any virtual tools that may be used in the environment for various purposes. For example, a map, a virtual display screen (for displaying documentation, source code listings, etc.), or editing tools for customising or annotating the visualisation.

Aspects such as the control technique used to move around the environment or to interact with objects are easily overlooked when creating a visualisation or virtual environment, yet they exist at such a fundamental level that the success or failure of the system may well depend on them. A poor control system can lead to a steep learning curve in order to use the visualisation. Coupled with the more complex and unfamiliar 3D environment this can often create a very off-putting and even unusable visualisation. Control systems should be intuitive yet powerful, catering for both novice and expert users.

4.2.7 Scaling

Software systems vary in size and complexity, from relatively simple programs to large multimillion line real-time systems. It is important to consider how a software visualisation will be able to adapt to the differing scale and complexity of software systems. This ties in heavily with the issue of abstraction mentioned previously. The level of abstraction used dictates what level of detail is shown in the code. Smaller systems will inevitably require a lower level of abstraction as the amount of information gleaned from high-level overviews will be of little use. However, in larger systems the use of high-level overviews will be more of a necessity in order to gain some notion of context and structure.

4.3 Visualisations and representations

In order to create or evaluate a 3D visualisation we must first identify desirable properties which are important for a visualisation to be effective. In order to derive these desirable properties it is necessary to redefine the concept of a visualisation into two distinct levels. These levels have been termed 'visualisations' and 'representations'. The properties and goals of each division vary sufficiently to warrant such a distinction. These two terms are defined as follows:

- **Representation:**
  A graphical (and other media) depiction of a single component.

- **Visualisation:**
  A collection or configuration of individual representations (and other information) which comprise a higher level component.

Effectively, representations are the graphical symbols used in a visualisation to depict each of its sub-components. For example, in a typical graph such as a call-graph or control-flow graph, the graph itself is the visualisation while the nodes and arcs are the representations. These terms are, however, interchangeable and they depend greatly on the level of abstraction and amount of detail being presented. A graphical object can be both a representation within one context and a visualisation within another. For example, if a node in a function call-graph provided further information on the structure or qualities of the function it represented. In the context of the software system (i.e. the graph) it is a representation, whereas in the context of the function (i.e. the node) it is a visualisation of further information on that function. In such cases we must consider carefully the structure and properties of the object both as a visualisation and as a representation and also in the transition between these distinctions.

Note that the term 'visualisation' has merely been redefined to provide a more specific meaning and to include the concept of a representation. Often the term 'software visualisation' is used to describe an entire visualisation system which incorporates both a visualisation and other ancillary information or views. This thesis will also use this interpretation of the term 'software visualisation' and will reserve the terms visualisation and representation to refer to the actual graphical aspect of a software visualisation system.

The desirable properties of both visualisations and representations are now discussed. These properties are intended both as guidelines for the design of a new 3D visualisation and also for use within a framework for evaluating an existing visualisation. These desirable properties were derived from a series of investigations, by creating simple visualisations and identifying the rational behind the design issues and decisions involved. The desirable properties identified here do not represent an exhaustive list and there will no doubt be many other desirable properties which are relevant to more specific software visualisations. The properties described here represent a core set which will be applicable to the majority of software visualisations and which demonstrate the different nature of visualisations and representations.

Visualisations and representations have been defined as two distinct but highly related concepts. Each has different properties and goals, many of which are mutually exclusive. In order to create a well-balanced visualisation it is necessary to achieve an effective compromise between these conflicting properties. It is also useful to identify the critical properties within a particular visualisation or representation. While all desirable properties are indeed relevant at all times, the presence of conflict between certain properties necessitates a bias towards one particular property. This bias will be

dependent largely on the nature of the visualisation or representation and should be identified where possible.

4.3.1 Desirable properties of a representation

As previously defined, the term *representation* refers to the depiction of a single component, typically within a group of such components which constitute a visualisation. The majority of visualisations will consist of a number (often a large number) of representations or groups of representations. For this reason, the main purpose of a representation is to present a component as an individual item, hence something must set it apart and show it as unique within a sea of other similar representations. This is only one desirable property of a representation, a more comprehensive list of other such properties follows. Several of these properties are mutually exclusive thus a good representation will achieve a suitable compromise between them. It should be noted that this list of properties is by no means exhaustive. The properties described here are illustrative of the core aspects of a representation although additional properties may be identified to cater for more specific aspects, for example the issues involved with multi-user, shared software visualisations.

- **Individuality**

  Representations of different components should appear differently. Also, representations of identical components, which are displayed in the same context, should appear identical. For example, in a standard function call-graph, it is typically a combination of the textual node label and node position which make a representation unique.

- **Distinctive appearance**

  Differing representations should appear as contrasting as possible. Representations should be easily recognisable as being either identical or dissimilar, even when in a large visualisation. This property contradicts with that of *low visual complexity*. To create a distinctive appearance among many representations you must increase the visual complexity of the representation.

- **High information content**

  Representations should provide as much information as possible about the component. The problem here is that as the information content is increased, inevitably the visual complexity will also increase. A high information content can also be either supportive or detrimental to providing a distinctive appearance. Components with widely varying properties will tend towards being distinctive purely from the encoding of these properties in the representation. However, for components with similar properties then the implicit higher visual complexity and similar features can often overpower the distinctive features of the representation.

- **Low visual complexity**

  Representations should not be visually over-complicated. This is beneficial both to the performance of the visualisation system and also to the user's comprehension of the information encoded in the representation. Additionally, to achieve a distinctive appearance the representation should maintain

as little visual complexity as possible. There must be a trade-off as to whether we can expect the user to distinguish between a small number of complex representations and a larger number of simple representations.

- **Scalability of visual complexity and information content**

  Mechanisms would be desirable for reducing or increasing the amount of information presented or the visual complexity of representations, depending on the context in which they are used. This would allow both detailed displays of a small number of components, giving maximum information, as well as overviews of large component collections that would display less information about the individual components.

- **Flexibility for integration into visualisations**

  This is a very important issue which affects both the representations and the visualisations in which they are used. Benedikt [Benedikt91] proposed that representations or objects contained both intrinsic and extrinsic dimensions that could be used for encoding information. Extrinsic dimensions include properties such as position and motion. Intrinsic dimensions include properties such as size, shape, colour and angular velocity. As a representation makes use of more intrinsic dimensions it becomes less flexible for integration into a visualisation. Using up resources such as colour, shape, and particularly size to encode information reduces the scope for providing information within the visualisation.

  For example, colour may be desirable for differentiating between different classes or groups of components within a visualisation. However, if the representations for the components have already been assigned colours (as an intrinsic dimension) to represent other information then colour cannot easily be used by the visualisation. The size of representations is also very important. Irregular or variable sized representations will be harder to position, group and view within the visualisation. For example, if a representation uses its size to encode some information and the visualisation uses the positional depth of the representations to encode other information. Confusion will occur when differentiating between a component being near or big, and far or small.

- **Suitability for automation**

  Another important aspect of any visualisation or representation is it's ability to be automatically generated relatively easily. The majority of useful visualisations will be generated by an automatic process with some degree of user intervention. Features such as irregular shapes or variable sized representations can often hinder or complicate this process. Representations should therefore be designed with automation in mind.

4.3.2 Desirable properties of a visualisation

As previously defined, the term *visualisation* refers to the depiction of a collection or configuration of individual representations, and other information, which comprise a higher level component. The majority of visualisations will often consist of a large number of representations. The purpose of a

visualisation is to present these representations in an orderly and structured sense, providing supplemental information on categories, collections and trends between these representations.

The following list highlights some important qualities that must be considered when designing a visualisation. As with the desirable properties of representations, many of these qualities are mutually exclusive and compromise must be made.

- **Simple navigation with minimum disorientation**

  Visualisations should be designed with the user in mind. As the user will be 'submerged' within the data, it will be necessary to structure the visualisation and add features to aid them in navigating through the visualisation. The goal here is for the user to become familiar with the information terrain and to reduce the chance of becoming lost. Techniques developed and highlighted in the LEADS project (Legibility Enhancement for Abstract Data Spaces) [Ingram95b] may be applicable here. These include the generation of features such as landmarks, districts and edges that have been proven to aid navigation through urban environments [Lynch60].

- **High information content**

  Visualisations should present as much information as possible without overwhelming the user. Again, there must be a trade-off between high information content and a low visual complexity. Both are desirable yet contradictory.

- **Low visual complexity, should be well structured**

  The structural complexity of a visualisation will undoubtedly be dependent on the complexity of the information presented, however, effort should be made to reduce the visual complexity of the visualisations. A well-structured data terrain should also result in a more understandable layout and easier navigation within the visualisation.

- **Varying levels of detail**

  The level of detail, information content and type of information presented should vary to cater for the user's interests. For example, when a user first enters the visualisation their first expectation will be to see the visualised system in its entirety. Once they have familiarised themselves with the overall structure they will then begin investigating areas of interest in detail. The visualisation should support this change in interest and provide increasing detail and information as the user moves towards a component or expresses an interest in a component.

- **Resilience to change**

  Small additions or changes to the information content of the visualisation or shifts in the user's interests should not result in major differences in the visualisation. Major changes such as a full repositioning of representations will result in the user becoming disorientated and having to re-learn the structure of the environment.

- **Effective use of visual metaphors**

  Metaphors introduce familiar concepts to the user of the visualisations and provide a good starting point for gaining an understanding of the visualisation. The emphasis here is on the *effective* use of metaphor within the visualisation, this does not mean that metaphor use is always beneficial. Strong metaphors can often produce more problems than benefits. Using a strong real-world metaphor will constrain the visualisation to displaying certain information in a particular way and may be over-restrictive for adding additional features. Often the information to be visualised will not 'fit' into the metaphor chosen, leading to the option of excluding that particular information or violating the metaphor in order to include it. The latter of these choices is undesirable, breaking a strong metaphor will often only confuse the user and lead to a doubt of the integrity of other metaphors in the visualisation. This clearly negates many of the benefits provided by the use of metaphor.

  A further issue with the use of strong metaphors is that they require some level of domain knowledge for the metaphor used. For example, a geographical based metaphor may require some implicit understanding of geography. A selected metaphor should be representative of widely familiar concepts and should be mindful of the level of domain knowledge required.

- **Approachable user interface**

  The user interface to the visualisations should be flexible enough to provide intuitive navigation and control, yet should not discourage the user or introduce any unnecessary overheads. For example, while fully immersive hardware may be of great advantage in using the visualisations, it would prove more practical to use standard hardware and interfaces, i.e. a computer monitor and suitable input device such as a mouse.

  The software interface should take advantage of well-established user interface techniques and strong, well-developed UI metaphors. For example, the use of a button symbol or object to indicate something which can be pressed. Simply changing the appearance of the mouse pointer over objects that may be interacted with is immensely useful, a technique used to great advantage in web browsers. Users will often scan a page using the mouse pointer, searching for links to other pages or information. Coupling this feature with a status bar or pop-up labels giving a description of the interaction or link provides a very powerful mechanism for exploration.

- **Integration with other information sources**

  Visualisations provide a different viewpoint on the information they are presenting, in most cases they cannot entirely replace that information. It is desirable to be able to correlate between the visualisations and the original information, or other views on it. For example, visualisations of a software module structure could be linked to the actual source code of that module. Providing this integration realises very powerful exploration and querying techniques, allowing the user to selectively use the view that provides them with the most useful or intuitive information at any point.

- **Effective use of interaction**

  Visualisations can benefit greatly by allowing the users to interact with them in various ways. This provides mechanisms for gaining more information and also helps maintain interest. Interaction goes further than simply providing 'click here' links within the visualisation, it extends to include the various control methods, navigation techniques and movement modes within the visualisation.

- **Suitability for automation**

  As with the representations, a good level of automation is required in order to make the visualisations of any practical worth. Many of the visualisations described later in this thesis have been created by hand but have all been designed with automation in mind. The process of their creation has included repetitive procedures using regular sized objects and placing. Everything in the visualisations lends itself towards a completely automatic generation. As full automation is often the end goal of many prototype systems, it is obvious that features of the visualisation should be designed with this in mind.

4.4 An example of the concepts involved

The previous discussion of the desirable properties of both visualisations and representations led to the identification of many dependencies and conflicts between these properties. A simple example of how these desirable properties apply to a 3D visualisation is now given to illustrate some of the issues involved. The example visualisation used here is the CallStax visualisation, the exact details of which are discussed in Chapter 5. This section concentrates on the selection of suitable representations for use within the CallStax visualisation. This section makes no attempt to describe the reasoning behind the CallStax visualisation or the advantages it offers over the standard 2D graph, which, in this simple example, is obviously more intuitive to understand. Full details and evaluation of CallStax are made in Chapters 5 and 6.

CallStax is intended as a 3D equivalent of the familiar 2D directed graph, incorporating many benefits for use within a 3D visualisation. Essentially the paths within a directed graph are represented as vertical stacks with each block in a stack representing a node of the graph. The paths through the graph flow vertically from the bottom of the stack to the top. Identical blocks within the visualisation represent the same node in the 2D graph. Figure 4.1 shows a CallStax visualisation of a simple directed graph. In this case all of the dark blue blocks represent a single node in the graph, all of the light blue blocks represent another single node, and so on. A correlation can be made with the original 2D directed graph shown in Figure 4.2. Node colouring is maintained between these two figures to allow comparison, with the exception of the grey nodes. The pyramidal object visible in Figure 4.1 represents the recursive call within the function 'qsort'.

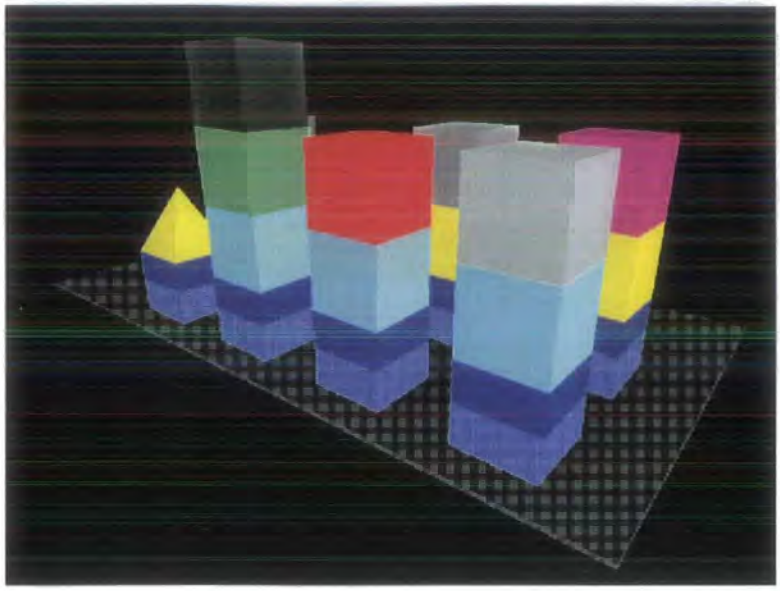


Figure 4.1. CallStax visualisation showing a simple directed graph.

The representations here (the blocks) use colour to provide both individuality and distinctive appearance. It becomes apparent for even a moderately larger graph that the range of discretely identifiable colours available is woefully inadequate. A better representation must be found in order for this visualisation to be useful. The example given here will concentrate on a representation which aims to satisfy the desirable properties important to the CallStax visualisation. Evaluation of the CallStax visualisation is not made at this stage but is provided later in Chapter 6. This example will concentrate on providing a description of the issues involved.

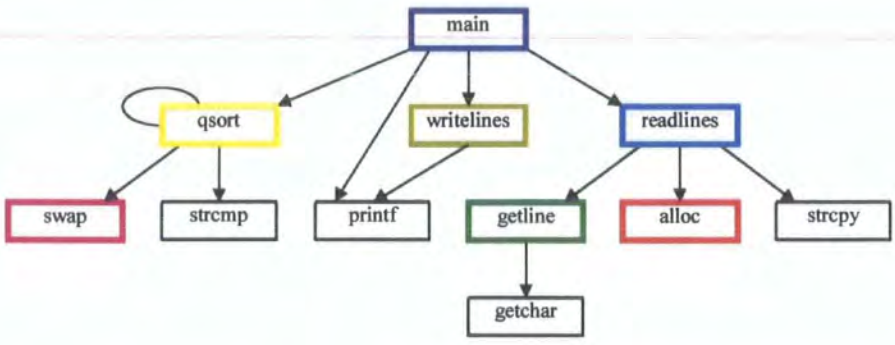


Figure 4.2. Simple 2D directed graph matching the CallStax of Figure 4.1.

Before discussing the alternative representation, mention should be made as to how the current representation (a coloured box) addresses the desirable properties, and what are the critical properties of a representation for use within CallStax. CallStax relies heavily on the identification of identical representations within the visualisation. CallStax also places a strong reliance on the dimensions of its representations with the necessity that they should be easily identifiable from any perspective. These demands are reflected in the desirable properties deemed critical for a representation used within

CallStax. These three critical properties are individuality, distinctive appearance, and flexibility for integration into the visualisation.

- **Individuality**

  Different components must appear different and identical components must appear identical. This is the fundamental basis of the CallStax visualisation and is critically important. In small scale CallStax visualisations this is achieved using box colour. There are essentially an infinite number of different colours that can be produced by modern graphics systems (16.9 million) so, in theory, individuality is not a problem. However, in practice the human eye can only discern between a relatively small number of these colours.

- **Distinctive appearance**

  Different representations must appear as contrasting as possible. Again, in this small scale CallStax example, this is achieved by selecting contrasting colours. However, the supply of contrasting colours is relatively low and would be easily exhausted when visualising any medium-size graph.

- **High information content**

  Although not necessary in the case of CallStax, any form of information content is desirable. The only information required of CallStax is provided by the visualisation – i.e. the structure of the graph. The representations used here provide no information content.

- **Low visual complexity**

  This is easily achieved by the simple nature of the coloured boxes. They have an absolute minimum visual complexity.

- **Scalability of visual complexity and information content**

  The coloured boxes provide no scaleable complexity or information content, however they could be used as a low-detail, low-information representation within a sequence of other representations. Such an arrangement would provide a low-detail representation when viewing from a distance, thus reducing visual complexity and display crowding. This would then be replaced with a higher-detail representation as the user approaches, thus providing greater information content.

- **Flexibility for integration into visualisations**

  Representations must be flexible for integration into a visualisation. The demands of the CallStax visualisation are considerably strict in this respect. CallStax relies on a fixed size representation regardless of perspective. The blocks used perform well in this respect.

- **Suitability for automation**

  Again, this suitability is easily achieved by this simple representation. The exact dimensions of the representation are fixed, there is no information to encode, and they only require a unique colour to be assigned to them.

To summarise, the coloured boxes perform well as representations in the properties of flexibility for integration, low visual complexity and suitability for automation. They perform poorly (in a large-scale sense) in the properties of individuality, distinctive appearance and high information content. Two of the properties they perform poorly in are critical to the CallStax visualisation, namely individuality and distinctive appearance.

These desirable properties, and in particular the critical properties, were considered for providing an alternative representation for use within CallStax. One possible solution is shown in Figure 4.3. A description of how the desirable properties relate to this particular representation will also be made.

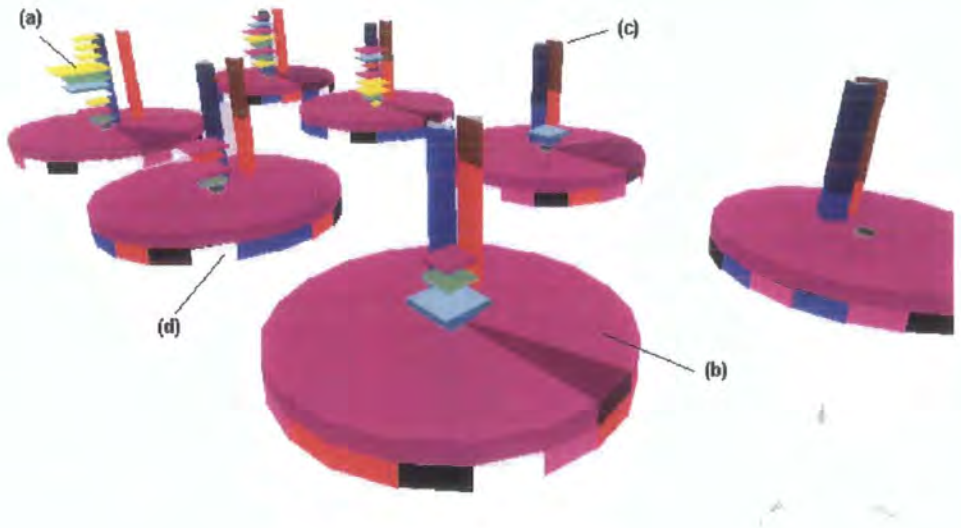


Figure 4.3. Possible representations for use in CallStax.

Each representation shown in Figure 4.3 corresponds to a single function within a small software program. It was decided to incorporate some aspects of each function into the representation in order to help provide a distinctive appearance while simultaneously increasing the information content. A circular base was selected to help maintain perspective with the information provided in the representation, and to satisfy the visualisation prerequisite of a fixed size representation. Information encoded into the visualisation includes:

- An indication of the function structure and control flow – (feature A in Figure 4.3).
- A pie-chart showing a breakdown of the number of lines of code, comment and blank lines within the function - (feature B).
- Bar charts showing two complexity metrics for the function. These were also selected to provide a fixed height to the representation and to help maintain depth and perspective - (feature C).
- A 4-segment repeated colour code surrounding the lower portion of the base. This provides a feature which is guaranteed to be unique between representations of very similar functions. Assignment of this colour code is arbitrary – (feature D).

By comparing the new representations with reference to the same set of desirable properties it can be seen that they provide a substantial improvement in most areas.

- **Individuality**

  The individuality is much improved with each representation being guaranteed a unique appearance due to the 4-segment colour code on the base. This is augmented by the inclusion of structural and metric information providing each representation with an appearance that matches its corresponding function.

- **Distinctive appearance**

  The distinctive appearance remains good when viewing a small collection of these representations. The high degree of individuality aids this somewhat. The distinctive appearance is also predominantly dependant on the extent to which the individual functions differ in structure, size and complexity. This could be compromised in larger software systems.

- **High information content**

  Information content is much improved with the representation providing insight into the structure, control flow and complexity of the function.

- **Low visual complexity**

  The representation is visually complex as a result of the need for individuality, distinctive appearance and increased information content.

- **Scalability of visual complexity and information content**

  The new representation provides no scaleable complexity or information content, however it could be used as a high-detail, high-information representation within a sequence of other representations.

- **Flexibility for integration into visualisations**

  Flexibility for integration remains high with the representation maintaining a fixed size and containing reference features that provide good depth perspective. The colour scheme of the pie-chart base remains unallocated so is available for use by the visualisation (it is the relative shading of pie sections that is important, not the overall colour).

- **Suitability for automation**

  Suitability for automation remains good. The representation contains fixed dimensions and allocation of features. Information encoded in the representation is well defined.

It can be seen from the above analysis that the new representations improve upon this set of desirable properties in almost all cases. The only exception is the increased visual complexity, which suffers as a result of the strong need for individuality and distinctive appearance. On first impression this appears to be a successful representation for use within CallStax, however, including the representation as shown in Figure 4.4 provides evidence to the contrary. Figure 4.4 shows the same CallStax visualisation as

seen in Figure 4.1 but with the alternative representation. Library functions are shown as simple grey diamonds.

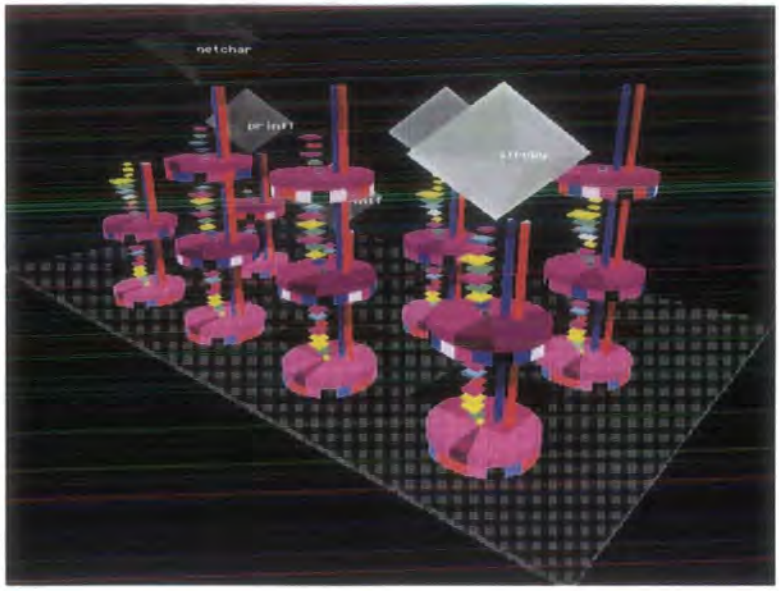


Figure 4.4. CallStax visualisation using a different representation.

It is apparent from Figure 4.4 that the increased visual complexity of the representation compromises its distinctive appearance. The predominantly transparent nature of the representation was initially thought of as beneficial by reducing background occlusion and allowing more of the visualisation to be seen from any one point. This feature compounds the effects of increased visual complexity making the representations hard to separate visually.

Although the new representation in its current form is of marginal use, modification to address the problem of high visual complexity could accentuate the beneficial features of the representation. By studying Figure 4.4 closely it becomes apparent that there is a technique to 'reading' the visualisation and it is possible to identify similar and dissimilar representations quickly, for example by concentrating on the base of each visualisation. However, such techniques would only be effective in the case of a single scene in which items for comparison are both visible. The high visual complexity of the representation would render comparison between two objects, that are not simultaneously visible, very difficult indeed. By combining the original simplicity of the coloured cubes with the added benefits of this new representation, for example by colour-encoding the base of the representation, its usefulness can be greatly improved.

This example has shown that the desirable properties are particularly important when designing either a visualisation or representation. It has also shown that effective compromise between the properties is essential for a visualisation or representation to be successful. Perhaps the most important point raised is that even if most of the desirable properties have been addressed, any one property can lead to a representation being successful or unsuccessful depending on the context in which it is used. This

highlights the need for a strong correlation between the development of the representations and the visualisations, and that identification of the key factors and appropriate compromises must be made.

4.5 Summary

This chapter has presented the main thrust of the research work within this thesis. Seven key areas of software visualisation have been identified, providing a framework of important issues which must be considered when creating a 3D software visualisation. The notion of visualisations and representations has also been introduced. This distinction provides a mechanism for facilitating meaningful discussion of 3D software visualisation systems, and indeed many other forms of 3D visualisation. Further to this, the division of visualisations and representations enables us to evaluate the desirable properties of each and to understand the issues and conflicts between the two. This is particularly evident when considering an object within a visualisation from two different perspectives, both as a representation and a visualisation.

This chapter has identified a set of desirable properties for visualisations and representations. These properties are useful both as a reference during the design of a 3D software visualisation, but also as a method of evaluation for existing visualisations. This evaluation is described further in Chapter 6. Finally, an example visualisation utilising two very different representations has been presented, concluding with a discussion of how each representation addresses the various desirable properties.

The following chapter presents the results of practical application of this research work. These results take the form of concept demonstrators, each used to explore the issues involved within 3D software visualisation. Chapter 6 provides an evaluation of these concept demonstrators, using the desirable properties as a framework for assessing both the visualisations themselves and the representations they contain.

Chapter 5.

3D software visualisation systems

5.1 Introduction

Chapter 4 introduced the ideas which form the basis of this research work, in particular the notion of visualisations, representations and the desirable properties associated with each. This chapter will introduce a number of prototype visualisations and concept demonstrators, developed throughout this research in order to explore the various possibilities afforded by 3D software visualisation. Each of the visualisations described here, except for Zebedee, was designed with constant referral to the desirable properties. Due to the conflicting nature of many of these properties it is interesting to see how each visualisation addresses these issues and what bias, if any, is selected towards one particular property. This assessment is made in the following chapter. Chapter 6 provides an evaluation of the prototype systems presented here and, in particular, uses the desirable properties as the basis for an evaluation framework. This framework is then used to evaluate, in detail, one of the visualisations described here.

While most of the research presented within this chapter is applicable in general to both 3D information visualisation and all forms of 3D software visualisation, the actual goals of this research work are more specific. This research is concerned with the structural visualisation of software systems. Such visualisations provide a gross overview of a software system, giving a variety of information on the many constituent components. A software engineer is able to navigate within the software, identifying important or interesting features merely by surveying his or her surroundings. Once areas of interest are identified the software engineer is then able to move in for a closer look and is provided with increased levels of detail. Each of the concept visualisations described here was developed with the primary goal of supporting this sort of structural view.

This chapter concentrates purely on a description of the concept demonstrators. Evaluation of the effectiveness of these visualisations, and the issues involved in their creation are described in Chapter 6. Chapter 6 also provides a detailed evaluation of one of these visualisations, including its associated representations, against the desirable properties identified in Chapter 4.

5.2 Zebedee

Zebedee represents the first prototype system developed from this research. The goal was to recreate an existing two-dimensional software visualisation technique within a 3D environment and to explore any benefits, limitations and issues involved. Zebedee allowed an exploration of the issues involved with

3D software visualisation. Such issues include visualisation of complex data and structure, automatic generation of visualisations and navigation within a 3D environment. Zebedee is an extension of the hyper-structure visualisations described in Chapter 3.

Zebedee provides a 3D visualisation of the directed graph formed from function call relationships, a common and well used 2D software visualisation often referred to as a call-graph. Zebedee derives its name from the force-directed placement (FDP) algorithm employed for layout. The FDP algorithm effectively simulates a physical system in which nodes and links exert forces, moulding the graph in an attempt to form a stable system. This stable system often results in an aesthetically pleasing graph layout, the algorithm being particularly suited to the additional freedom available within 3D space.

Zebedee produces a standard graph within the 3D environment, consisting of nodes that represent functions within a software system and links corresponding to the calls between these functions. The layout of the nodes is completely arbitrary and is determined by the FDP algorithm [Fruchterman91]. This algorithm models the nodes as free-moving spheres that possess a repulsive charge towards other nodes. Adjacent nodes will repel each other with a force inversely proportional to their separation, so the closer two nodes are together then the stronger the force will be trying to separate them further. Each link acts as a spring connecting two nodes. These have the effect of pushing or pulling nodes depending on the extension of the spring. Each spring has a predetermined natural length. If two linked nodes are further apart than this natural length then the spring is stretched and it will exert a force, proportional to its extension, trying to pull the nodes together. If, however, the nodes are closer together than the natural length then the spring will be compressed and have the effect of pushing the two nodes apart (in addition to the repulsive charge between the nodes). Graph layout is performed by providing the nodes with some arbitrary initial placement, followed by a number of iterations of the physical model. The graph will eventually settle into a minimum energy state (provided damping is present to inhibit oscillations). This final layout is generally very acceptable with nodes positioned at suitable distances, though there are occasional problems that become particularly evident with complex graph structures.

Figure 5.1 shows an example of a small function call-graph displayed using the Zebedee FDP layout algorithm. The ability to manoeuvre within the 3D environment and to examine it from different angles contributes greatly to comprehension of the structure. The simple 2D screenshots displayed here are not capable of imparting a true sense of depth resulting in portions of the graph appearing closely packed, whereas in the reality of the 3D visualisation they are well spaced.

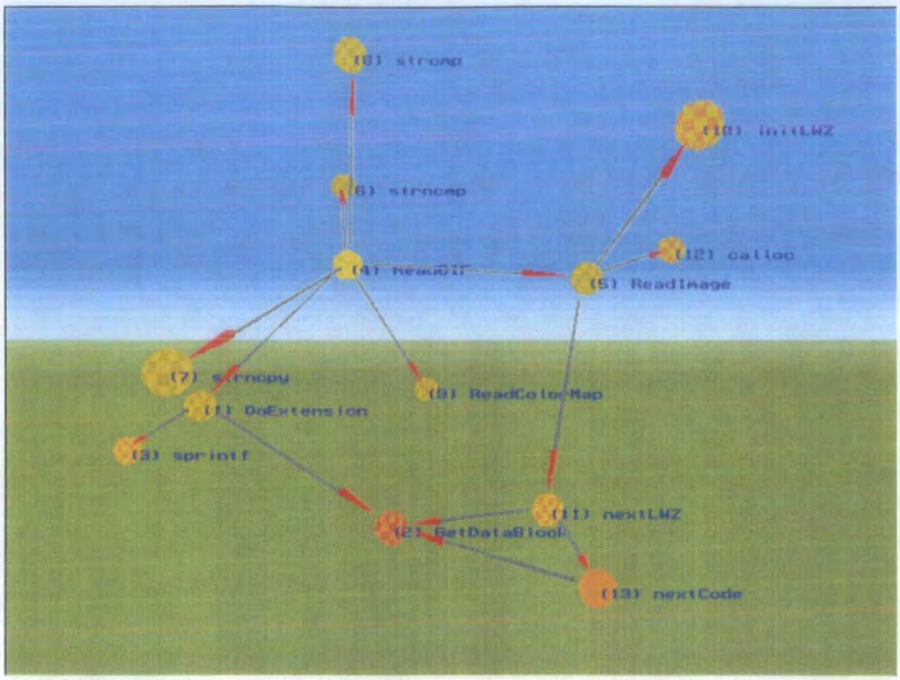


Figure 5.1. Zebedee graph layout.

Figure 5.2 shows a basic user interface for Zebedee. This includes two cameras in addition to the main view window. These cameras enable tracking of the user's position within the visualisation and help provide a constant frame of reference for location and orientation. Other features of the interface allow the user to hide node labels to reduce visual clutter, collapse or expand levels of the graph hierarchy, or traverse links within the graph automatically by simply selecting a destination node.

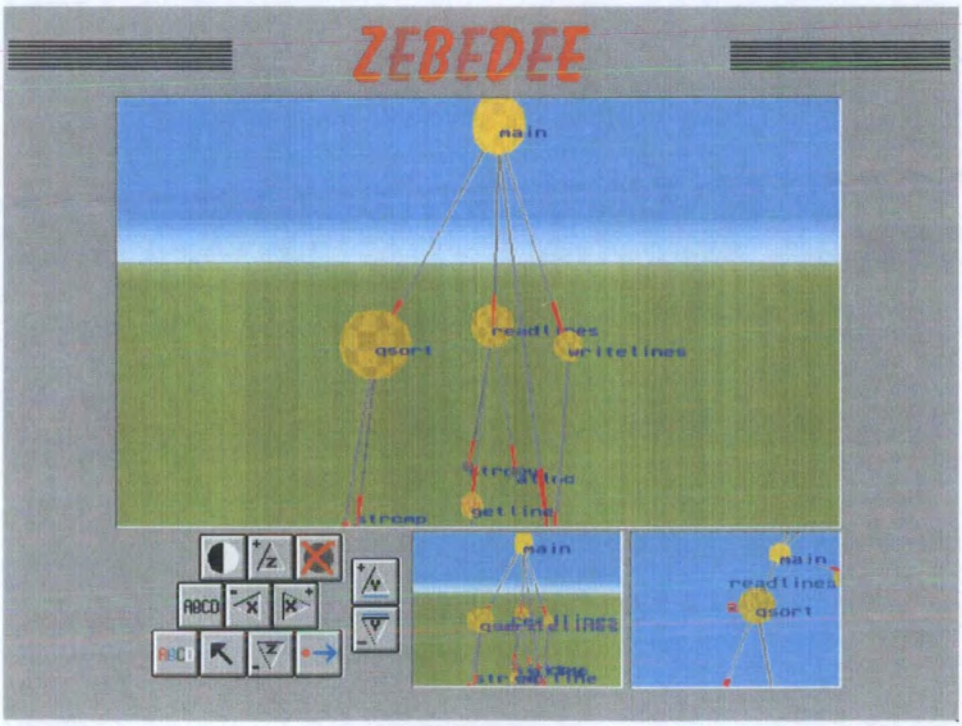


Figure 5.2. Zebedee user interface.

Various features of the visualisation were experimented with, including altering the properties of the representations within Zebedee, i.e. the nodes and the links. Various attributes were altered such as size, colour, animation and rotation, as illustrated in Figure 5.3. Additionally, a number of different approaches to the graph layout were applied in order to improve the structural clarity of the graph. One such modification to the graph layout in Zebedee involves constraining node movement to a set of 2D horizontal planes within the virtual environment. Another approach involves maintaining the original FDP algorithm but providing hierarchical information by colouring nodes depending on their hierarchical level within the graph structure.

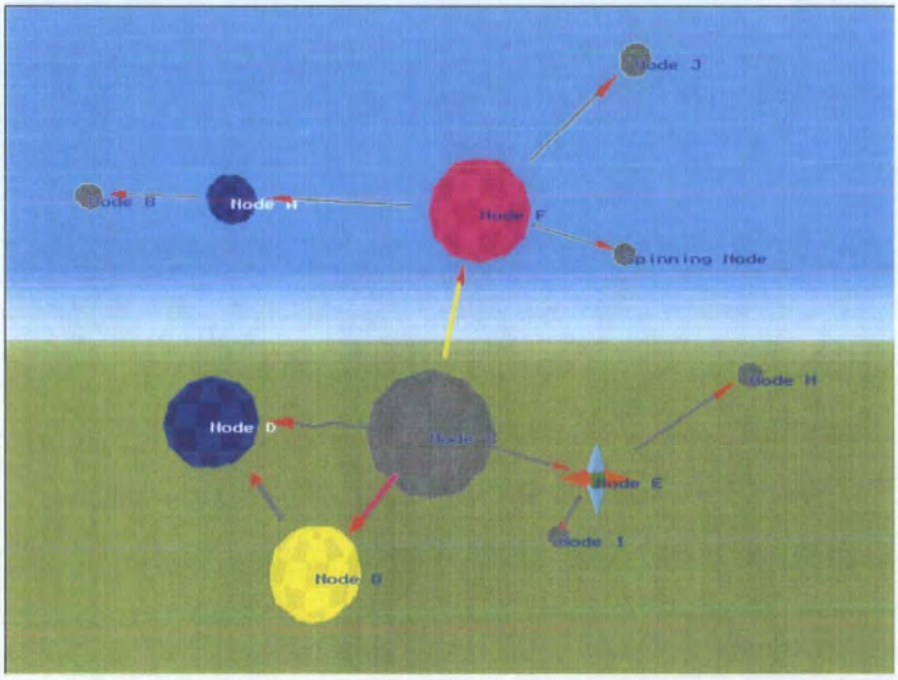
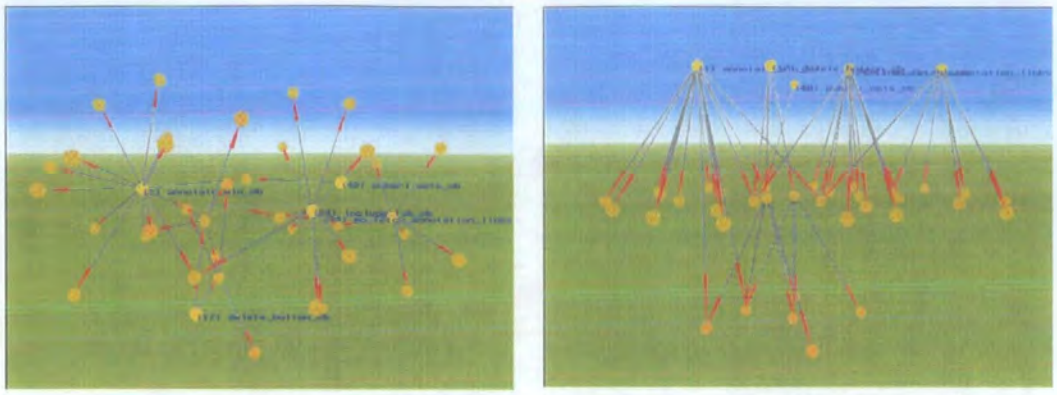


Figure 5.3. Zebedee graph illustrating a variety of node attributes.

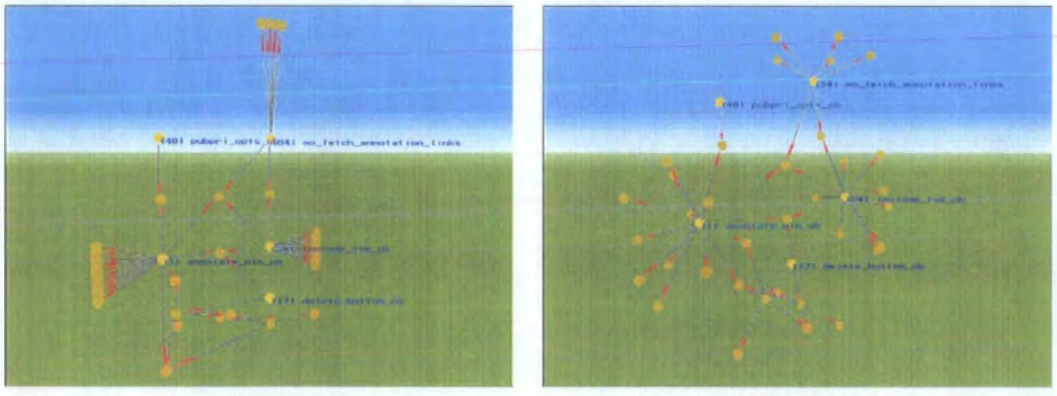
Figure 5.4(a) shows the call-graph generated from a source code module of a large-scale software application. The top-level nodes can be immediately identified by their bright yellow appearance. Three distinct clusters are also apparent, with two of them sharing a number of common functions. Figure 5.4(b) shows the same graph but with an imposed vertical hierarchy. Nodes are given an initial vertical position dependent on their position within the function hierarchy. The FDP algorithm is then applied with the nodes constrained to move only within their allocated horizontal plane. Again it is obvious that there are three main clusters within the graph, however it is now easier to identify the two common clusters and the nodes which are shared between them.



(a) (b)

Figure 5.4. Constrained and unconstrained views of a 3D graph.

One problem with the FDP algorithm used within Zebedee is that the final node placement is strongly dependent on the initial node placement before application of the algorithm. In all the examples shown previously the initial node placement has simply been random, which often results in some undesirable features, particularly when nodes become 'trapped' in unsuitable areas of the graph. This can happen when a node is isolated on the far side of a large cluster of nodes. The combined repulsive force of the cluster is so great that it overpowers the attractive force of the link, which is trying to bring the stranded node through. This has the effect of pulling the graph together. By providing a rough initial placement, which basically ensures that closely related nodes are placed in reasonable proximity of one another, it is possible to provide a superior final layout.



(a) (b)

Figure 5.5. Improved layout using manual initial placement.

Figure 5.5(a) shows a rough manual placement of nodes grouping closely related nodes together and ensuring minimum crossing of links. The layout positions are predominantly two-dimensional with a very small variation in the third dimension to allow the nodes to spread evenly in all directions. Figure 5.5(b) illustrates the resultant graph after application of the FDP algorithm. Comparing this with the same graph shown in Figure 5.4(a) illustrates the improvement in structural clarity.

To summarise, Zebedee provides a 3D graph visualisation, an extension of a well-established two-dimensional software visualisation technique. Within software visualisation and program comprehension the function call-graph is commonly used as a tool for understanding software structure and dependencies. Zebedee supports the visualisation of this structure within a 3D environment, with a view to incorporate this information into a more general, large-scale 3D visualisation of a software system such as FileVis, described in Section 5.4.

5.3 CallStax

CallStax attempts to move away from the standard visualisation of call-graph structures, i.e. a network consisting of nodes and arcs, as depicted by Zebedee. CallStax makes full use of the extra dimension afforded by cyberspace to maximise the amount of information available and the flexibility for displaying and interacting with that information. Development of Zebedee highlighted a number of problems inherent in visualising large, complex graph structures (such as function call-graphs) within a 3D environment. CallStax attempts to address some of these issues by providing a unique method of viewing the graph structure that is specifically designed for integration into a 3D visualisation environment.

It is not generally the number of components within a function call-graph which complicates the visualisation, rather it is the typically much larger set of relationships between these components. Simple operations that we may wish to perform on these graphs, such as grouping particular nodes or finding an acceptable layout, are complicated by the large set of arcs following these nodes wherever placed. CallStax attempts to reduce the complexity overhead that these explicit relationships place on the visualisation, by making them implicit. This effectively reduces the complexity of the visualisation but increases the cognitive load on the user, as they then have to reconstruct these relationships mentally. The benefit of such an approach depends on which is more mentally demanding to the user, attempting to decipher the relationships from a complex visualisation or attempting to reconstruct the relationships from a simpler visualisation. CallStax assumes that the latter will prove more profitable.

CallStax takes a different view on visualising the network of call relations within a software system. 2D visualisations of call-graphs generally visualise the call relations as a network or graph. That is, each component is represented as a single node in the graph and the relationships between components are drawn as arcs between the corresponding nodes. There is no redundancy in this system. Each entity or relationship within the software has exactly one representation in the call-graph. CallStax takes a different approach in that it visualises the *paths* through the graph rather than the graph as a network.

CallStax visualises each possible path through a program as a stack of individual function representations, in the simplest case as coloured cubes. Each of these cubes represents a particular component or function and identical representations or cubes represent the same component. The base function (e.g. `main`) resides at the bottom of the stack, with the functions called along a particular path stacked above it. In order to create the CallStax visualisation, the call-graph must be structured as a

hierarchical tree. Unfortunately, almost all software does not have a tree-structured call-graph, making it necessary to transform the more typical network into a tree prior to visualisation. This transformation is performed by introducing redundancy into the graph, duplicating nodes to remove links between branches of the tree. A similar approach is adopted in the ConeTree and CamTree visualisations described in Chapter 3. The NavigationCones within LyberWorld introduce duplicate nodes and hidden links to produce a tree structure. CallStax introduces duplicate nodes but with visible, implicit links between them.

A CallStax visualisation of a typical software system will result in a large number of stacks, each positioned arbitrarily throughout the 3D environment. In order to make any use of this information it is necessary to provide facilities for querying and exploring the information presented. The basic technique used in CallStax allows the user to select a particular function, or cube, as their current focus of interest. Once selected, all of the stacks in the visualisation will move vertically to align all occurrences of that function within all stacks into a horizontal plane. Any stacks which do not contain that function fall a set distance below the horizontal plane, thus moving them from the immediate attention of the user yet leaving them visible to maintain a notion of context in the results. Optionally, these deselected stacks may be hidden from view completely to allow greater focus on the stacks of interest.

The following example illustrates the construction of a CallStax visualisation using a simple program as the basis. Figure 5.6 shows a standard 2D call-graph of a simple program which reads in lines of text, performs a lexicographical sort, then outputs the ordered lines. The nodes on the graph have been coloured to show the main functions belonging to the program, whereas the plain black-bordered nodes represent library functions called by the program. The CallStax visualisation is constructed by generating a number of stacks of function representations, each stack corresponding to a single path through the call-graph. Figure 5.7 shows a 2D representation of these stacks. The path represented by each stack begins at the lowest function ('main' in the case of these stacks) and proceeds upwards, the deeper the call nesting then the taller the stack.

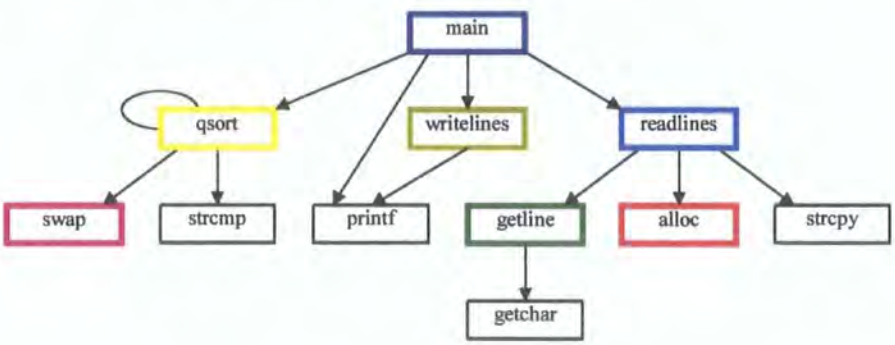


Figure 5.6. Standard 2D call-graph of a simple program.

As can be seen from the stacks shown in Figure 5.7, the CallStax visualisation contains the same information presented in the standard call-graph of Figure 5.6. The main difference between these

visualisations is the manner in which that information is communicated to the viewer. It can be seen from the 2D CallStax shown in Figure 5.7 that the use of 2D makes for a very space inefficient display. Stacks as we would imagine them are inherently three-dimensional structures, they have a variable height and typically a uniform surface area at each level. Figure 5.8 shows the same CallStax visualisation but shown in a 3D perspective view using a virtual reality application.

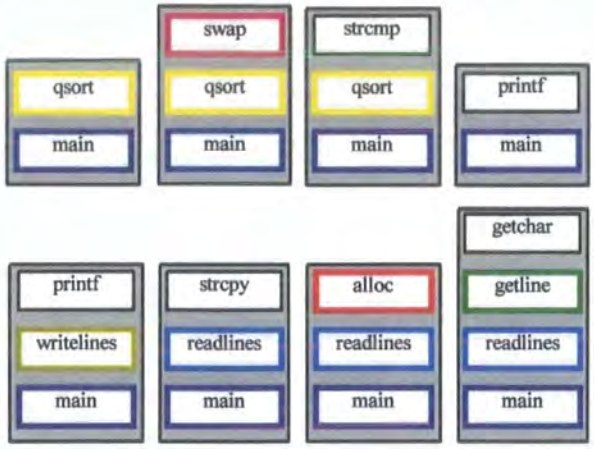


Figure 5.7. 2D CallStax visualisation of the graph shown in Figure 5.6.

Figure 5.8 shows the stacks in a position where the user has expressed an interest in the function *qsort*. The stacks have aligned themselves with all occurrences of *qsort* on the same horizontal plane. This 'selection plane' is visible as a translucent mesh. All stacks that do not contain an occurrence of *qsort* have receded to the bottom of the view. These stacks are still visible to give the user some notion of context when reviewing the results. Similarly, the selection mesh contains holes corresponding to the deselected stacks. This aids users in relating the deselected stacks to their corresponding positions within the currently selected stacks. From looking at the currently selected stacks, it can be easily seen which functions call *qsort*, and which functions are called by *qsort*. Additionally, its depth within the call hierarchy can be rapidly found from the stack which extends the furthest down from the selection plane.

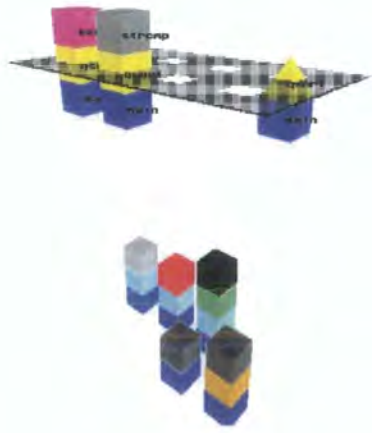


Figure 5.8. CallStax visualisation shown in 3D.

Function call information provides only a superficial view of the software being visualised. When a particular area of interest is located it is often desirable to retrieve further information on that area. CallStax implements this functionality in two ways. Firstly the user can move closer to an item of interest, e.g. a function. As they approach it the original basic representation of the function will change to a more detailed visualisation of the component and its properties (Figure 5.9). The representation shown here displays simple structural and metric information about the function. Secondly, CallStax can be integrated with other information sources such as the source code, metrics information, profiling information or other visualisations.

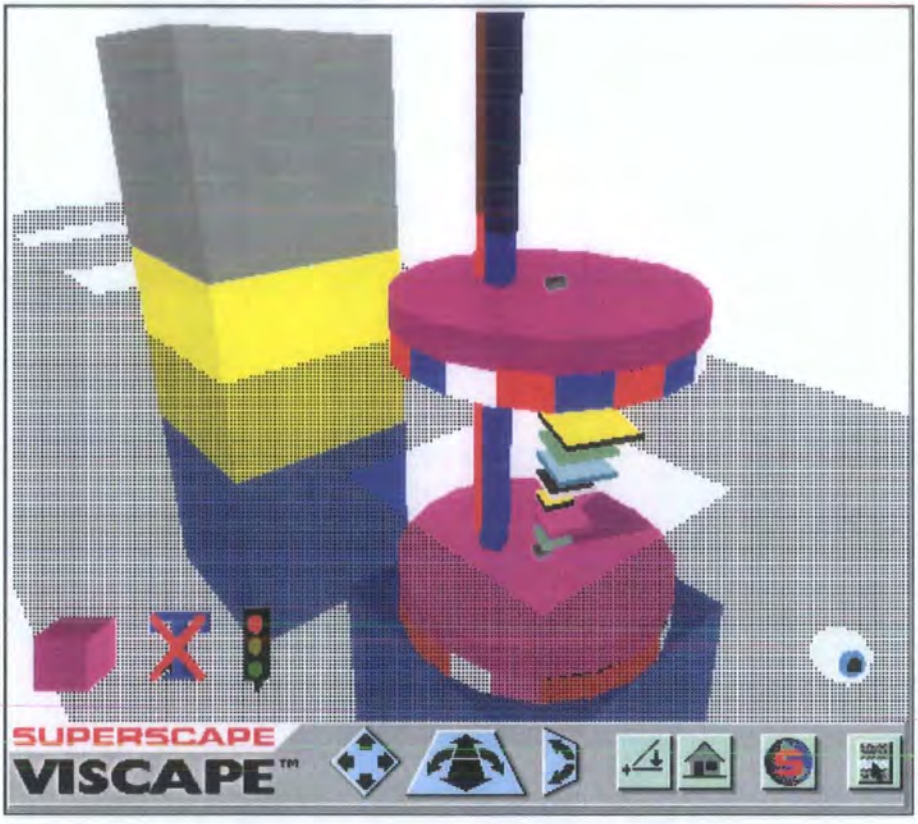


Figure 5.9. CallStax revealing further detail on an item of interest.

The more detailed function representations shown in Figure 5.9 are revealed as the user approaches one of the stacks. Each of these representations displays metric data and simple structural information about the function. The metric details are represented by the two vertical bars displaying the McCabe complexity and Lines Of Code (LOC) metrics [Fenton91, Fenton97] as a percentage of the maximum values within this program. The base of each representation shows a pie chart indicating the distribution of lines of code, comment lines and blank lines within each function. Finally, the coloured planes are a crude depiction of the control structure of the function. The more complex the control structure then the more complex this visual arrangement will become. A quick comparison can be made between the two representations apparent in Figure 5.9. The function *swap* is represented at the top of the stack with *qsort* below it, it is easy to see the relative complexity and structure of these two functions. These representations were described previously in Section 4.4.

Also shown in Figure 5.9 is a simple control set allowing the user to control the visualisation itself, their movement through it and their viewpoint on it. In this simple case, users can specify whether or not deselected stacks remain visible and they can toggle text labels on or off. Other controls governing movement and viewpoint orientation are also provided.

An important feature of software visualisation systems is the ability to correlate or integrate the information they present with other forms of information on the system, such as the source code, documentation or other visualisations. In the case of CallStax this integration is provided via existing web technology which allows a rich multimedia presentation. Figure 5.10 shows the CallStax visualisation integrated into a web presentation. In this example a syntax highlighted source code window and 2D call graph are also available. The user may navigate between windows by selecting appropriate areas with the mouse. For example, selecting a function in the CallStax window will automatically position the source code at the corresponding function definition. Similarly, selecting functions from within the source code or 2D call-graph will result in the CallStax aligning to that function.

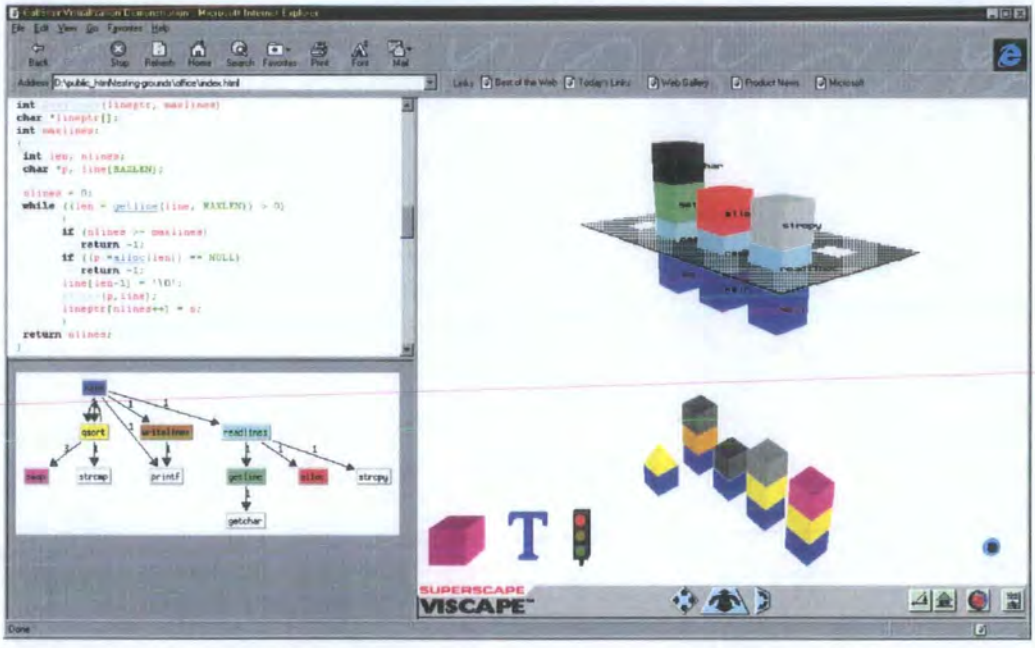


Figure 5.10. CallStax visualisation integrated with other views.

It is important to remember that the CallStax visualisation is not intended to be a single software visualisation in itself. CallStax was designed to provide a method for reproducing the dependency information shown by standard graphs, but in a form that allows for easier integration into a larger 3D visualisation. This property was the main driving influence behind the nature of the CallStax visualisation; it must provide maximum flexibility for integration into other visualisations.

5.4 FileVis

The File Visualisation (FileVis) is a prototype visualisation aimed at providing a high level overview of a software system's structure (Figure 5.11). The goal of this visualisation is to allow a maintainer to familiarise himself or herself with the software and identify any important or interesting areas before they commence their maintenance work. It is often the case that the maintainer will have no previous experience with the software system on which they must work. First contact with the system involves a high degree of learning, during which the maintainer must understand as much as possible about the system as a whole before beginning work on a specific area. Visualisations such as FileVis attempt to support this learning process by providing a more intuitive and easily accessible method for browsing and investigating various aspects of a large software system. The software system is no longer an abstract mass of files and information, it has become something tangible, you can *see* the software. The overall structure and feel of FileVis is similar to that of FSN, described in Chapter 3.

Figure 5.11 Overview of a software system using FileVis.

As well as the 3D visualisation, FileVis is an integrated WWW presentation consisting of three main frames or windows, illustrated in Figure 5.12. The primary frame contains the actual 3D visualisation itself along with a button-bar of pre-set camera positions. The remaining two frames are used to display any other information which is currently relevant in the 3D visualisation. Virtual environments are typically not very effective at displaying textual information, hence the 3D visualisation simply displays any detailed information in these HTML frames. One frame is reserved for displaying information on currently selected objects in the 3D world (top left frame), while the other is used to display the relevant section of the source code files for selected objects (bottom left frame). This integrated approach supports close correlation between the 3D software visualisation and the software system it represents. This provides the maintainer with mechanisms to explore the software system in an as-needed manner, using whichever interface is appropriate at the time.

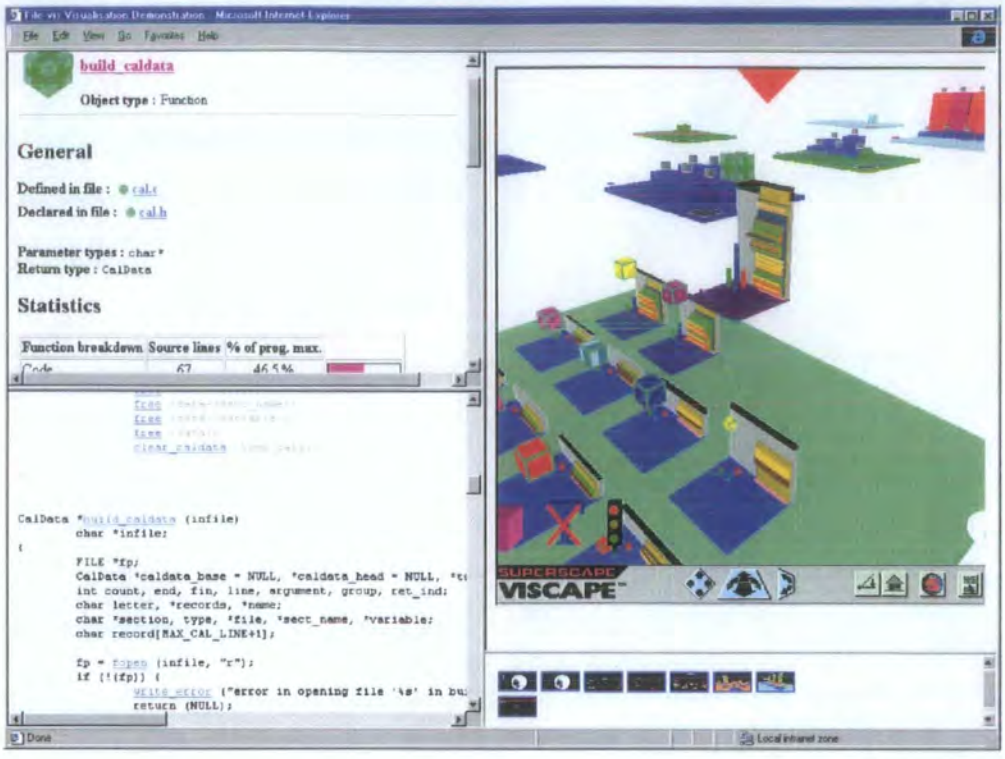


Figure 5.12. FileVis integrated web application.

FileVis is a prototype visualisation based around the C programming language. As the name implies, FileVis is structured around the various C source code files within a software system and the contents of those files. Each file is represented within the virtual environment as a flat, coloured box or platform. The nature and syntax of the C language dictates much of the form and concepts used in FileVis. As such, there are two main types of file, definition files ('`.c`') and declaration files ('`.h`'). These two file types are indicated by a glyph (the 3D equivalent of a graphical icon) in the bottom right corner of each file platform. A spherical glyph indicates a definition file and a cylindrical glyph indicates a declaration file. Files with the same name (i.e. '`test.c`' and '`test.h`') are both coloured identically to indicate their relationship. The declarations in '`test.h`' should (though not necessarily) match the definitions in '`test.c`'.

A CallStax visualisation, which uses the file glyph representations, is constructed in the centre of the visualisation to show the dependencies between the various files. Selecting any of the glyphs in the CallStax or a glyph on a file platform results in the CallStax aligning themselves accordingly. From this it can be seen which files include other files or libraries, and also which files are shared through the system. Additionally, selecting any files in this manner results in detailed information on that file being displayed in one of the HTML frames, and the file contents displayed in the other. This information includes hypertext links between each of the frames and the 3D world allowing the user to browse the software system using a variety of techniques.

In its current state FileVis shows only the function definitions within each of the files. As such, only the definition files (`.c`) are populated whereas the declaration files (`.h`) do not contain any further

information. Upon each file platform are a number of blocks representing each of the functions defined within that file. These function representations have two levels of detail, as the user approaches a particular file or function they switch from a low-detail to high-detail representation which gives more information on the function definition. This has the effect of minimising screen clutter and information overload when viewing from a distance, but more importantly it emphasises the main characteristics of the function of interest by reducing the number of functions visible. Figure 5.13 and Figure 5.14 illustrate a high-detail and low-detail view of the functions within a particular file.

Figure 5.13. Illustrating a file containing low-detail function representations.

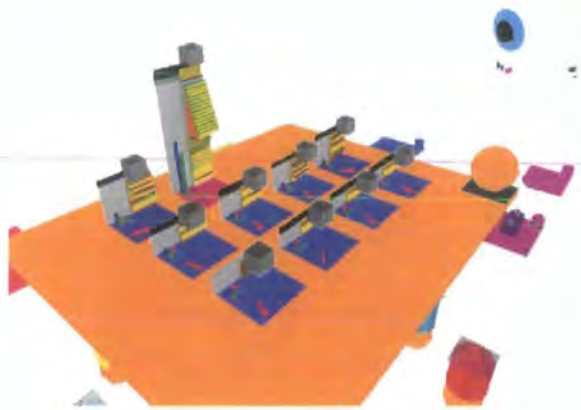


Figure 5.14. The high-detail function representations appear as the viewer moves closer.

The function representations are split into two versions, one low-detail for distance viewing and one high-detail for closer inspection. The low-detail representation emphasises only two characteristics of the function, its length and relative complexity (Figure 5.13). This is represented as a featureless block of fixed base dimensions. The height of the block represents the length of the function, in lines of code, while the colour of the block represents the function's relative complexity with respect to all other functions in the program. The complexity is shown as a colour scale from deep blue to bright red depicting low to high complexity. Using these simple attributes it is possible to quickly assess the

distribution of functions within the software system. Long or highly complex functions or files can be identified quickly by simply surveying the visualisation landscape.

As the user's viewpoint moves towards a particular function representation, at a certain threshold the low-detail tower will be replaced with a similar sized high-detail representation (Figure 5.14). This high-detail representation consists of a number of different information items or attributes related to that function. This includes various metrics such as complexity and a breakdown of the lines of code, comment and blank lines in the function. Additionally a simplified representation of the function's control structure and textual structure is provided.

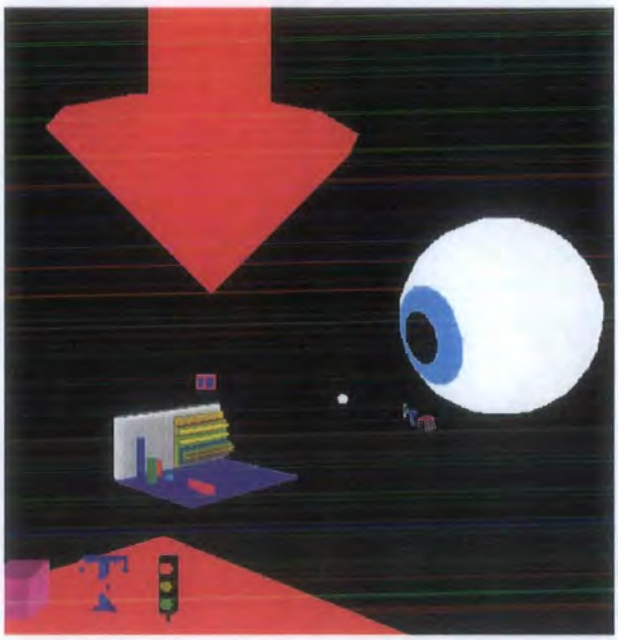


Figure 5.15. Another user within a shared visualisation.

The FileVis prototype was constructed as a predominantly single-user visualisation system. While such a system is of obvious advantage in a single-user capacity, possibly the greatest potential for the application of 3D graphics to software visualisation is the capability for shared, multi-user environments. Within such environments maintainers would be able to explore the software system while being aware of the actions and interests of other users or colleagues. Multi-user visualisations have tremendous potential for supporting software maintenance and software engineering activities, although the large range of issues involved are beyond the scope of this thesis.

FileVis provides a basic exploration into one aspect of multi-user visualisations, that of presence within the visualisation. In order to have a presence or, more specifically, an awareness of other users within a shared environment, each user must be represented graphically in some manner. FileVis represents users as large eyeball objects, indicating each user's current position and orientation. An example of such a representation is shown in Figure 5.15. Users also possess a toolkit containing a variety of options and actions, represented as 3D graphical objects. The currently selected toolkit is visible for all users, hovering in front of their vision. Within FileVis this consists of options to hide sections of the

visualisation, toggle textual labels, and alter the current movement mode. In a more complex visualisation environment the tools currently selected by a user may be representative of their task within the visualisation. With the toolkit of each user visible to others, it becomes possible to identify the interests and actions of users, possibly aiding co-operative working and user interaction.

The following scenario represents a typical use of FileVis, taking the user from initial entry into the visualisation through to a more focused examination of a particular aspect of the software system it represents. The images displayed within this scenario concentrate on the individual frames within the FileVis integrated application. It should be remembered that all the views shown are available within a particular frame, with the user able to select what information is displayed in each frame. This enables the user to survey various aspects of the software system simultaneously.

Figure 5.16 shows the initial viewpoint as the user enters the visualisation. This view shows the software landscape in its entirety from a distant perspective. The user is able to immediately survey the structure of the source code files within the system. At this point the two textual display frames simply show a brief introduction to the visualisation and an explanation of their function, which is to display detailed information on the objects within the visualisation and the underlying source code for these objects. From this distant viewpoint the user can already gain a feel for the structure of the software system, identifying areas within the system containing long or complex code, extensive functionality or small scale utility functions. Simple measures such as the size of the software system, the number of files and functions it contains, and the distribution of functionality, are all intuitively available.

Figure 5.16. Initial entry to the visualisation showing an overview.

In this scenario the user has identified a feature of interest and is opting to concentrate on a densely populated file containing a number of small, simple functions. This file is shown as the boxed area in Figure 5.16. The user then moves towards this file for a closer inspection of its contents, as shown in Figure 5.17. As the file is approached the user becomes aware of the individual function identifiers. These are shown as coloured, patterned, cubic glyphs positioned above each function representation.

These identifiers provide the primary means of correlation between functions in the 3D visualisation and any corresponding reference to these functions within the detailed documentation. The colour of the border on each identifier matches the colour of the file platform, thus aiding rapid identification of a function within the 3D environment by allowing the user to immediately narrow their search field to within one file. Also visible is the file identifier, represented as a coloured sphere in the bottom right corner of the file platform. This serves much the same purpose as the function identifiers, providing correlation with the detailed system information. By browsing the CallStax structure in the centre of the visualisation the user can immediately identify this file's relationship within the software structure.

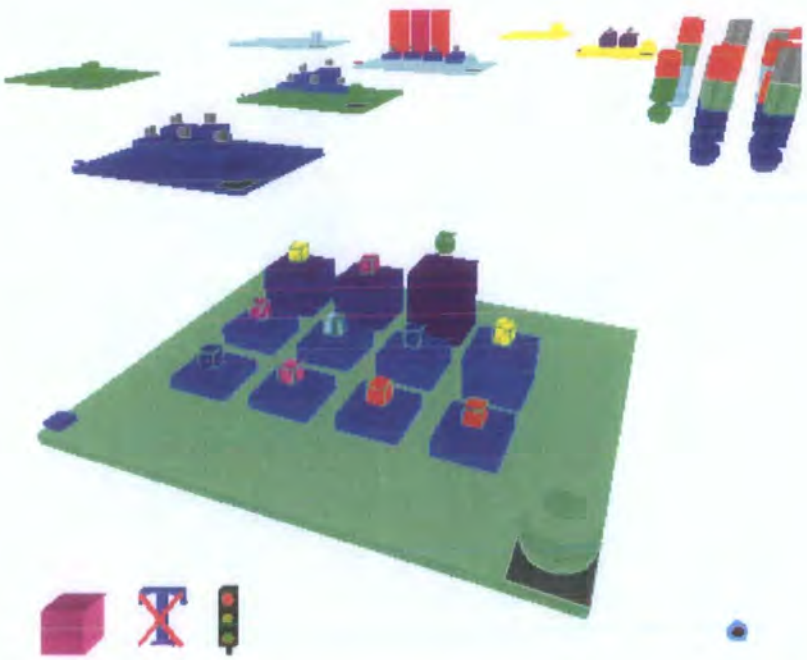


Figure 5.17. Surveying a particular file within the visualisation.

At this point the user decides to call up more detailed information regarding this particular file. Touching the spherical file identifier with the input controller results in a detailed synopsis of the file being displayed in the top left frame of the visualisation. The contents of this file are shown in Figure 5.18. Provided here are a variety of metrics describing this particular file, many of which are also available in a graphical form within the 3D visualisation. This display provides the actual numbers behind the graphical features of the visualisation. Extensive use of the function and file identifiers provides good correlation between the various views of the software system.

Presented within this detailed file view are a number of statistics and details regarding this particular source code file. These details include: files included or imported; files that include or import this file; functions defined within this file; functions called from within this file; and finally some statistics and metrics regarding the contents of this file. For each function identified, a brief summary of its properties are provided. This allows the maintainer to glean important overview information without having to delve deeply into function details. If desired, a hypertext link will provide the user with detailed information on a particular function, while also highlighting its location within the 3D visualisation.

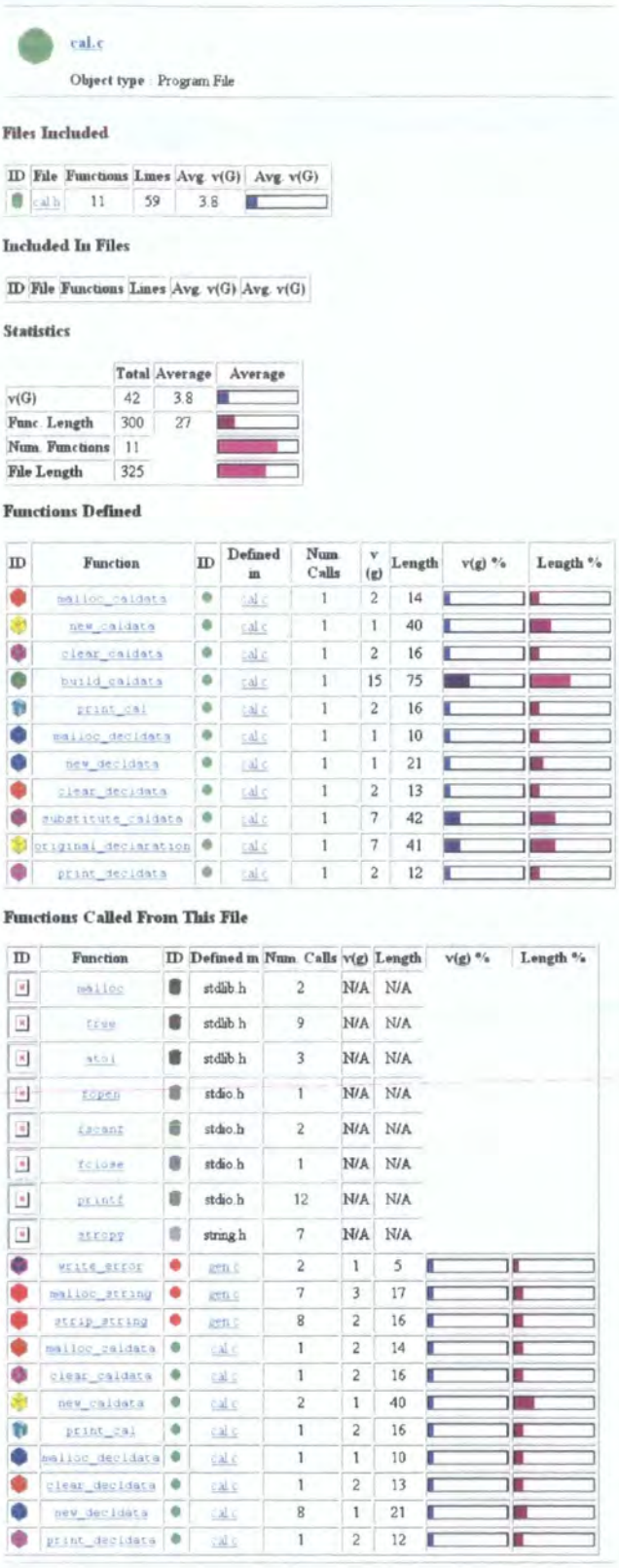


Figure 5.18. Requesting detailed information about a particular file.

While one of the 2D frames displays the detailed information and statistics shown in Figure 5.18, the other frame presents the actual source code of the file. The source code is formatted with the inclusion of hypertext links to both defined and called functions, and also to other included files. These hypertext

links take the user to detailed information on each function or file, and also highlight the location of these objects within the 3D visualisation. If the object selected is a function, the source code window will change to reflect the code of this function. The entire file in which the function is defined is displayed, with all code lightly greyed out except for the selected function. This provides focus on the function of interest while still maintaining contextual information in the surrounding source code. There is also scope for providing additional features such as syntax highlighting, colouring code blocks to match the visual representation, or displaying the graphical function identifiers within the code.

Satisfied that this particular file is of interest to the user, they may then move closer within the virtual environment to acquire more overview information on the contents of the file. As they approach the file, low-detail function representations 'open' to provide a more detailed display of the code structure and various metrics for each function defined within this file. Moving closer still, the colourful function identifiers shrink in size (shown for the two functions furthest right, Figure 5.19). This feature is provided to minimise occlusion of details by these identifiers, making them both clearly visible from a distance but remaining unobtrusive at close range.

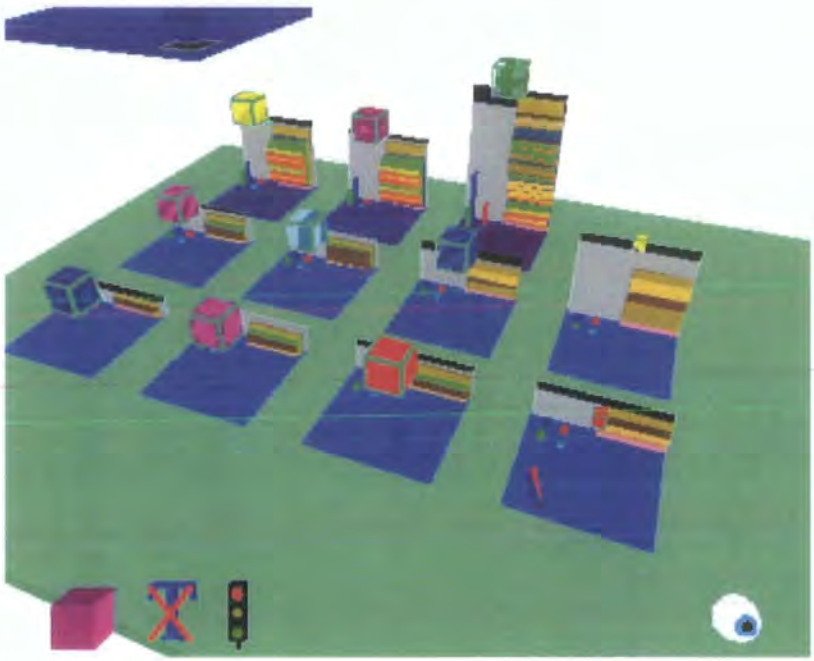


Figure 5.19. Moving in for a closer look at the functions.

Viewing these more detailed function representations draws the user's interest towards the relatively large and more complex function shown towards the rear of this file. The user selects this particular function for closer inspection by touching the function identifier with the input controller. This results in the function representation slowly rising above the file platform, while simultaneously a large arrow moves overhead to indicate the selection, shown in Figure 5.20. This feature provides a readily identifiable locator of selected objects within the 3D world. This is particularly important if the user selects an object, say a function, from within one of the detailed 2D information frames. In this case the corresponding object may be located at any point within the 3D visualisation, and not necessarily within

view. Simply moving the user's viewpoint towards this object would be disorientating and may destroy any notion of location or context. The large arrow allows the user to immediately identify the location of a selected object without having to move position within the 3D environment. Coupling this identification feature with 3D positional audio cues would provide a very powerful location mechanism for objects which may be out of immediate sight.

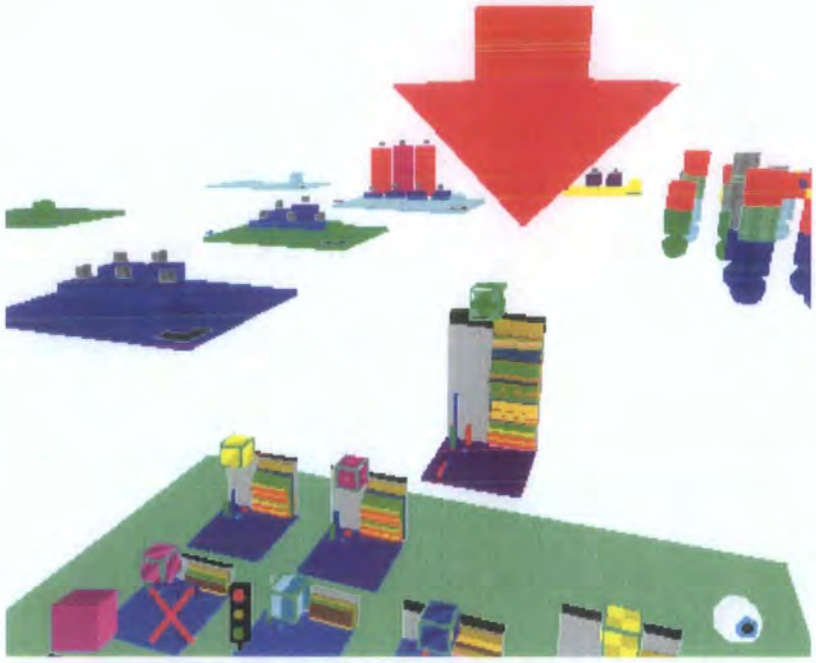


Figure 5.20. Selecting a function of interest for a more detailed inspection.

The detailed function representations provide, at a glance, a variety of information concerning a particular function. This information includes details of the code structure and a number of software metrics. Figure 5.21 shows the particular function in which the user has expressed an interest. A number of features of this representation and the information they encode will now be identified.

Firstly by glancing at the overall size, shape and construction of the representation and comparing it against others within the same file, it can quickly be summised that this function is both the longest and most complex within this file. The colour of the representation base provides a good backup to this assumption, its slightly off-blue appearance indicating that this function has a moderately low complexity with reference to the system as a whole, but is certainly not the lowest. The back-board of the representation indicates that this function contains predominently code-filled lines with a small number of blank lines and, perhaps most importantly, it contains no comment lines indicating a lack of any additional design or operational information. Extensive comments within software code can often considerably aid understanding. The lack of any comments in this particular example indicates that the maintainer must extract all information from the program code, a potentially lengthy and complex process. The four small bars give an indication of a number of software complexity metrics, providing mostly relativistic information with respect to other functions. The blue/red coloured bar placed horizontally on the function base indicates that this function has undergone a number of revisions. This

bar spins about its vertical axis, the rate of turn being proportional to the number of revisions made. In this case the rate of turn is low indicating that the function has undergone only a small number of revisions.

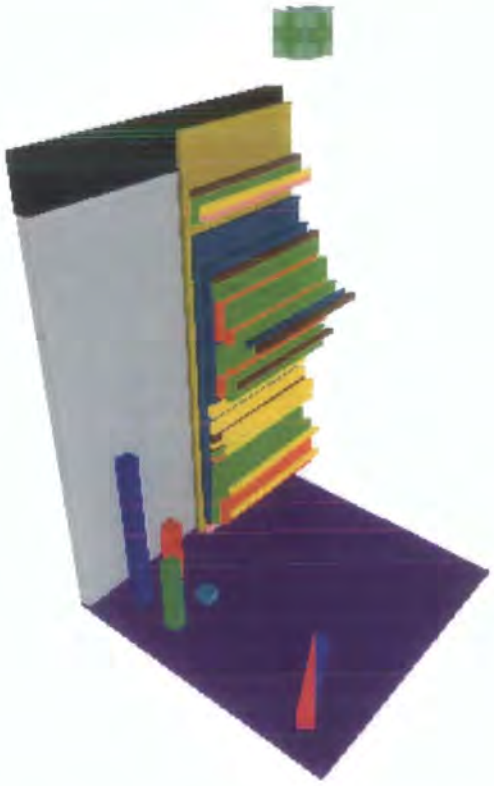


Figure 5.21. Identifying the features of a specific function.

The centrepiece of the high detail function representation provides an overview of the code structure within the function. This is presented as a compound structure of coloured vertical planes. The full height of the function representation is representative of the length of the function, measured in lines of code. Each vertical plane represents a source code block and its scope within the function. As the level of scope nesting increases, additional nested code blocks are presented horizontally displaced. The type of each particular code block is represented by the colour of the vertical plane. The bottom level scope, i.e. the function block, is shown as a light brown plane. Function calls are indicated by either yellow sections, or dark brown sections which represent calls to library functions. Looping code blocks, such as 'while', 'do', or 'for' loops, are shown as blue planes. Similarly, conditional sections of code are shown as green planes which represent the condition-true code ('if' block) and red planes which represent the condition-false code ('else' block). Finally, function return statements are shown as pink blocks. The principles behind this structure are somewhat similar to those of SeeSoft and SeeDiff, described in Chapter 3. The aim is to provide a gross, high-level overview of detailed textual information, maintaining structure and encoding additional attributes.

Using this simple yet effective representation, it is possible to gain an understanding of the function code structure. A number of features can be readily identified from the example shown in Figure 5.21.

This function consists of two main parts, the first is a simple conditional test which may result in an early return from the routine, the second is a large loop structure. The loop structure contains within it another smaller loop which, in turn, contains a structured set of conditional statements and an even smaller loop. Within the middle loop there are no function calls, other than library calls, indicating the processing of some internal or globally available data. Following this loop are a large number of sequential function calls, followed by a number of smaller conditional statements containing additional calls. While the information obtained from this view is very simplistic, it does provide an overview of the function, imparting some of its characteristics in an easily readable and uncluttered format. Using this information it is already possible to form assumptions regarding the role of this function. Additionally, the structural insight provided is also valuable when viewing the detailed source code itself, which may be considerably complex and formatted in an unstructured manner.

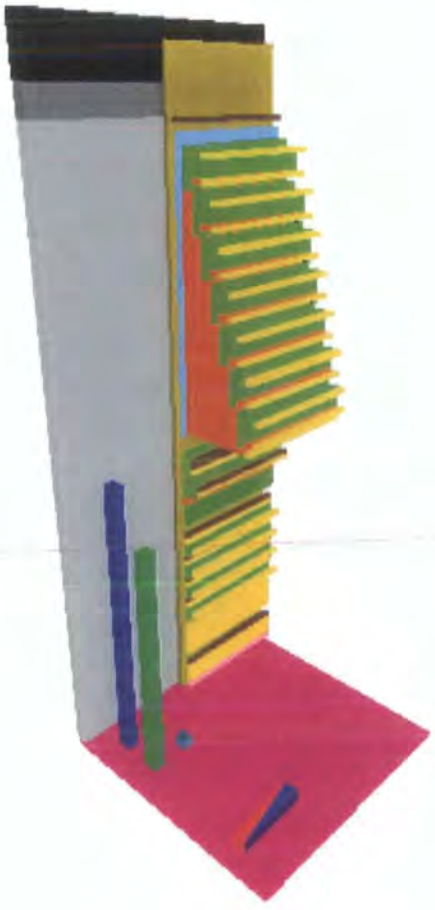


Figure 5.22. Characteristics of the function 'main'.

Figure 5.22 illustrates the properties of a function more familiar to C/C++ programmers, the function 'main'. This function acts as the starting point or entry point to a program. Any command-line options passed to the program on execution are received as parameters to this function. By examining the function representation shown in Figure 5.22 it is possible to identify some common features which are characteristic of the function 'main'. The first noticeable feature is the relatively high complexity metric, shown as both the purple base colour and the high vertical bars. Also of note is the fact that this

function contains only a small number of comment lines, and the rapidly rotating 'change' bar indicates it has received a large number of revisions.

While the metric information is of interest, the representation of the function structure may perhaps provide the most information. There is a very large, regular structure of conditional statements within a loop at the beginning of the function. An experienced programmer could instantly identify this structure, making a fairly accurate assumption as to its function. In this example, that particular section of code is responsible for processing any command-line arguments passed to the program. The loop advances through an array of command-line parameters checking each element against a list of allowable parameters and executing an appropriate function when a match is found. The flat section below this large structure can also be identified as the initialisation and cleanup phase of the program. This section executes the various steps of the program by calling the appropriate functions in sequence before finally exiting. The representation shown in Figure 5.22 could be described as being characteristic of the function 'main'. It is this 'character' of functions, files and even whole software systems which can only be provided by software visualisation. The ability to quickly identify such characteristics enables users to focus their attention quickly on areas of potential interest. This ability is one of the most powerful assets which software visualisation can provide.

Returning to the scenario, selecting the function of interest (as shown in Figure 5.20) results in the topmost 2D frame displaying detailed information on the function while the lower 2D frame presents the relevant source code. Figure 5.23, on the following page, shows the detailed function information. Included within this information are details of where this function is declared and defined; various statistics and software metrics; a list of other functions which are called from within this one; and finally a list of functions which call this particular function. Again, for each other function identified, a brief summary of its properties and a hypertext link to both detailed information on the function, and also highlighting its location within the 3D visualisation are provided.

This example has illustrated a typical usage of FileVis, highlighting how the user can efficiently gain information on the structure and details of a software system. FileVis supports multiple views on the software system, providing detailed, contextual and overview information. The user is able to utilise an as-needed approach to program comprehension, selecting whichever view provides the most relevant information during exploration of the software system. Armed with this knowledge the user can quickly identify areas of interest or sections of the software system that may present potential maintenance problems.

build_caldata

Object type : Function

General

Defined in file : cal.c

Declared in file : cal.h

Parameter types : `char*`
Return type : `CalData`

Statistics

Function breakdown	Source lines	% of prog. max.
Code :	67	46.5 %
Comment :	0	0.0 %
Code + comment :	0	0.0 %
Blank :	8	53.3 %
Total length :	75	46.3 %

	v(G)	ev(G)	iv(G)	pv(G)
Complexity	15	6	7	1
% of program max	31.9 %	18.2 %	29.2 %	100 %
Deviation from mean	+7.9	+1.6	+3.3	0

Functions Called

ID	Function	ID	Defined in	Num. Calls	v(g)	Length	v(g) %	Length %
	fopen		stdio.h	1	N/A	N/A		
	fscanf		stdio.h	2	N/A	N/A		
	atoi		stdlib.h	3	N/A	N/A		
	fclose		stdio.h	1	N/A	N/A		
	write_error		gen.c	1	1	5		
	strip_string		gen.c	8	2	16		
	new_caldata		cal.c	2	1	40		

Functions Called From

ID	Function	ID	Defined in	Num. Calls	v(g)	Length	v(g) %	Length %
	main		convert.c	1	29	129		

Figure 5.23. Requesting more detailed information on a function.

5.5 Software City

The previous visualisations described within this chapter, Zebedee, CallStax, and FileVis, have all concentrated on a very abstract visualisation of a software system. Software City (SoftCity) is a visualisation which attempts to utilise the power of metaphor by matching the information to a metaphor as opposed to matching the metaphor to the information. Visualisations such as FileVis create an information environment based loosely on a particular metaphor, for example, FileVis utilises a weak, city metaphor with files representing cities or districts and functions representing the buildings within those districts. In contrast, SoftCity takes the reverse approach by using a realistic metaphor and mapping the information onto it.

SoftCity is an information city containing buildings, streets, districts and people. These artefacts all appear as they should, within the limitations of the virtual environment. Information is mapped onto various properties of these artefacts to create the structure of the city. For example, in SoftCity each building corresponds to a particular source code file. Each floor within that building represents a function defined within the corresponding file. This simple mapping allows the creation of all the buildings in the city, ranging from small single-storey buildings to giant multi-storey towers.

The primary objective of SoftCity is to present information about the evolution of a software system, showing how the system grows and changes over time. As with the other visualisations described previously, SoftCity is aimed at providing predominantly structural information, and in this case concentrates on a simplistic overview of the file and function structure.

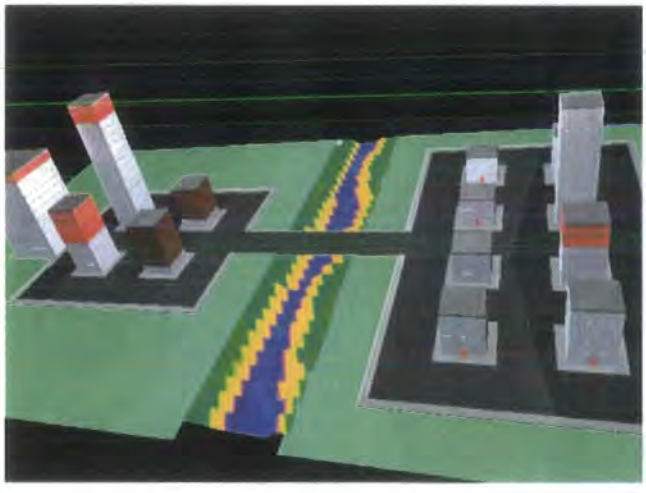


Figure 5.24. Overview of Software City.

Further information must be introduced in order to provide the city with some character and structure, as opposed to a simple arrangement of buildings. In SoftCity, the windows on each storey of a building represent the revisions made to that function. Rather, the absence of, or damage to a window marks a single revision to the corresponding function. In this respect, the floor of a building with two windows either boarded up or broken corresponds to a function that has received two revisions. Additionally, the

age of a function is determined by the brickwork used to construct the corresponding floor in that building. True to life, the foundations of a building represent the oldest parts. Subsequently, if during later maintenance of the software system some new functions are added, then these floors will be built upon the corresponding building in a more modern brickwork. A quick glance at a particular building provides valuable information as to how the corresponding file has changed with time. Figure 5.24 shows an overview of SoftCity. In this image the detail has been reduced to emphasise the variations in function age. The oldest code is shown as light grey, with additions represented chronologically from dark brown through to salmon pink. With full detail selected, these simplistic colours are replaced by a more realistic brick texture with the windows also visible.

FSN, described in Chapter 3, uses a similar approach by colouring file representations within its weak, city metaphor to indicate age. Similar to FileVis, FSN is still a predominantly abstract visualisation and the colour-age relationship is not immediately obvious. Within SoftCity the strong, real-world metaphor enables the user to immediately assume that buildings which appear older represent older code. This is also a strong reminder of why metaphors should not be violated, resulting in false expectations.

Figure 5.25. The 'downtown' area of Software City.

In order to provide filtering, grouping and classification it is possible to divide the city area into a number of districts according to some criteria. As a simple example, SoftCity is divided into two distinct sections, 'downtown' which is west of the river and 'uptown', east of the river. Any files (buildings) which have received greater than a certain number of revisions are placed in the downtown area. Both areas have distinct qualities. The downtown area appears run-down with many boarded and broken windows (Figure 5.25), whereas the uptown area appears relatively unscathed (Figure 5.26). This metaphor can be extended further by assuming that code within the downtown area will also be 'run-down' and typically in the worst condition, having received the most maintenance, bug-fixes, updates and tweaks. In this particular example it is assumed that maintenance performed on a software system is potentially troublesome, leading to degradation of the system structure. The reverse may be true, with maintenance actually improving the structure and quality of the code, for example by adding

comments and documenting the code. It is apparent that the metaphor could easily be extended to represent this, as well as encoding a variety of other information.

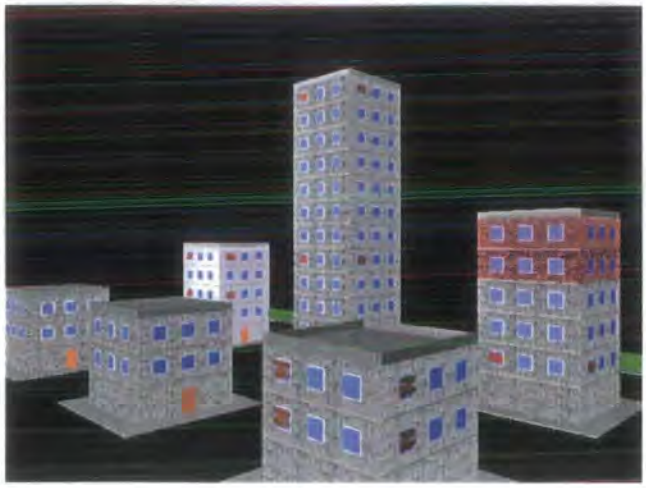


Figure 5.26. The 'uptown' area of Software City.

The original concept of Software City was extended further to provide a more scalable visualisation with scope for visualising an entire software system within an understandable, realistic metaphor. The SoftCity visualisation was adapted to provide a similar information structure to that of FileVis. This adaptation involved a shift in the information mapping. In the original SoftCity a building was representative of a file within the software system, and the floors of each building corresponded to functions defined within that file. Within the new mapping, an entire city is representative of a source code file and the buildings within each city are the functions defined within that file. This mapping corresponds closely to the file and function representations within FileVis. An extension of this real-world metaphor would see a collection of cities (files), for example a country or continent, as a visualisation of an entire program. The metaphor could be extended even further to show collections of programs which make up one particular application or software system, collections of these systems residing on a single platform, and further still. This new visualisation will be referred to as Software Cities, or SoftCities, to reflect its more global metaphor and to provide distinction from the original concept.

A SoftCities visualisation is created through a fully automatic process. The visualisation program takes as input, files which describe the software system to be visualised over a number of revisions. The program then processes these files and generates a number of VRML (Virtual Reality Modelling Language) worlds, one visualisation for each revision of the software. These visualisations may then be explored using any VRML compliant browser. The primary goal of SoftCities is to visualise how the structure of a software system evolves, thus, many of its features are geared towards visualising change.

The cities within SoftCities are represented by a circular base upon which the buildings are placed. This circular base grows to accommodate the size of the city. Each building is shown within the visualisation using two levels of detail. The first level of detail is provided for distance viewing and consists of a

plain, grey tower. Above each tower hovers a small coloured cube, providing a visual cue to the complexity of the function that building represents, in a similar fashion to the FileVis function representations. Future enhancements to SoftCities could replace this cube with a function identifier, similar to those incorporated into FileVis. The reduced-detail building also provides a significant performance increase and helps minimise visual clutter. The high-detail buildings are shown with a textured surface, providing a more realistic appearance. Again, a coloured cube is visible above each building to indicate its relative complexity. This is now coupled with the colour of the building roof matching the visual complexity measure. A tall spire resides within the centre of each city. This provides a visual aid to navigation and easy identification of a city centre, while its colour indicates the average complexity of functions within that city. Such a feature provides a distinctive landmark, one of the legibility enhancement features identified in Chapter 3. Figure 5.27 illustrates a number of cities placed arbitrarily within the virtual environment, one of which is shown in high detail.

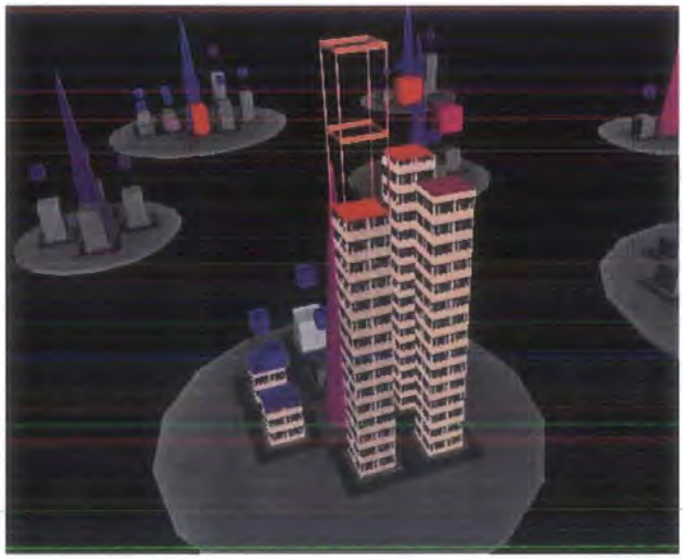


Figure 5.27. One particular city within a Software Cities visualisation.

Buildings are initially placed within one of four quadrants of the city dependant on the values of two metrics, in this case Lines of Code (LOC) and the McCabe Cyclomatic complexity metric. Using this method, long functions are placed in the northern half of the city and short functions in the southern half. This is then divided further so that complex functions are to the east with low-complexity functions to the west. Once a building is placed it cannot be moved in future visualisations of the system. This results in the structure of the city remaining intact and functions, which change drastically, may be noticeable as being uncharacteristic of a particular quadrant.

As previously mentioned, the aim of SoftCities is to visualise how the structure of a software system evolves over a number of revisions. Whereas the original SoftCity prototype viewed change as having a detrimental effect, SoftCities presents it as a positive action. In this way, code which remains unchanged over a number of revisions will be seen to visibly age and appear dirty. Once maintenance is performed on such code its corresponding representation will appear clean again. This simple model

provides information on those areas of a software system which are receiving maintenance effort, while also highlighting areas that remain stable and unchanged. Within the high-detail visualisations this information is shown as a visible dirtying of the building texture with age and neglect. This can be observed in Figure 5.27, in which the central building has a visibly dirtier appearance than its neighbours.

Further to this model of software age, the texture and style of brick used to create a particular building reflects at which stage in the software's life this particular function was added. With each new revision any functions added are constructed using a different style of brick. The low detail representations simply show this as a change in the colour brightness, a light grey illustrating a recently added function whereas a gradually darkening colour shows increased age. This allows immediate assessment of where and when additional functionality has been added. Similar to the addition of functions, new files added to the visualisation are illustrated by their bright base, which gradually darkens with age. In order to aid rapid recognition of areas that have received some degree of activity since the last revision, a thin layer of 'smog' develops above any such cities. A number of these features are visible in Figure 5.28, such as the newly created file towards the front of the image, a number of recently maintained functions to the centre and rear, and smog layers indicating areas of activity.

In order to show how a particular function is modified between revisions, the corresponding buildings will change in size. This is visible as either construction work, indicating an increase in the length of the function since the previous revision, or as demolition work indicating a decrease in the function length. In both cases, the visualisation clearly shows the previous and new length of each function. Figure 5.27 illustrates a function that has increased in length since the last revision. The previous height of the building is apparent, with the construction framework extending to show the new height.

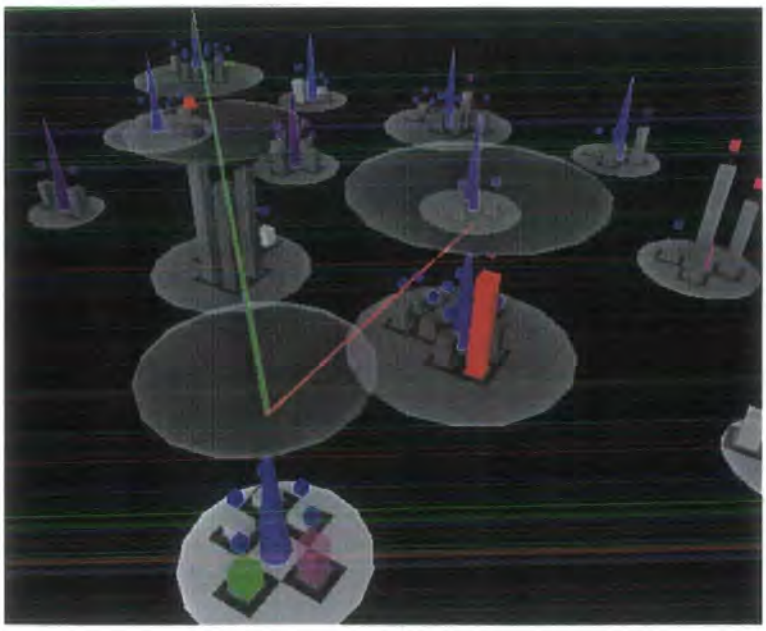
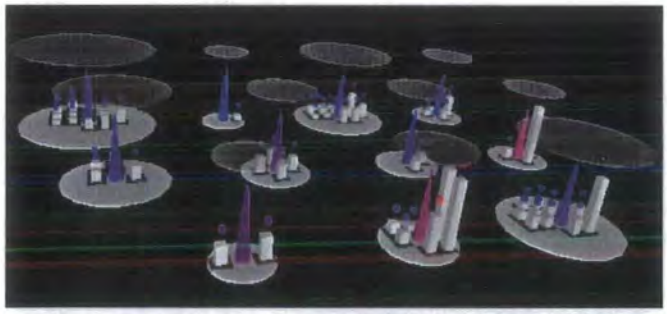


Figure 5.28. Illustrating function dependencies within Software Cities.

Figure 5.28 illustrates the primary means of identifying dependencies between the functions represented in SoftCities. Selecting a particular function with the input controller immediately highlights other functions within the visualisation that either call or are called from the selected function. Functions which call this particular function are highlighted red, whereas functions which are called from this function are shown in green. This is also coupled with red and green lines connecting the file containing the currently selected function to other files containing related functions. This feature immediately highlights any related files and draws the attention of the user to these areas, removing any need to search the visualisation for dependencies.

On selecting a particular function the building representation becomes translucent. This feature can be activated permanently for any number of buildings, allowing a selective filtering-out of uninteresting functions, while still maintaining a slight visual presence. This is particularly useful for removing unnecessary clutter from the display. Figure 5.28 shows a selected function near the bottom of the image. Three dependencies are highlighted. One function is called within the same file, another is called from within a file near the top of the image. Also, this selected function is only called from one other function, which is highlighted red in an adjacent city.

(a)



(b)

(c)

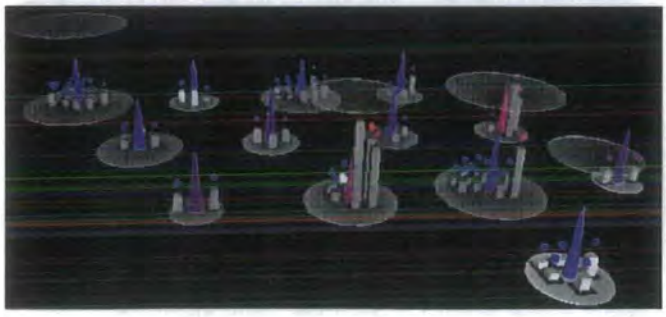


Figure 5.29. Visualising the evolution of a software system.

SoftCities is designed to provide insight into how the structure of a software system changes over a number of revisions. Figure 5.29 illustrates this change by showing three revisions of a particular software system. Figure 5.29(a) shows the first version of the system released. In this visualisation all aspects of the software are new, hence every file and function is shown in the same light-grey colour, and each city has a smog layer above it. It is apparent that the software system consists of 11 source code files of varying sizes. Figure 5.29(b) shows a later release of the system. It can be seen that the functions and files within the system have aged, becoming visibly darker in colour. In this first revision, a number of new functions and a completely new file have been added to the software system. These are identified by their much lighter appearance and can be quickly spotted by examining the cities with a smog layer, indicating recent change. One city at the far left and rear of the image has an associated smog layer, yet no new functions are visible. This indicates that some change has been made to the existing functions within that file. In order to see the details of this change the user would have to move in towards that city for a closer look at the high-detail representations. Figure 5.29(c) shows a second revision to the software system. Again, a new file is added to the system, with another new function added to an existing file. Several files have also received changes to existing functions. Also noticeable is how all the functions have aged visibly, yet the additions made in the first revision are still clearly discernible from the original parts of the system.

Two similar visualisations have been presented within this section, SoftCity and SoftCities. Both have a strong metaphorical basis, making good use of an understandable, realistic metaphor and mapping the characteristics of a software system onto it. Both systems are specifically targeted towards visualising the structural evolution of a software system.

5.6 Summary

This chapter has described a number of prototype visualisations and concept demonstrators that have been developed throughout the course of this research. The emphasis within this chapter was solely to describe the systems, leaving evaluation of their effectiveness and assessment of any strengths or weaknesses to the following chapter. The five visualisations described here are Zebedee, CallStax, FileVis, SoftCity and SoftCities. Each of these visualisations presents a different view of a software system. Zebedee and CallStax both provide different techniques for visualising graph structures and dependency information, while FileVis, SoftCity and SoftCities concentrate on providing more global, structural visualisations of whole software systems. SoftCity and SoftCities provide additional information highlighting the structural evolution of a software system.

Each of the visualisations described within this chapter makes full use of the 3D environment afforded by cyberspace. The subject of automation was considered to be of great importance throughout the development of these visualisations, each of which is designed to be as unproblematic as possible to generate automatically. Two of the visualisations, Zebedee and SoftCities, are already generated using an automatic process, which takes as input a number of relevant facts about a software system, providing a degree of language independence.

The following chapter provides an evaluation of some of the visualisations and representations presented here, using a variety of evaluation techniques, including the desirable properties identified in Chapter 4 as a framework for this evaluation. Each of the visualisations described here is also assessed in an informal discussion regarding their relative merits and weaknesses.

Chapter 6.

Evaluating the visualisations

6.1 Introduction

Chapter 4 presented the heart of this research work, including an identification of seven key areas of 3D software visualisation, a definition of visualisations and representations, and the desirable properties of visualisations and representations. Chapter 5 presented the results of this research by way of the concept demonstrators. These were developed to explore the issues raised in Chapter 4 and to better understand the practical aspects of 3D software visualisation. This chapter presents an evaluation of the concept demonstrators described in Chapter 5 using a number of methods.

6.2 Evaluation framework

Evaluation of the visualisations described in Chapter 5 will take a number of forms. Due to the detailed and lengthy nature of the evaluation presented here, as well as the variety of evaluation techniques used, this evaluation will concentrate predominantly on the FileVis visualisation. A more informal evaluation and discussion of the merits and deficiencies of the other visualisations will also be made.

FileVis has been selected for detailed evaluation because it represents the most complete software exploration tool developed within this research. It has been integrated with other views of the software system and endowed with correlation facilities between these views. It also provides the most varied detail of the visualised software system, enabling the user to delve deeply from a high level overview to detailed statistics and source code. FileVis is considered to embody the essence of this research and a detailed evaluation of FileVis will prove most representative of what this work hopes to achieve.

Section 6.3 presents an informal evaluation of each visualisation described within Chapter 5. This informal evaluation is presented as a discussion of the various merits and deficiencies which were identified within each visualisation during the course of this research. The goal of Section 6.3 is to provide an understanding of the issues involved with each of the visualisations produced.

Section 6.4 describes each of the visualisations from Chapter 5 with reference to the seven key areas of 3D software visualisation. A discussion of how each visualisation applies to these key areas is then presented.

The desirable properties of visualisations and representations introduced in Chapter 4 can be used as both guidelines for the development of 3D software visualisations, and also as a framework for evaluation of an existing visualisation. The FileVis visualisation was constructed with these guidelines in mind. Some desirable properties are in direct conflict with each other meaning that a great deal of compromise must be made in order to produce a well-balanced visualisation. By now evaluating various aspects of FileVis against the desirable properties it becomes apparent where compromise was made and which aspects suffered as a consequence. Section 6.5 provides an evaluation of various constituent visualisations and representations within FileVis, against the desirable properties of visualisations and representations.

Finally, in Section 6.6 an evaluation will be made against a hierarchy of cognitive design elements proposed by Storey *et al* [Storey97]. The authors describe a hierarchy of cognitive issues which can be used to guide or evaluate the design of software exploration and comprehension tools. This evaluation will be performed predominantly against the FileVis visualisation, but with reference to other visualisations within this research as appropriate. Whereas the other forms of evaluation described here are concerned with the suitability of a visualisation as a graphical model, the cognitive design elements provide a better indication of suitability as a tool for program comprehension.

The inclusion of this additional evaluation adds weight to the argument that the visualisations presented in this thesis, or more specifically FileVis, present a viable tool for program comprehension. The other evaluations presented within this chapter are based upon the seven key areas and desirable properties described in Chapter 4. While these provide a suitable evaluation framework for the visualisations, it should be remembered that the majority of visualisations were designed and developed with reference to these key areas and desirable properties. Provision of an external evaluation framework, such as the cognitive design elements of Storey *et al*, adds credibility to the overall evaluation and provides an impartial assessment of FileVis.

The rich combination of evaluations presented within this chapter provide a good indication to the suitability of each visualisation as both a software visualisation and also as a program comprehension tool. Focus is made upon the FileVis visualisation to provide a detailed analysis of a software visualisation tool developed from this research, while also investigating the various merits of other visualisations described here.

6.3 Informal evaluation

This section provides an informal evaluation and discussion of the merits and deficiencies of each visualisation described in Chapter 5.

6.3.1 Zebedee

Zebedee provides a 3D visualisation of the directed graph formed from function call relationships. While Zebedee performs well as a stand-alone visualisation of graph structures, it is not intended to be

used as the sole visualisation of a software system. Zebedee supports the visualisation of these structures with a view to incorporate this information into a more general, large-scale 3D visualisation of a software system. The primary goal during the development of Zebedee was to investigate the direct migration of an existing software visualisation technique into a 3D virtual environment. Zebedee served this purpose well and highlighted a number of issues with this approach.

When applied to relatively small graphs it was found that the force-directed placement (FDP) algorithm produced interesting and readily understandable graph structures. The resulting visualisation is of indeterminate size and aspect but can be readily scaled within the virtual environment to any desired size and aspect. The representations used within the visualisation are basic placeholders, providing little or no information on the software artefacts they represent. This is of little concern as the primary interest here is the visualisation and not the representations used within it.

The unconstrained nature of the layout algorithm was often found to produce some interesting results. Clusters, groups and formations of nodes can be found in the resulting visualisation, which perhaps map to some structural attribute of the software system. Although software has no inherent shape or colour, a graph can be drawn in such a way that it communicates key characteristics of the software [Storey97]. For example, a graph containing a large number of crossing links gives the impression of increased complexity. It is possible that the nature and structure of the graphs produced using Zebedee may impart some feeling as to the nature of the underlying software system. Structural aspects of the graph such as large clusters of leaf nodes or complex, tightly interconnected sections, and isolated branches all provide clues as to the structure of the software. Such benefits are unfortunately countered by problems also inherent in the layout algorithm that tend to complicate the graph structure, leading to apparently tightly coupled areas which, in reality, were not interconnected. This is perhaps due to the complex and tightly connected structure of most call-graphs. The FDP algorithm may be of use in visualising other forms of program relationships which contain natural clusters within their structure, such as class and method relations.

As with almost any graph-based visualisation, Zebedee soon runs into problems as the size and complexity of the software system to be visualised increases. Even some small-scale programs contain horrendously complicated calling structures resulting in complex graphs with a high number of relationships between nodes. Such structures cause problems for the layout algorithm. Tightly coupled function groups tend to bunch together forming tight, complex clusters. Very rapidly the structure of the graph deteriorates to the point where it becomes incomprehensible. Often the 3D nature of the graph would contribute to its apparent complexity, depending on the location and orientation of the user. Links, which appear to cross from one viewpoint, can be seen as clearly separated from another. This necessitates the user to explore the graph structure thoroughly by interacting with the visualisation and navigating through it.

This leads on to another problem, inherent with 3D visualisation, which becomes readily apparent when navigating within Zebedee. The VR system used to implement the visualisation allows a full range of

movement, providing six degrees of freedom (3 translation and 3 rotation axes). This effectively means the user can assume any orientation they desire within the environment, enabling them to view the structure from any feasible position. This is itself a powerful feature allowing the user to explore the visualisation as they see fit, however, it also provides great potential for disorientation within the visualisation. Without any fixed landmarks or features within the graph visualisation it becomes very difficult to maintain a notion of location or orientation within the structure. Zebedee has, by default, a distant horizon that provides limited attitude information. Removing this horizon presents a greater sensation of freedom to explore the graph structure but also incurs a cost in increased disorientation. The provision of orientation and navigation cues must be considered if incorporating Zebedee into a larger visualisation.

Investigation was made into altering the properties of the representations within Zebedee, i.e. the nodes and the links. Additional attributes of the nodes included size, colour, animation and rotation. Each of these was found to provide a powerful means of presenting additional information and particularly for drawing the attention of the user, if used sparingly throughout the visualisation. Spinning or distinctively coloured nodes were instantly noticeable and commanded the user's attention. Altering the size of the nodes, however, presents additional problems with depth perception and was found to be detrimental to understanding the structure of the graph. It proved extremely difficult to differentiate between the distance of a node, and its size, thereby destroying any instinctive assumptions on the 3D structure and inducing disorientation. Providing fixed depth cues, such as a bounding cage surrounding each node, would alleviate such problems but would inevitably clutter the display further.

Various different approaches to the graph layout were addressed in order to improve the meaning of the information presented and also to improve the structure of the graph. The freedom of the 3D FDP layout was found to produce particularly interesting graph structures, leading to readily apparent features of the visualisation such as clustered collections of leaf nodes and densely packed areas of high connectivity. Coupling this with hierarchical information by shading or colouring the nodes, so the top-level functions would appear brightest and the leaf nodes darkest, provided excellent additional structural information. This colouring provides a starting point for the visualisation allowing the user to quickly identify paths through the structure. Other techniques investigated the use of constraints upon node movement within the visualisation, and also the provision of some initial manual node placement to aid the FDP algorithm.

Possibly the biggest problem with the Zebedee visualisation in its current form is the lack of any static visual references to be used as positional aids and, more importantly, as depth cues. Without a static reference point it becomes very difficult to judge depth perception accurately, hindering the comprehension of a 3D structure such as the graphs presented in Section 5.2. This can be illustrated clearly using the graph shown in Figure 6.1. This graph is similar to Figure 5.5(b) as it is based on the same initial manual node placement, the difference is that this graph is purely two-dimensional in structure. The entire graph exists on a single vertical plane within the 3D environment. By inspecting the cluster on the left-hand side of the visualisation, first impressions would indicate that the structure is

in fact 3-dimensional. This is due to the viewer perceiving that the many links emanating from this single node are all of approximately the same length. The assumption is made that the variation in length is due to the nodes being nearer or further from view. In fact, the nodes are all in the same 2D plane but the significantly shorter links give the impression of 3D. These illusions, while difficult to avoid, should be minimised within 3D visualisations by providing static reference objects as depth cues and carefully structuring the environment to avoid potential confusion.

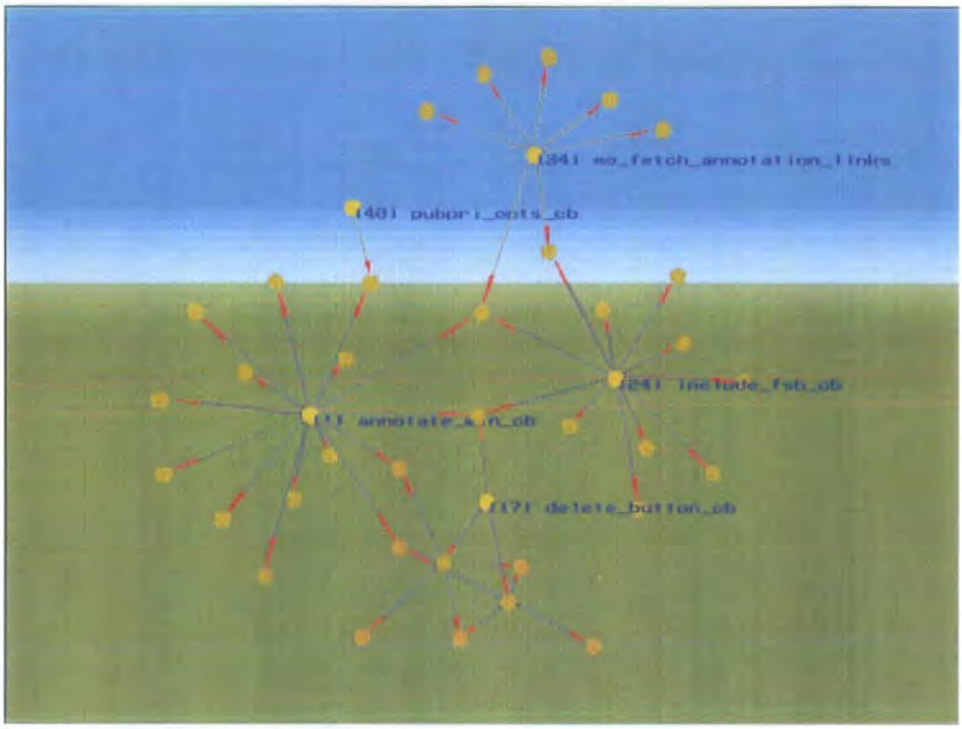


Figure 6.1. 2D graph illustrating the difficulties in depth perception.

To summarise, Zebedee was used to examine the direct integration of an established 2D software visualisation technique into a 3D environment. It was found that the FDP algorithm provided an effective placement solution within the 3D space when a suitable rudimentary initial layout was provided, or when the graph structure was conducive to clustering. By viewing the 3D graph structure it was possible to identify various features which correspond to structural features within the software system. Graph complexity presents the biggest challenge, with large and complex graph structures resulting in large, complex and unintelligible visualisations. It became apparent that the Zebedee visualisation is only suitable for small-scale graph structures and that inherent problems necessitate a more dedicated visualisation of this information within the 3D environment. This necessity was addressed by developing the CallStax visualisation. Zebedee does, however, represent a useable visualisation of small graph structures, capable as a stand-alone visualisation but developed primarily for incorporation into a more comprehensive 3D visualisation.

It was found that as with any graph presentation technique, the layout algorithm used provides the dominant characteristics of the visualisation and contributes greatly to the ease or difficulty with which the structure can be understood. The application of 3D graphics does, however, provide a new medium

within which the layout algorithm can express structure. This was found to reduce visual complexity in some instances while complicating comprehension in others.

6.3.2 CallStax

The goal of CallStax was to provide a method for displaying graph structures within a 3D environment, which would offer maximum flexibility for integration into other 3D visualisations. The aim was to move away from the standard node-link representation of graphs and provide a visualisation that makes full use of the extra dimension afforded by cyberspace. CallStax succeeds in providing a unique solution to the problems of representing graph structures within a 3D environment. There are a number of benefits associated with CallStax, as well as a number of problems. These will both be discussed here.

The greatest advantage of using CallStax is that the individual stacks have no explicit connections between them, making them extremely flexible for placement or grouping within a visualisation. The need for good layout, while still necessary, becomes a less important issue. In a typical graph structure, relations are shown as explicit connections between nodes. These explicit relations become extremely restrictive when attempting to position the nodes of a graph within the visualisation. For example, when wanting to move a semantically related group of nodes together, the large set of explicit relations to other parts of the graph structure become a hindrance. Simply removing these additional relations deprives the maintainer of valuable structural information. The individual stacks within a CallStax visualisation contain only implicit, visual relationships meaning they can be positioned arbitrarily within the 3D environment without loss of information or complication of the visualisation.

Other benefits of the CallStax visualisation include:

- The base dimensions of each stack are identical, thus aiding automation and placement. This also provides a fixed frame of reference to aid in perception of distance and size. This offers maximum flexibility for integration into another 3D software visualisation.

- Information on the position of a node within the graph hierarchy is readily available from observing the height of the stack.

- An entire path through a graph is shown within each stack. This has potential for quickly identifying the possible impact of any changes to a function within that path.

- CallStax effectively provides a collection of discrete graphical slices into the structure of a software system.

- The scale of the system under scrutiny does not adversely affect the visualisation to the same extent as in a standard graph.

- The method of interaction provides a useful filtering mechanism, focusing the visualisation on information directly relevant to the user's current interests while still retaining context with other deselected stacks.

Unfortunately, the additional flexibility of CallStax comes at a price. The implicit relationships between duplicate representations of functions are reliant solely on the visual appearance of those functions. The representations used must therefore concentrate on being unique and distinctive. It is relatively easy to fashion such properties in the case of a small visualisation but it becomes very difficult when a large number of distinct functions are present. Within a standard call-graph, it is the node position and the node label that provide the unique and distinctive representation, CallStax can rely on neither of these (though a label can provide a unique identity within a small collection of stacks). Section 4.4 highlighted the complications involved with selecting a suitable representation.

The two properties, individuality and distinctive appearance, are so critical within CallStax that the representations should be designed to support only those, if necessary at the sacrifice of displaying any other information. In order to provide more detailed information on each representation, multiple levels of detail should be incorporated, triggered either by proximity or by user selection. This would provide a good compromise allowing the CallStax to work effectively while still providing more detail as needed by the user. An example of using varying levels of detail for the function representations is shown in Figure 5.9. At a distance the representations used in the stacks remain extremely simple and concentrate solely on the two properties of individuality and distinctive appearance. By moving in closer to a stack of interest, these representations 'open up' to display more detailed information on each item.

To summarise, CallStax represents a completely new view of graph structures, designed to fully utilise the additional benefits of a 3D environment. The visualisation is intended to provide maximum flexibility for integration into other 3D visualisations and succeeds admirably in this respect. By far the biggest problem with the CallStax visualisation is in the selection of suitable representations for use within it. This is a non-trivial task, but is one that can be overcome by judicious selection of shapes, colours and textures for use in representations, and the thoughtful grouping and positioning of stacks. As with any visualisation of graph structure, it is inevitable that there will be an upper limit to the size of the graph it is possible to visualise. CallStax succeeds in reducing the graphical clutter inherent in such large structures.

6.3.3 FileVis

FileVis represents a prototype visualisation system aimed at providing a high-level overview of software structure. The goal was to represent the system structure and static relationships in their entirety, allowing the user to gain both overview information then delve deeper for specific details. The intention was to produce a software landscape, in which the user's surroundings provided clues as to the

nature of the underlying software, allowing the user to explore it freely using a variety of interrogation techniques.

CallStax presented a relatively simple scenario in identifying the various visualisations and representations used within it. The necessary desirable properties were easily identifiable from the outset, enabling them to be designed into the CallStax visualisation. FileVis offers a much more complicated scenario in which, at each level, the various components must be considered as both visualisations and representations. A careful balance must be struck between the desirable properties of both. An evaluation of how the various components of FileVis apply to each of the desirable properties, both as visualisations and representations, is described in Section 6.4.

FileVis has a number of beneficial features as a software visualisation. Most notable is that it provides an instantly accessible overview of the software system file and function structure. On entering the visualisation, a user can gain an immediate impression of the nature of the software system by simply surveying the landscape of the visualisation. Every feature of the visualisation provides some form of information or metric about the software.

The various representations used within FileVis have minimum explicit connections, allowing greater flexibility for positioning and layout. The prototype described in Chapter 5 used an arbitrary layout, however, more meaningful collections of files and functions could be created by a more sophisticated system. Additionally, the lack of explicit connections between any of the files or functions enables the user to position or group these units as they see fit, allowing them to impart some semantic structure to the visualisation.

The varying levels of detail used provide a clear overview while still supporting focus of interest to obtain more detailed information. The user is able to quickly identify features of interest then move in closer for a more detailed look. As they approach, program units such as the function representations expand to provide more information. Similarly, as the user moves away, they reduce in detail to decrease the amount of visual clutter in view. This dynamic focus maximises the information content given to the user by ensuring that only the most relevant information is visible at any time.

Integration with standard 2D windows, by way of HTML frames, provides views on the underlying code for the system, and highly detailed information which is difficult to present within the 3D environment. Additionally, these alternative views allow the user to explore the software using a variety of techniques at their discretion. This approach supports the use of both top-down and bottom-up comprehension models in an as-needed manner. It provides support for the integrated comprehension models proposed by Mayrhauser and Chan [Mayrhauser95, Chan98].

Movement within FileVis is supported by the use of multiple cameras. These cameras allow the user to effectively 'bookmark' their position within the visualisation by leaving a camera in place. The user is able to swap between camera units at will, as well as selecting some pre-defined viewpoints displaying key aspects or popular positions within the visualisation. For example, one predefined viewpoint

positions the user at his or her original point of entry to the visualisation enabling them to return to a familiar point if they become disorientated or lost. In the current implementation of these camera units the user may exercise some control over the movement technique used.

The biggest deficiency of FileVis is the current lack of information regarding the function calling structure. This information could be presented using an additional CallStax structure, which would make use of the coloured function identifiers. A further deficiency is the lack of any strong metaphors to guide the user towards interpretation of the visualisation. The information presented is very abstract and it is unclear what each aspect of the visualisation represents without prior instruction. Similarly it is not obvious which aspects of the visualisation can be interacted with. The VR system does provide some support for this by altering the mouse pointer over objects to signify they can be activated. The user should be able to identify such objects more easily, without the need to search for them.

Another major deficiency with FileVis is the fact that all high-detail function representations face in one direction. While such an approach helps by providing a common format to the visualisation, it means that a user approaching from the wrong direction must circumnavigate a file to view its contents clearly. The low-detail function representations are omni-directional but are ordered with the largest representations towards the rear of each file platform. This essentially occludes any smaller function representations if the user is viewing the platform from the rear. A much better solution would be to allow the function or file representations to pivot about their vertical axis, orientating themselves to face the user at all times. This would allow the user to view any file or function in detail while still maintaining a useful view of the rest of the visualisation. Such an approach would have further implications in the case of multi-user, shared visualisations, and may actually be inappropriate.

To summarise, FileVis is an information-rich visualisation which provides both an overview of, and detailed information on, the structural aspects of a software system. The user is able to make use of a variety of exploration techniques, enabling him or her to freely browse the software system and investigate any aspect of interest. FileVis supports an as-needed, integrated program comprehension model.

6.3.4 Software city and software cities

Software City is a visualisation which attempts to utilise the power of metaphor in creating a readily understandable information environment. Visualisations such as Zebedee, CallStax and FileVis concentrate on very abstract representations of the software system. SoftCity encodes information about the software system onto realistic and identifiable real-world metaphors in an attempt to aid user understanding of the visualisation. Section 5.5 in Chapter 5 describes two variations of the software city visualisation, SoftCity and SoftCities. Both use a similar metaphor but encode information onto the metaphor in a different manner. Both visualisations are designed to visualise structural change within a software system over a number of revisions. To avoid confusion, the merits and deficiencies of these two visualisations will be described in turn. To reiterate, Software City (referred to as SoftCity) and

Software Cities (referred to as SoftCities) are different visualisations which share a commonality in the metaphor used. SoftCity will be described first, followed by SoftCities.

SoftCity represents the files of a software system as buildings within a city, the individual floors of these buildings represent functions within those files. Revisions made to a function are represented as a deterioration in the appearance of a building floor, function age is illustrated by the brick texture used in its construction.

The metaphor used within SoftCity is easily identifiable, although the exact mapping of features within the visualisation to features of the software system requires some initial explanation. Once this mapping is understood, the evolution of the software structure can be related to the changing state of the software city. Other aspects of this metaphor can be extrapolated to provide further information on the characteristics of the software system. For example, the segregation of the city into districts provides other contextual clues. SoftCity is divided into two districts, downtown and uptown. Downtown contains all the files that have received significant change and revision, while uptown contains relatively unscathed files and functions. This simple feature already provides an indication of the quality of code which a maintainer is likely to encounter within the files of each district.

Although the SoftCity visualisation does provide information on the evolution of software structure, perhaps the biggest deficiency is the lack of any substantial detailed information. The real-world metaphor used leaves little flexibility for encoding software information, providing only a very basic set of details. The fact that such a concrete metaphor is used also restricts the nature of the visualisation. Metaphors are very effective tools for visualisation, but only remain so as long as the user's expectations are not violated. Breaking a metaphor can destroy any benefits gained from their use, adversely affecting comprehension of the visualisation. This restriction makes incorporation of additional information particularly difficult.

The SoftCities visualisation applies information about the software system to the city metaphor in a slightly different manner. Files within the software system are shown as cities within the visualisation, with the functions defined within each file shown as the buildings of the city. SoftCities attempts to find a suitable compromise between the use of realistic metaphors and the flexibility of a more abstract visualisation. This is achieved by providing a realistic metaphor as a basis for the visualisation and introducing more abstract features as necessary, without breaking strong metaphors. This enables further information on the software to be incorporated, such as metric information and function call relations.

SoftCities provides a more complete picture of how software structure changes over time. The visualisation supports the display of incremental changes over a number of revisions to the software, as well as highlighting the location and nature of these changes. The visualisation supports varied levels of detail, reducing the visual complexity and information content when viewing from a distance. During this distance viewing the most salient features of the visualisation are visible. Software age and change information is presented at this stage, providing an overview of how the software system as a whole has

changed. When moving in closer towards areas of interest the detail level is increased, providing specific information on the changes.

All of the visualisations and representations presented within this thesis have been designed with ease of automation as a primary factor. SoftCities is one visualisation in which automatic generation has actually been implemented, Zebedee being the other. It demonstrates that automatic generation of these visualisations is a realistic goal. The generation process takes as input a set of program facts and relations, providing a more language independent implementation. A similar automation process could be developed for FileVis.

To summarise, SoftCity and SoftCities present a more metaphor-orientated software visualisation. As with the original SoftCity, SoftCities lacks somewhat in the extent of the information provided when compared to more abstract and information-rich visualisations such as FileVis. The focus of the information presented is different to that of FileVis. SoftCity and SoftCities are geared towards visualising change whereas FileVis is not. Both succeed in utilising a strong, realistic metaphor which provides an understandable and identifiable impression of the software system.

6.4 Evaluation against the key areas of 3D software visualisation

This section will evaluate how each of the visualisations described in Chapter 5 addresses the seven key areas of software visualisation. These seven areas were identified in Chapter 4 as representation, abstraction, navigation, correlation, automation, interaction and scalability.

6.4.1 Representation

The representation of the software structure within Zebedee is a relatively simple node-link graph structure. This structure moulds the nature of the visualisation to provide a graphically explicit representation of relationships within the software. The size, complexity and structure of the visualisation are all unknown factors, depending entirely on the input data and the layout algorithm used. The representations used within this visualisation are necessarily plain, providing only limited additional information. The lack of fixed depth cues within the visualisation, and the need to understand the 3D structure necessitate minimal visual complexity.

CallStax provides the same information presented in Zebedee but in a completely different manner, one more suited for inclusion within other 3D visualisations. The representation of relations in CallStax is in complete contrast to the explicit relations shown in Zebedee, using predominantly implicit relationships. The dependence on easy identification of these implicit relationships places very strict design guidelines on the representations used within CallStax. This reduces the information content of the visualisation to show purely relational information.

FileVis uses a weak, city metaphor in its representation of software structure. The visualisation utilises very abstract representations to maximise information content. Every feature and aspect of FileVis encodes some form of information about the underlying software system. The overall structure of FileVis provides an information landscape, submerging the user within the software system. This is the intended use of FileVis, as an environment representative of the software system within which the user can explore and query any aspects of interest. The abstract and information-rich nature of the visualisation creates a colourful and surreal world with basic concepts of use which, although not obvious, can be easily understood.

SoftCity and SoftCities use a metaphor-rich representation, mapping the information from a software system to the attributes of a realistic city structure within the visualisation. This use of strong, identifiable metaphors provides both advantages and disadvantages. The strong metaphors ease understanding of the concepts involved in the visualisation while also restricting the amount and type of information which can be encoded into the visualisation. Both of these visualisations create a realistic environment with simple, understandable concepts but provide a relatively small amount of information on the software system.

6.4.2 Abstraction

The visualisations presented in Chapter 5 can be split into two categories of abstraction. Zebedee and CallStax both abstract various structural relationships of a software system into a graph. This information is then presented in two distinctly different formats. The remaining visualisations, FileVis, SoftCity and SoftCities all abstract structural information about the software using gross software objects such as files, functions and lines of code as the basis. The visualisations are structured around these software objects, encoding additional information such as software metrics and revision information into the various representations. All of the visualisations here present a gross overview of software structure, a view which the maintainer is unable to obtain himself or herself from the source code alone.

6.4.3 Navigation

Zebedee is very lacking in the provision of navigation features. The visualisation produces complex network structures within 3D space, structures which have no explicit orientation. Despite this, there are very little navigation cues to aid comprehension of this structure or to reduce disorientation. The only feature present to provide any cue to orientation is a horizon. Textual labels are also available, but due to the nature of the VR system used in the implementation, these rotate so that they are correctly readable regardless of user orientation or position. Additionally, there are no depth cues present within the visualisation to aid comprehension of the graph structure. Again, the text labels do not scale with distance so provide no assistance and only serve to clutter and obscure the display as more become visible.

CallStax also lacks any navigation cues but was intended as a component within a larger visualisation, which would presumably support navigation itself. Some features are available in a pure CallStax visualisation, such as the translucent selection plane. This provides a form of horizon, acting as an orientation cue. It also helps to provide contextual information for deselected stacks by containing holes within which the stacks will fit. The selection plane also aids in depth perception of distant stacks, enabling the user to ascertain more easily their vertical position.

FileVis provides navigational assistance in a number of ways. The overall structure of the visualisation is as a number of distributed platforms, upon which reside a varied collection of function representations. Each file platform is distinctively coloured with an identifier to provide a definitively unique appearance. These platforms each have very different and individual characteristics, behaving as landmarks in their own right. The structure and distribution of these platforms is in a logical hub formation, with the main files positioned in the centre of the hub. The centre of the hub is also a highly visible and recognisable structure containing a number of CallStax representations. This provides a central landmark to which the user can return at any time. Further navigation features are provided such as the multiple cameras and pre-set camera positions. These allow the user to return to a known location from any point within the visualisation. One such camera position provides an overhead view of the visualisation, essentially acting as a map, which the user can switch to at any point and observe their current position within the visualisation from a third person perspective.

SoftCity and SoftCities also provide additional navigation features. In a similar manner to FileVis, the cities within these visualisations have certain characteristics which can be identified and recognised. These characteristics are, however, not as distinct as those in FileVis. Within SoftCity, navigation is aided by the structure of the visualisation, which is divided into two very different districts. The division is also reinforced by the image of a river and bridge, providing directional and positional cues within the visualisation and acting as a combined central landmark, path and edge. SoftCities does not have any such navigation features between cities, but within each city the buildings are split into four districts with an instantly identifiable and visible central landmark at the junction of these districts. District boundaries are also regularly placed and easily identifiable when crossed.

6.4.4 Correlation

Both Zebedee and CallStax were implemented into a prototype system which provides correlation with other views and information on the software system. These systems use a web interface similar to that shown for FileVis. The three frames of the interface contain the 3D visualisation, the syntax highlighted source code for the software, and an equivalent 2D function call-graph. Interaction within these interfaces is bi-directional allowing both the selection of relevant 2D information from the 3D visualisation, and the identification of software objects within the 3D visualisation when selected from another source.

FileVis provides the highest degree of correlation within the visualisations presented here. The web-based interface provides integration with a number of 2D views on the software system. Again, this interaction is bi-directional allowing the user to browse the software system using whichever view suits at any stage. The web-based prototypes of Zebedee and CallStax provide relatively limited correlation by simply allowing the user to select objects in one window and have them appear in another, there is very little visual correlation. FileVis provides additional correlation by using the function and file identifiers as a visual link between the different views on the software system. 2D images of the 3D identifiers are used extensively alongside any mention of a software object which appears in the 3D visualisation. This allows the user to quickly identify any relationships and to help tighten the integration between the different software views. This graphical correlation is complimented by the provision of links between software objects in each view, allowing exploration of the software system through a multitude of routes.

SoftCity and SoftCities both provide no correlation with other information sources. The automatic generation of SoftCities currently only deals with generation of the 3D visualisation. It would be possible to introduce links between the visualisation and other information sources, as demonstrated in the other visualisations presented here. The strong metaphor-based nature of these visualisations does, however, hinder graphical correlation as in FileVis, due to the similar appearance of many of the representations. Function identifiers, again as demonstrated in FileVis, could be introduced to help.

6.4.5 Automation

Zebedee and SoftCities represent the only two visualisations within this research for which automatic generation has been implemented. A simple automatic generation of CallStax was also implemented to demonstrate its potential, but was not developed further. Both FileVis and SoftCity were developed with the capability for automatic generation a high priority. SoftCities does in fact represent the automation of SoftCity, while advancing the design of the visualisation in the process.

Each of these visualisations was created with features designed specifically to ease automatic generation. Features such as regular sized, and regular spaced representations aid in producing a suitable layout. Aspects of FileVis such as the colour scale indicating complexity and the structural image of the code all have clearly defined limits and relationships to the software system and surrounding representations.

As well as providing ease of automation, these visualisations are also designed to be as flexible as possible with minimum explicit connections. This allows the user to alter aspects of the visualisation manually with little impact on its structure or graphical complexity. Files and functions within FileVis can be easily rearranged and grouped to provide more semantic meaning. The goal is to use automation to provide a rough framework for the visualisation, such as FileVis in its current state, then allow the user to manipulate this as she explores, providing a deeper semantic meaning which is extremely hard to encapsulate automatically.

6.4.6 Interaction

The visualisation systems described in Chapter 5 all provide very little interaction, although many have been designed to support future enhancements. The exception to this is CallStax in which simple interaction plays a vital aspect in the understanding of the underlying graph structure. The user can select any object within the CallStax resulting in the corresponding stacks aligning vertically. This provides an interactive method for querying the visualisation through a very simplistic interface. Other virtual tools are also available as a floating toolbar in front of the user's viewpoint, allowing the user to alter various aspects of the visualisation. FileVis, Zebedee and SoftCities also provide simple interaction by way of clicking and selecting objects within the visualisation and other related information. As mentioned previously, visualisations such as FileVis and CallStax have very few explicit relationships between software objects. This provides significant freedom allowing the user to position and group objects manually, imparting some semantic information into the visualisation. Support of features such as annotating, highlighting and editing visualisation contents would prove advantageous, but are not currently implemented.

6.4.7 Scalability

All of the visualisations described within Chapter 5 are designed to visualise medium to large scale software systems. The actual suitability of each visualisation to this task is somewhat different.

Zebedee provides an effective visualisation of 3D graph structures, however, its effectiveness reduces with increased graph size. The visualisation becomes increasingly visually complex due to the high number of explicit relations visible. This is a problem shared with more traditional 2D graphs. With small to medium sized graphs, Zebedee provides a very effective and intuitive visualisation of their structure.

CallStax attempts to alleviate some of the problems associated with large graph structures by greatly reducing the number of explicit relations. This solution is effective in reducing graphical complexity, however, for large systems the mental load of identifying identical representations distributed through 3D space becomes much greater. The provision of a filtering and query mechanism aids greatly in this respect, allowing the user to quickly identify identical representations and their neighbours. In theory, CallStax is able to cope with very large graph structures, however the size and number of stacks also grows very rapidly. Similarly the number of individual and distinctive representations needed becomes greater. As identified earlier, the creation of such representations is particularly difficult within large systems.

FileVis provides perhaps the most scalable visualisation described here. The lack of any explicit relationships between software objects, and the simple mapping from software system to visualisation provides plenty of scope for expansion. The biggest issue to address would be the layout of the various files and functions, though as already mentioned it is hoped that this layout would only need to be

simple, being manipulated manually by the user as they begin to understand the system. Although the file platforms use distinct colours to aid their identity, there is sufficient scope for using various patterns to maintain this distinctive appearance. In short, FileVis should support scalability from small to large scale software systems. Additionally, the ability to move smoothly from a low-detail overview to a high-detail specific view of the software system provides scalability within the visualisation. This makes FileVis particularly suitable for visualising any size of software system.

SoftCity and SoftCities maintain some of the advantages of FileVis, enabling increased size of the software system to be visualised. It should be noted that both SoftCity and SoftCities view the system at a fixed level of abstraction. It is not possible to move in for a detailed view of a function structure, as in FileVis. For this reason, both SoftCity and SoftCities would be of little value for visualising small software systems. SoftCities excels in its ability to be applied to very large software systems and highlight the location and details of any change over successive revisions of the software.

6.5 Evaluation against desirable properties

This section presents a more detailed evaluation of the various visualisations and representations within FileVis. The desirable properties described in Chapter 4 are used as a framework for the evaluation, assessing how each aspect of the visualisations and representations addresses these properties. This evaluation is intentionally qualitative, giving only an impression of the strengths and weaknesses of a particular visualisation or representation. The applicability of a particular representation or visualisation is entirely subjective and dependent on the context of use. It would therefore be inappropriate for an evaluation to give a definitive statement as to whether a visualisation or representation is suitable or not. The wider picture must be considered. This evaluation against the desirable properties aims to allow such a subjective assessment to be made.

Within the FileVis visualisation as a whole there are many individual visualisations and representations, particularly as some must be considered from both viewpoints. A small set of each is selected for evaluation, concentrating on the most prominent visualisations and representations that provide a representative sample of FileVis as a whole. The various components within FileVis must first be identified as being either visualisations or representations, or indeed a mixture of both dependent on context. Table 6-A lists all the visualisations within FileVis, showing the corresponding representations used within each.

The set selected for evaluation includes the *FileVis*, *files* and *high-detail function* visualisations, and also the *Files*, *file identifiers*, *high-detail* and *low-detail function* representations. These seven items represent a good cross-section of the visualisations and representations used within FileVis. An evaluation will be made as to how each of these visualisations and representations addresses the desirable properties.

Visualisation	Representations
FileVis	Files Functions (high + low detail) CallStax
Files	Functions (high + low detail) File identifier
Functions (low detail)	Function tower Function identifier
Functions (high detail)	Control structure Complexity measures Revision measure Function identifier Back plane Ground plane
CallStax	Individual stacks Function blocks

Table 6-A. Breakdown of FileVis into visualisations and representations.

6.5.1 Evaluating the visualisations

Table 6-B, Table 6-C, and Table 6-D list each of the desirable properties of visualisations against the three visualisations selected from FileVis. Within these tables, short descriptions of the suitability of a visualisation to each of the desirable properties are given. These are then summarised in Table 6-E to provide a simple assessment and comparison of the strengths and weaknesses possessed by each visualisation.

Figure 6.2. Overview of the FileVis visualisation.

Figure 6.2 provides an overview picture of FileVis. The visualisation displays the files, functions and source code structure of a software system, as well as a collection of software metrics. Table 6-B provides an assessment of how FileVis addresses each of the desirable properties of a visualisation.

Properties	FileVis
Simple navigation with minimal disorientation	Layout is simple, but could prove complex for larger systems. Labelling of filenames would be an advantage. The visualisation remains planar to remove the need for extensive vertical navigation. Hub structure with easily identifiable central feature aids navigation. A loss of context occurs when viewing rearward files and functions due to their fixed orientation.
High information content	FileVis provides a reasonably high amount of 'at-a-glance' information. All the files within the system are apparent, including the information in the files visualisations. Other information such as the file dependencies is also readily available. Function distribution, length and complexity is intuitively readable.
Low visual complexity; well structured.	Visual complexity is fairly high but is scaled depending on viewer interests. Information clutter is kept to a minimum. Structure is dependent on layout, which in this case is a logical, hub formation.
Varying levels of detail	Features such as the CallStax can be adjusted to vary detail levels. Function representations also vary in detail automatically depending on viewer interests.
Resilience to change	This is dependent greatly on the layout algorithm used. The lack of explicit connections between representations provides flexibility in positioning, enabling greater resilience to effects of change. The current layout provides poor resilience.
Good use of visual metaphors	Use of metaphor is limited, with a lack of any real-world metaphors. The information is readily interpreted after initial instruction.
Approachable user interface	The interface is dependent on the VR package used. In this case the control method is very effective with practice. Additional tool support is provided allowing the user to vary the control method. FileVis makes use of a web-browser interface providing a common and well understood front end.
Integration with other information sources	The visualisation is integrated well with a variety of textual information on the software system. Integration is bi-directional, allowing the user to explore the software using whichever interface is preferable at any stage.
Good use of interaction	Interaction is limited to selection of objects for additional information.
Suitability for automation	All aspects have been designed to enable automatic generation. All features have a regular, procedural construction with layout being the only issue. Layout is also eased due to the lack of explicit connections between objects.

Table 6-B. Assessing FileVis against the desirable properties of a visualisation.

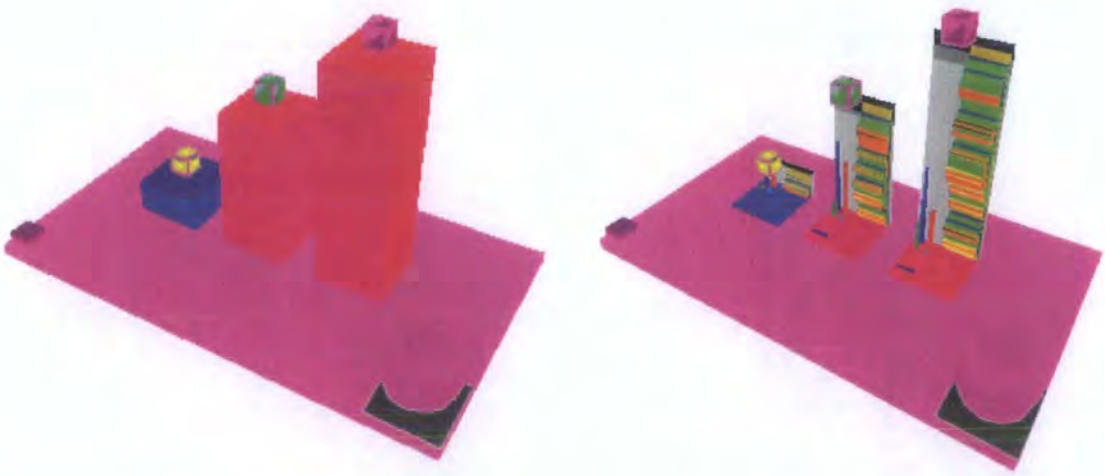


Figure 6.3. Low and high detail file visualisations.

Figure 6.3 shows examples of a low and high-detail file visualisation. This consists of a file platform with identifier, which is then populated with a number of function representations. The function representations switch between low-detail (left) and high-detail (right) depending on viewer proximity. Table 6-C provides an evaluation of the file visualisation against the desirable properties of a visualisation.

Properties	Files
Simple navigation with minimal disorientation	A very simple layout is used for the placement of functions, although one undesirable feature is that the position of functions may vary over versions of the software system. The layout in the prototype is arbitrary and a more appropriate scheme could be implemented. Labelling of functions would provide a big aid to identification. The file identifier provides a visible, fixed landmark.
High information content	The files show a great deal of information, which is readily apparent. This mainly concerns details of the functions within that file.
Low visual complexity; well structured.	At a greater viewing distance the files are low in visual complexity and readily interpreted. At closer distances visual complexity increases greatly, possibly leading to confusion. The population of the files is well structured.
Varying levels of detail	The function representations vary in detail, which play a predominant role in the visual complexity of the file visualisations.
Resilience to change	The files are not resilient to change. The current layout positions function representations depending on function length (height). Any changes may affect their positioning which may prove disorientating. Again, this is a layout issue.
Good use of visual metaphors	The files visualisation with low-detail function representations are based upon a 'bar-chart' metaphor which is readily understood.
Approachable user interface	Not Applicable. (Covered by FileVis as a whole).
Integration with other information sources	Not Applicable. (Covered by FileVis as a whole).
Good use of interaction	Interaction is limited to the selection of functions and files which provides additional information. This is presented in the 2D HTML frames.
Suitability for automation	All aspects have been designed to enable automatic generation, including regular structures and placement.

Table 6-C. Assessing the files within FileVis against the desirable properties of a visualisation.

Figure 6.4. High-detail function visualisation.

Figure 6.4 illustrates a single high-detail function visualisation, similar to those populating the file platform shown in Figure 6.3. This visualisation provides information on the function code structure, textual structure, complexity metrics and number of revisions. Table 6-D provides an assessment of how the high-detail function visualisation addresses the desirable properties of a visualisation.

Properties	Functions (high detail)
Simple navigation with minimal disorientation	Navigation is not an issue here, as the function visualisations are not meant to be explored internally.
High information content	A reasonable amount of statistical information regarding the function is instantly available. This is more of an overview than a detailed visualisation of the function code. Information content is relatively high.
Low visual complexity; well structured.	Static features lead to instantly recognisable and interpreted information. The code structure aspect of the visualisation is, however, cluttered and therefore harder to interpret. Visual complexity is relatively high.
Varying levels of detail	The high detail visualisation does not vary any further in detail. However, it is replaced entirely with a low-detail representation as the viewer moves away.
Resilience to change	Change in the visualisation is linked directly to changes in the function code. No unnecessary instability is introduced. Resilience to change is high.
Good use of visual metaphors	The high detail function visualisation is not based upon any metaphors. Only the metric bars present a readily understood metaphor - that of a bar chart.
Approachable user interface	Not Applicable. (Covered by FileVis as a whole).
Integration with other information sources	Not Applicable. (Covered by FileVis as a whole)
Good use of interaction	Interaction is limited to the selection of functions, providing additional information in the 2D HTML frames.
Suitability for automation	All aspects have been designed to support automatic generation. All structures are of predeterminate size and are regularly placed.

Table 6-D. Assessing the high-detail functions within FileVis against the desirable properties of a visualisation.

The evaluations given in Table 6-B, Table 6-C and Table 6-D can be summarised to provide a simple overview of how each of the visualisations rates against the desirable properties. This summary is shown in Table 6-E. The summary uses a subjective scale, which indicates how well each visualisation addresses a particular property. If a property is not applicable, or is related to some other visualisation or representation then this is also stated. The subjective scale rates each visualisation or representation against the desirable properties using one of four grades: excellent, good, fair or poor.

Properties	FileVis	Files	Functions (high Detail)
Simple navigation with minimal disorientation	Good	Good	Not Applicable
High information content	Good	Good	Good
Low visual complexity; well structured.	Good	Excellent to Poor (low and high detail functions)	Fair
Varying levels of detail	Good	Good	Not Applicable
Resilience to change	Poor	Poor	Good
Good use of visual metaphors	Poor	Good	Poor
Approachable user interface	Fair	Not Applicable	Not Applicable
Integration with other information sources	Good	Covered by FileVis	Covered by FileVis
Good use of interaction	Poor	Poor	Poor
Suitability for automation	Good	Good	Good

Table 6-E. Summary showing strengths and weaknesses of each visualisation.

It is apparent from examining Table 6-E that the three visualisations selected perform well against the desirable properties and have a mixed coverage of essential areas. One area that lets down both FileVis and the Files visualisation is their resilience to change. In both cases though, this is dependent on the layout algorithm used and, in this prototype system, the layout used was exceptionally simple. Selection of a different layout criterion or maintaining object positions between revisions of the visualisations would help considerably. The intention of the visualisation is to support manual placement by the user, enabling them to position and group software objects with some higher-level semantic meaning which is difficult to capture using automatic layout algorithms. In such an instance, the initial layout algorithm used need only be simplistic.

One area of note is the very poor support for interaction throughout the three visualisations. Interaction in the FileVis prototype is provided by a simple 'click here' interface. Further development of FileVis would need to address this deficiency and introduce deeper interaction techniques, including features such as user annotation and manipulation of the visualisation. In general, the visualisations perform well in addressing the desirable properties. Important deficiencies have also been highlighted.

6.5.2 Evaluating the representations

Evaluation of how the four representations selected address the desirable properties of a representation are given in Table 6-F, Table 6-G, Table 6-H and Table 6-I. Within these tables, short descriptions of the suitability of a representation to each of the desirable properties are given. These are then summarised in Table 6-J to provide a simple assessment and comparison of the strengths and weaknesses possessed by each representation.

Figure 6.3, on page 147, provides an example of a low and high-detail file representation. An assessment of how these representations address each of the desirable properties of a representation is given in Table 6-F.

Properties	Files
Individuality	Attributes such as the representation's colour, size, identifier, number of functions and profile of the functions aid individuality. The file identifier and location provide absolute individuality. Providing a textual label with the file name would prove beneficial
Distinctive appearance	All the features mentioned above make for a distinctive appearance. Additionally, the files derive character from their structure and contents.
High information content	The representation embodies a high amount of information regarding the functions within the file. This is particularly the case when viewed with the high-detail function representations.
Low visual complexity	Complexity is low when low-detail functions are present, but suffers from increased complexity with high-detail functions.
Scalability of visual complexity and information content	No scalability of the file representation itself. Detail control is provided by the function representations. However, when viewed as one representation, this feature provides the files with a very scalable nature.
Flexibility for integration into visualisations	Positional freedom is preserved, however, all intrinsic dimensions are used leaving little room for encoding additional information in the visualisation. The overall size of the representation is also variable.
Suitability for automation	Designed to support automation through regular sized and placed constructs.

Table 6-F. Assessing the files within FileVis against the desirable properties of a representation.

Figure 6.5. A number of function identifier representations.

Figure 6.5 illustrates a number of function identifiers which are used extensively throughout the FileVis visualisation. Each function in the software system is assigned a unique function representation, this representation is then used within both the 3D visualisation and the 2D documentation to provide a correlation between the different views. Each identifier consists of a cube with a patterned face and coloured border. The colour of the border matches the file in which the function is defined. The pattern and colour of the face is assigned to be unique to a particular function when combined with the border colour. Table 6-G provides an assessment of how the function identifiers address each of the desirable properties of a representation.

Properties	Function Identifiers
Individuality	Each representation is guaranteed to be unique from all others. This unique identity is provided by a coloured, patterned face and a coloured cube border. Individuality could be enhanced further by provision of a text label with the function name.
Distinctive appearance	Function identifiers appear distinctive due to the variety of patterns and colours used. Distinctive appearance is occasionally compromised with certain similar colours or patterns between identifiers.
High information content	Information content is minimal. The only information provided is an indication as to which file a function belongs to, by virtue of the coloured cube border. Information content is intentionally low to minimise visual complexity.
Low visual complexity	Visual complexity is low. Function representations consist of two elements - a patterned face and a coloured border.
Scalability of visual complexity and information content	There is no scalability of visual complexity or information content. This could be supported in future by providing textual function labels as the user approaches, removing them as they withdraw to reduce visual clutter.
Flexibility for integration into visualisations	Flexibility is very good with the identifiers having a fixed size and a simple shape. Several intrinsic dimensions remain unused allowing the visualisation to encode further information as necessary.
Suitability for automation	Suitability for automation is good, again the fixed size and shape aid in this respect. The patterns and colours are selected from a predefined library, requiring a 1 to 1 mapping to functions to ensure individuality.

Table 6-G. Assessing the file identifiers against the desirable properties of a representation.

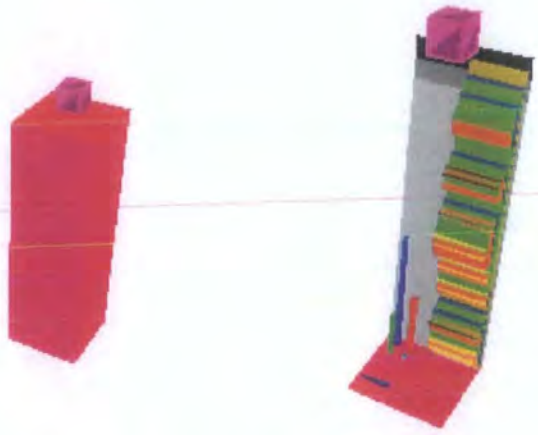


Figure 6.6. Low and high-detail function representations.

Figure 6.6 shows a low and high-detail function representation. The low-detail representation (left) provides only very basic information on the function, such as relative complexity, length and a unique identifier. The high-detail representation (right) provides considerably more detail, including further metrics and a view of the function code structure. Table 6-H and Table 6-I provide an assessment of how each of these representations addresses the desirable properties of a representation.

Properties	Functions (low detail)
Individuality	Only the function identifier offers true individuality. Height and colour of the tower are of little help.
Distinctive appearance	Appearance is not distinctive. Function identifiers are unique but may possibly appear similar without a closer inspection.
High information content	Information content is intentionally very low. This serves to provide only the most salient information when the user is gaining an overview of the software system.
Low visual complexity	Visual complexity is intentionally very low to minimise visual clutter within large collections of similar representations
Scalability of visual complexity and information content	Switching between high and low-detail function representations within the visualisation provides scalability of visual complexity and information content.
Flexibility for integration into visualisations	Positional freedom is preserved. Base dimensions are fixed, aiding placement and grouping. Height is variable, which may prove restrictive. Overall, flexibility for integration is good.
Suitability for automation	Designed to support automation through a simplistic structure.

Table 6-H. Assessing the low-detail functions within FileVis against the desirable properties of a representation.

Properties	Functions (high detail)
Individuality	Only the function identifier and code structure offer true individuality. All other features may be similar between representations.
Distinctive appearance	Appearance is not distinctive. Function identifiers are unique but may possibly appear similar without a closer inspection. The function structure helps to provide character to each function, often aiding in providing a distinctive and memorable appearance.
High information content	High information content about each function is available, including a variety of software metrics and details of the function code structure.
Low visual complexity	Visual complexity is fairly high, although the majority of features are well structured and clearly interpretable.
Scalability of visual complexity and information content	Switching between high and low-detail function representations within the visualisation provides scalability of visual complexity and information content.
Flexibility for integration into visualisations	Positional freedom is preserved. Base dimensions are fixed, aiding placement and grouping. Height is variable, which may prove restrictive. Overall, flexibility for integration is good.
Suitability for automation	Designed to support automation through a clear structure with regular sized and shaped objects.

Table 6-I. Assessing the high-detail functions within FileVis against the desirable properties of a representation.

The information presented in Table 6-F, Table 6-G, Table 6-H and Table 6-I can be summarised to provide an overview of how each of the representations compare against the desirable properties. Table 6-J provides this overview and enables the identification of the strengths and weaknesses of each representation, as well as highlighting how related representations compliment each other. This summary uses the same subjective scale as previously mentioned while evaluating the visualisations.

Properties	Files	Function Identifier	Functions (LD)	Functions (HD)
Individuality	Excellent	Excellent	Excellent (via identifier)	Excellent (via identifier)
Distinctive appearance	Good	Fair	Poor	Fair
High information content	Good	Poor	Poor	Good
Low visual complexity	Good to poor (varies)	Good	Excellent	Fair
Scalability of visual complexity and information content	Good (high and low detail function representations)	Poor	Good (linked to high detail version)	Good (linked to low detail version)
Flexibility for integration into visualisations	Poor	Good	Fair	Fair
Suitability for automation	Good	Good	Good	Good

Table 6-J. Summary showing strengths and weaknesses of each representation.

It can be observed from Table 6-J that in general, each of the representations performs very well, though all have a number of weaknesses and deficiencies. Of particular interest is how the low-detail and high-detail function representations compliment each other's strengths and weaknesses. This is particularly evident for the properties of distinctive appearance, high information content and low visual complexity. This enables the visualisation to provide the optimum amount of information to the user at each stage. When gaining an overview, low-detail representations are used and information content is reduced to present only the most salient details, while visual complexity is reduced to minimise display clutter. When focussing on a particular item of interest then high-detail representations are used thereby providing maximum information content when visual complexity is less critical due to the low number of representations on screen.

Also of note is that the function identifier is present in each of the other three representations. The sole purpose of the function identifier is to provide individuality. This individuality is then imparted onto the other visualisations in which it is used.

6.6 Cognitive design elements for software exploration

Storey *et al* [Storey97] present a hierarchy of cognitive design elements which are applicable to the design and evaluation of what they call 'Software Exploration' tools. Software exploration tools are described by the authors as tools which "... provide graphical representations of software structures linked to textual views of the program source code and documentation". There is a clear relationship with the research work produced within this thesis. The goal of the work here is to produce structural visualisations of software systems while providing a strong correlation with existing information and documentation on each system. The prototype visualisation FileVis, described in Chapter 5, epitomises these goals in the development of a software exploration tool.

This section will evaluate how the FileVis visualisation addresses these cognitive design elements, providing evaluation of its suitability as a software exploration tool. The other methods of evaluation described in earlier sections are all concerned with evaluating the attributes of a visualisation, and not necessarily as a tool to aid program comprehension. The following evaluation attempts to address this shortcoming by evaluating against a set of cognitive design elements, which are specifically aimed at program comprehension. Although the main subject of this evaluation is the FileVis visualisation, occasional reference will also be made as to how certain cognitive elements apply to the other visualisations described in Chapter 5, where relevant.

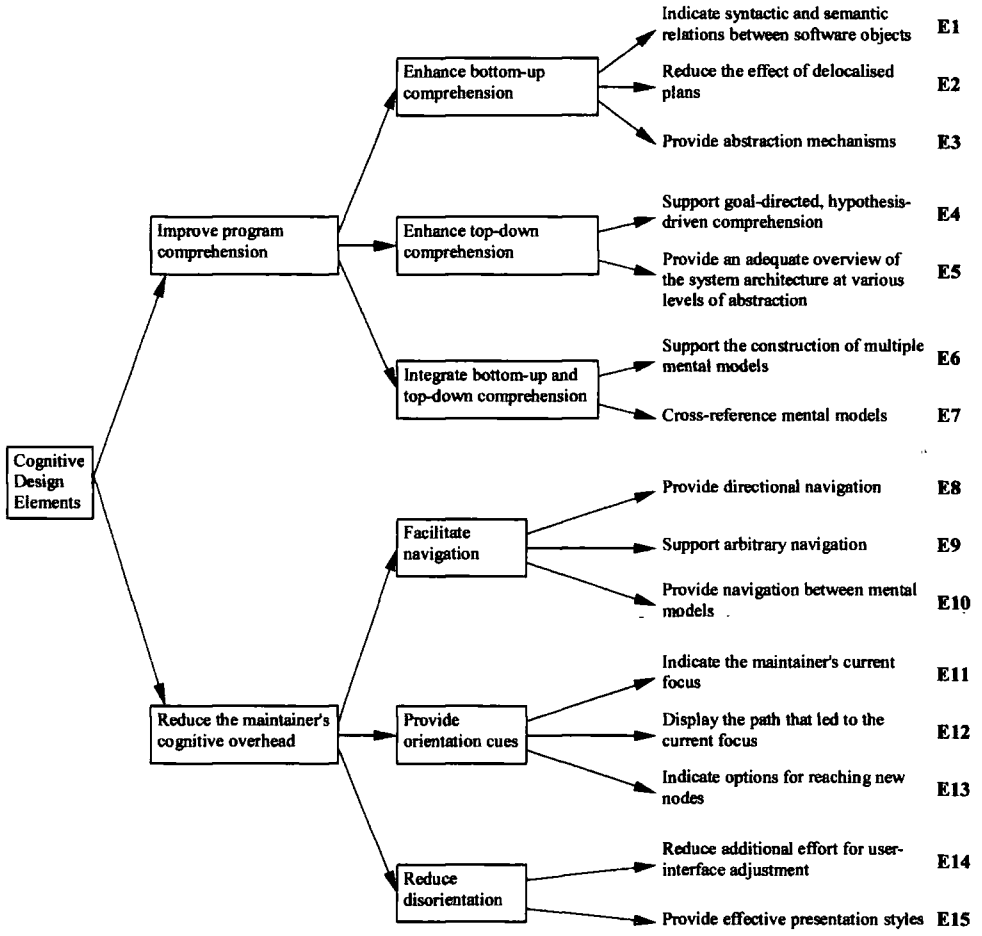


Figure 6.7. Cognitive design elements for software exploration.

The cognitive design elements are organised into two branches, the first of which deals with improving program comprehension while the second is concerned with reducing the maintainer's cognitive overhead. Figure 6.7 shows the hierarchy of cognitive design elements. Under the first branch, 'Improve Program Comprehension', the intention is to capture the essential processes of the various comprehension strategies. Cognitive design elements E1 to E7 fall under this remit. The second branch, 'Reduce the Maintainer's Cognitive Overhead', addresses the cognitive issues of a maintainer while browsing and navigating a visualisation of the program structure. Cognitive design elements E8 to E15

are considered within this branch. This hierarchy of cognitive issues, E1 to E15, is derived through an examination of the various cognitive models of program comprehension which were discussed in Chapter 2. Each of the cognitive issues will now be described in detail, with reference to their applicability to this research.

6.6.1 Improve program comprehension

It is argued that the comprehension strategy employed by a maintainer is dependent on a variety of factors governed by the experience of the maintainer, the software system under scrutiny and the type of maintenance task undertaken. It would be advantageous for a software exploration tool to support a wide range of comprehension activities and models. Presented below is a list of cognitive design elements extracted from various program comprehension strategies (Chapter 2) and discussed by Storey *et al* [Storey97], then further investigated by Chan [Chan98].

I. Enhance bottom-up comprehension

Storey *et al* argue that a bottom-up comprehension involves reading program statements and constructs and chunking them into higher level abstractions. This is repeated until an overall understanding of a program is obtained. Bottom-up comprehension involves three main activities:

- Identifying program items, such as variables, statements, functions, files and the relationships between them.
- Browsing code in delocalised plans.
- Building abstractions (through chunking) from lower level items.

A software exploration tool to assist in bottom-up comprehension should address these three activities. Each of these activities is addressed in cognitive elements E1 to E3, which are described in greater detail here.

E1. Indicate syntactic and semantic relations between software objects

Storey *et al* suggest that the syntactic and semantic relationships are essential during a bottom-up comprehension. The syntactic relation can be derived from the source code by systematically identifying a set of program units. The semantic relation can be attained by an analysis of the relationships between these program units.

Both syntactic and semantic relationships are highlighted in FileVis and the other visualisations. Within FileVis, syntactic relations shown include identification of various program components such as files, functions, and various types of code block, such as loops and conditional statements. The relationships

between these objects are shown both explicitly and implicitly. The provision of various software metrics enables hypotheses to be drawn about the function of, and semantic relationships between, various program components.

E2. Reduce the effect of delocalised plans

A delocalised plan results from the fragmentation of source code related to a particular algorithm or program plan. Reading fragments of code belonging to a delocalised plan can be cumbersome, requiring frequent switching between files which may lead to a feeling of disorientation. Static tools such as program slicing, whether graphical or as the result of some filtering, can help identify code belonging to a delocalised plan.

Only FileVis and CallStax provide any comprehension support for delocalised plans. In general, the visualisations are aimed at static, structural visualisation and do not contain the necessary detail required to view program plans. CallStax does, however, provide a slicing mechanism for understanding program paths. This information can aid the user in identifying the possible location of delocalised plan segments, as well as providing probable impact paths for any change. Additionally, the coloured function structure shown in FileVis enables an overview of the key components of a function code to be viewed. This enables the identification of possible plan components and beacons, such as loops and conditional statements. Certain plans can be identified visually using this aid, enabling a more focused investigation of the source code.

E3. Provide abstraction mechanisms

Storey *et al* believe that the process of building hierarchical abstractions from low level software objects is the most difficult task during bottom-up comprehension. Facilities should be available to allow the maintainer to create their own abstractions and label and document them to reflect their meaning. One possible technique would be to allow the aggregation of low-level program components into a higher level abstraction.

Software abstraction is provided directly in visualisations such as FileVis, which provide a direct, scalable visual abstraction of the software structure, from low-level software objects such as lines of code through to high-level objects such as files and the system itself. None of the visualisations described in this research support user interaction such as annotation or manipulation of the visualisation structure, although FileVis and CallStax are designed to support such enhancements. These visualisations possess minimum explicit relationships between software objects, enabling a vastly greater flexibility for placement and grouping. Other features such as annotation would prove a very powerful comprehension tool and could be integrated into these visualisations relatively easily.

II. Enhance top-down comprehension

Storey *et al* believe that top-down comprehension requires application domain knowledge or previous exposure to the software system. The maintainer formulates hypotheses then reads the code in a depth-first manner to verify or reject these hypotheses. Support for top-down comprehension can be provided by:

- Supporting the recording of hypotheses and linking them to relevant parts of the program, as well as supporting the refinement of hypotheses.
- Providing overviews of the software system allowing the maintainer to explore its structure in a top-down manner.

Cognitive elements E4 and E5, which support top-down comprehension, are described in more detail below.

E4. Support goal-directed, hypothesis-driven comprehension

A software exploration tool should provide the capability to create, record and relate hypotheses or annotations to specific parts of the software system. As well as providing support for hypothesis verification such annotations would prove invaluable during future maintenance or for co-operation and communication between maintainers.

None of the visualisations described in this research support user interaction such as annotation of the visualisation structure. The multiple cameras used within FileVis do allow a maintainer to effectively explore a number of different routes through the software, swapping between these views as necessary. This enables the user to 'bookmark' their position and explore a particular hypothesis, with the ability to observe this point of divergence and to jump back to it at any time.

E5. Provide an adequate overview of the system architecture at various levels of abstraction

To explore programs top-down, access to the software architecture should be provided at various levels of abstraction. This allows the maintainer to systematically explore the program structures in a top-down fashion.

Visualisations such as FileVis and SoftCities directly support top-down exploration of a software system. The user is initially presented with an overview of the entire software system and is able to delve deeper to lower levels of abstraction, obtaining more specific information as they progress. At each stage of the visualisation it is apparent at what level of abstraction the user is currently at, and what the possible opportunities are for moving further in.

III. Integrate bottom-up and top-down approaches

Storey *et al* acknowledge that a maintainer will create and switch between various mental models during the course of comprehension. Relationships such as control flow, data flow and function abstractions are believed to be the key to the creation of these mental models. These relationships can be illustrated using a graph model. The integration of bottom-up and top-down approaches can be facilitated by supporting the construction and integration of various different views on the software system.

E6. Support the construction of multiple mental models

Storey *et al* state that not only do mental models differ in context and level of abstraction, but they also differ from one maintainer to another. Support should be given for the construction of multiple mental models. The authors suggest that various mental models of a program may be represented by using a multitude of both textual and graphical views on a software system.

FileVis provides excellent support for multiple views, and so multiple mental models, of a particular software system. The 3D visualisation in FileVis is just one of these views and is supported by detailed textual information on various software objects. The software visualisation can be explored using any of the views presented enabling the user to effectively move between mental models as needed.

E7. Cross-reference mental models

Maintainers frequently switch from one model to another in the course of comprehension. Storey *et al* believe that these switches are often the result of a maintainer mentally cross-referencing different mental models. This activity can be facilitated by supporting the cross-referencing of the corresponding representations between various views on the software system.

As mentioned above, FileVis provides excellent support for exploration of a software system using a multitude of mental models, both top-down and bottom-up. This support is provided by different views on the software system, both graphical and textual. These views are cross-referenced using graphical identifiers which provide visual relationships between objects in each view. Additionally, the user is able to interact with and interrogate the visualisation to explicitly highlight any relationships between views.

6.6.2 Reduce the maintainer's cognitive overhead

Storey *et al* agree that when comprehending large software systems, the maintainer's cognitive overhead increases rapidly. Visualisation tools are often supplied in an effort to reduce this cognitive overhead. In order to be successful, visualisation tools must provide good navigation facilities, meaningful orientation cues, and be effective in presenting the information so that it can contribute to

program comprehension. Storey *et al* refer to each of the above terms in the following context. Navigation provides the facilities to go from one place to another. Orientation cues show the user where they are currently, how they got there and how to go somewhere else. Effective presentation techniques are used to alleviate the effects of displaying large, overwhelming amounts of information.

I. Facilitate navigation

Large software systems will invariably result in large software visualisations. When exploring large software systems, it is important that a maintainer is equipped with the appropriate facilities to ease navigation through the vast amount of information presented. Storey *et al* suggest that navigation facilities should include mechanisms for browsing source code, program documentation, graphical views of software structure and documented mental models of the program. Cognitive elements E8 to E10 address the issues of navigation and are described below.

E8. Provide directional navigation

Storey *et al* describe directional navigation as mechanisms for reading source code and program documentation sequentially, browsing the software using data-flow and control-flow relationships, traversing software structure in hierarchical abstractions and by following user-defined program or application dependent links.

FileVis makes use of web-browser technology to implement a prototype software exploration tool. This technology provides built-in aids for browsing text documents such as source code and program information. The glue that binds together the various views of a software system is provided by a proliferation of hypertext links. The user is able to move smoothly from document to document, document to visualisation and visualisation to document. At each stage the user is made aware of the possible options for navigation, with hypertext links visibly highlighted and graphical icons in the text illustrating a relationship to the 3D visualisation.

E9. Support arbitrary navigation

Arbitrary navigation is supported by allowing a maintainer to navigate to locations not necessarily reachable by following an application or user-defined link.

Arbitrary navigation is supported in FileVis primarily using the 3D visualisation window. Through this interface the user is able to view the entire software system and navigate freely to any point within it. There are no fixed links or paths to follow, the user has complete freedom to choose their route and destination. The information-rich environment of FileVis provides sufficient information for a user to decide which aspects of the software system to investigate. Further arbitrary navigation is supported in the textual source code window. When a program object such as a function is selected, the source code window displays the corresponding source file positioned at the beginning of the function code. The

function code is highlighted with a darker text colour, but the user is able to freely browse the remainder of the source file and follow links within other functions as they desire.

E10. Provide navigation between mental models

The key to successful comprehension is believed to be the ability to navigate smoothly between various mental models. In the context of a software exploration tool, this equates to a smooth navigation and correlation between various views on the software system. Storey *et al* argue that this is a non-trivial problem as there may be one-to-many and many-to-one links from one model to another.

As previously mentioned, FileVis directly supports the creation of multiple mental models through multiple views on a software system. Both implicit and explicit relationships between these views are available and the user is constantly made aware of possible navigation paths through and between views. Graphical identifiers support correlation between the various views.

II. Provide orientation cues

Orientation cues are described as indications to the maintainer as to where they are currently exploring within the software structure, how and why they are there, and how to move to a different focus or interest.

E11. Indicate the maintainer's current focus

During comprehension and depending on the task at hand, a maintainer may be interested in viewing source code for a function, examining a diagram that describes some of the program's functionality or browsing a set of documentation. The focus of interest may be fragmented as the maintainer tries to understand non-local interactions in the code. Storey *et al* believe that the use of judicious orientation cues can reinforce the maintainer's sense of focus and orientation. Indicating the maintainer's current focus requires both showing the artefacts that are of immediate interest, but also displaying context for those artefacts.

Both focus and context are highlighted well within FileVis. The 3D visualisation allows the user to select any amount of contextual information by simply positioning themselves at different distances from an object, thereby providing more or less background information surrounding the object. The centre of the viewpoint acts as a point of focus for the user, much the same as the centre of one's eyesight provides indication of interest in the real world. This simple concept of focus is backed up by the user being able to select program objects and have them visually highlighted. They can then explore surrounding, contextual information while having a positive indication of their original interest. Also, within the detailed textual information display of currently selected objects, any links to related objects are presented alongside summary information. This enables the user to gain some initial insight into the linked object and judge whether or not it warrants further investigation.

E12. Show the path that led to the current focus

Recording why a maintainer is interested in a particular program object may be very important. Often, the reason for inspecting a particular item may be the result of verifying a hypothesis, or because the item is known to be critical to the maintenance task. The maintainer should be supported with facilities to show the sequence of actions or the direction taken to reach the current focus.

Displaying the path which led to the current focus is considerably more difficult to support within a 3D environment in which the user has total freedom. The only support provided by FileVis is the ability to move between multiple camera views and use these to 'bookmark' points along a path through the visualisation. The textual documents do not provide any support for viewing the path taken through them, although the web-browser provides a 'history' feature enabling the user to step back through the set of documentation viewed.

E13. Indicate the options for reaching new nodes

Given that a user is at a certain point in the exploration of a software system, this design element addresses not which facilities are available for further exploration, but rather how the user is made aware of these facilities. Further navigation options should be clearly identified.

Within the 3D visualisation of FileVis, the user is always aware of the possible options for exploration. Objects within the visualisation can be browsed and explored by simply moving towards them. Objects which can be selected for more detail using the textual windows are provided with a graphical identifier. Selecting this identifier displays more detailed information in the textual windows. Within the textual windows the possibilities for navigation are shown, with hypertext links visibly highlighted and graphical icons in the text illustrating a relationship to the 3D visualisation.

III. Reduce disorientation

The problem of disorientation is a major issue when exploring a large information space or large software visualisation. Storey *et al* suggest that disorientation can be alleviated by removing some of the unnecessary cognitive overheads resulting from poorly designed user interfaces, and by making use of specialised graphical views for presenting large amounts of information.

E14. Reduce additional effort for user-interface adjustment

Poorly designed interfaces will induce extra overhead. Available functionality should be visible and relevant and should not impede the more cognitively challenging task of understanding a program. Significant cognitive overhead may be introduced due to the disorientation caused by switching between different mental models or views on the software system.

FileVis makes use of web-browser technology for implementing the interface to the software exploration tool. This provides users with an instantly recognisable interface and familiar methods for navigating through the documentation and views on the software system. The 3D visualisation does, however, present a vastly different interface to that with which most users will be accustomed. Navigation through the virtual environment becomes natural with practice but will almost certainly lead to some disorientation on first use. The move between different views of the software system is intentionally smooth, with various views visible at any time.

E15. Provide effective presentation styles

For complex graphical representations, automatic layout algorithms are often used to display the representations in a more readable manner. Although software has no inherent shape or colour, a graph can be produced in such a way that it communicates key characteristics about the software.

The presentation style used is consistent throughout the FileVis prototype and is presented in a clear, well-structured manner. The source code is formatted and indented clearly, with the possibility for including syntax highlighting at a later stage. Detailed displays of information on selected software objects are consistently formatted and presented in a structured form. The 3D visualisation is also well structured and contains minimal graphic clutter.

6.6.3 Conclusions

It can be seen from the previous discussion that FileVis performs well against the cognitive design elements. In particular, it directly supports elements E1, E5 to E11, E13 and E15. These elements include support for top-down comprehension, bottom-up comprehension, construction of multiple mental models, navigation and orientation. Elements E3 and E4 are only weakly supported by FileVis but are integral within the visualisation design, enabling them to be supported relatively easily with future development. These elements address aspects of user-defined abstractions, manipulation and annotation of the visualisation, and are not currently supported by the limited interaction provided in FileVis.

The only elements which would be inherently difficult to support or address within FileVis are E2, E12 and E14. Element E2 is concerned with reducing the effect of delocalised plans which would require considerably more low-level detail within the visualisation. Element E12 requires the visualisation to show the path which led to the user's current focus. This is a particularly difficult feature to implement within a 3D environment in which the user has complete navigational freedom. Displaying a history of visualisation elements investigated by the user could provide some support, as well as a history of hyperlinks followed within the textual views. Finally, element E14 is concerned with reducing the additional effort for user-interface adjustment. Although FileVis does use the common and identifiable interface of a web browser, there will always be a relatively steep learning curve required for navigation

within the 3D virtual environment. Such navigation is relatively intuitive with practice but presents a significant challenge to the unfamiliar user.

It can be seen from the above summary that FileVis performs well against the cognitive design elements proposed by Storey *et al.* These design elements are intended to evaluate the effectiveness of a software exploration tool, specifically for the purpose of program comprehension. The evaluation presented in this section provides an initial indication that FileVis is suitable as a software exploration tool to aid program comprehension. Experimental evaluation of FileVis as a program comprehension tool would prove an interesting extension to this research work, with a particular focus on the limitations of the 3D environment and the interface familiarisation time required.

6.7 Summary

This chapter has presented an evaluation of the prototype visualisations developed within this research and described in Chapter 5. The evaluation has taken a number of forms, ranging from an informal discussion of the pros and cons of each visualisation, through to a more formal evaluation of FileVis against the desirable properties identified within this research. In order to provide evaluation of the prototypes as program comprehension tools, a further evaluation of FileVis was performed against a framework of cognitive design elements identified by Storey *et al* [Storey97].

Chapter 2 highlighted the need for both an evaluation framework and design guidelines to aid the effective application of software visualisation. In order to evaluate 3D visualisations and representations, this research has proposed an evaluation framework based upon the desirable properties identified in Chapter 4. This framework was used to evaluate the FileVis visualisation. It was found that FileVis performed well against this evaluation, focusing on a number of specific properties while providing broad support for many of the other desirable properties. It is also apparent from the evaluation that the different components of FileVis must behave both as visualisations and representations depending on their context. Chapter 4 highlighted the fact that, in such circumstances, it is necessary to find a good compromise between the desirable properties of each. It is evident from the evaluation that FileVis manages to achieve this compromise. The visualisations and representations of FileVis often displayed a bias towards certain desirable properties, but were paired so as to compliment the strengths and weaknesses of each other.

The informal evaluation (section 6.3) illustrates that the visualisations developed within this research each address different issues within the field of 3D software visualisation. This approach has provided a wide range of possible models for future visualisations and identified the benefits and possible pitfalls of each approach. The FileVis visualisation was highlighted as a particularly well-developed software visualisation, being subjected to a number of different evaluation techniques to arrive at this conclusion. The use of a framework of cognitive design elements establishes the suitability of FileVis as a software exploration tool. This showed the visualisation supporting a variety of program comprehension models

and enabling the user to obtain maximum information by exploring the software system using whichever technique proves most profitable at each stage.

Finally, this chapter has also highlighted the usefulness of the desirable properties, as an evaluation framework for assessing the strengths and weaknesses of visualisations and representations. This evaluation enables an objective assessment of properties which a visualisation or representation should support, though the overall suitability of a particular visualisation or representation is very subjective and dependent on the context of use. Considering the desirable properties does, however, allow the identification of particular strengths or deficiencies within a software visualisation thus enabling more focused improvement and modifications.

The following chapter provides the conclusions drawn from this research work and evaluates it against the criteria for success identified in Chapter 1.

Chapter 7.

Conclusions and summary

7.1 Introduction

This chapter presents a summary of this thesis and evaluates the success of the research work against the criteria defined in Chapter 1. Possibilities for further work and future directions are also discussed. The following section presents a review of the background of this work, summarising the introduction from Chapter 1.

7.2 Background

Visualisation provides a powerful analytical tool to aid understanding of complex systems, particularly enabling the identification of features, patterns and irregularities within such systems. Software systems are inherently abstract structures; they are information artefacts and have no physical form or intuitive appearance. For this reason, software systems are very difficult to visualise. Various theories of program comprehension put emphasis on the construction of a mental model of the software within the mind of the maintainer. These same theories hypothesise a number of techniques employed by the maintainer in the construction of this mental model. Software visualisation attempts to provide tool support for generating, supplementing and verifying this mental model or understanding.

Software visualisation is concerned with all aspects of generating and manipulating graphical models of a software system. Most software visualisation systems concentrate on producing two-dimensional representations and animations of some aspect of a software system. This research investigates the application of 3D graphics and virtual reality technology to software visualisation. The use of 3D addresses a growing need for more expressive powers to represent the inherently multidimensional data, facts and relationships, which constitute a software system. Humans possess naturally developed capabilities for navigating and interacting within three-dimensional environments. This research aims to maximise utilisation of such skills by creating software landscapes and immersing the user within them. Software landscapes are detailed, information-rich virtual environments in which the characteristics and attributes of a software system are mapped directly to features of the virtual world.

The field of information visualisation is concerned with the visualisation and representation of semantic information, in particular, large document collections and other such information structures. Information visualisation, as opposed to data visualisation, concentrates on the visualisation of more abstract components that have no inherent visual representations. A significant amount of research has

investigated the application of 3D graphics and VR technology to information visualisation, as documented in Chapter 3. Software systems can be considered as simply another collection of abstract data, files, facts and relations; essentially as information. Techniques and ideas generated from research into 3D information visualisation can be applied to the visualisation of software systems. Similarly, techniques and ideas identified within this research should be of interest to the more general field of 3D information visualisation.

The full potential of using three dimensional views and interaction techniques is far from being realised, however, a number of software visualisation systems have been developed to investigate its worth, as described in Chapter 3. In general, these systems do not make full use of the 3D environment or advance software visualisation to any great extent. The majority simply extend and adapt already established 2D visualisation techniques into 3D. Such techniques consist mainly of displaying a graph or network structure showing the relationships between components in a software system. These visualisations rarely extend beyond the simple node-link representations evident in almost all software visualisation systems to date. The field of software visualisation should not be restricted to solely one view of a software system, other possibilities must be investigated and the goal of this research is to do just that.

This research provides an investigation into the application of 3D graphics and VR technology to the visualisation of software systems. The goal of this research is to investigate other representations, visual abstractions and appropriate metaphors, which make full use of the 3D virtual environment afforded by cyberspace. Two aspects are addressed, the first being a theoretical look at the important issues of a visualisation and what properties define a good or bad visualisation. The second aspect provides a practical investigation of possible visualisation scenarios by discussing and evaluating a number of prototype software visualisation systems created throughout the course of this research.

The following section provides a discussion of the work presented within this thesis, identifying what was accomplished. This is followed by an evaluation of this research work against the criteria for success defined in Chapter 1. Finally, possibilities for further work and future directions of this research are identified.

7.3 Results

The overall aim of this thesis is to investigate the application of 3D graphics and VR technology to software visualisation, identifying new representations, visual abstractions and appropriate metaphors which make full use of the 3D environment afforded by cyberspace. The work described within this thesis succeeds in accomplishing this goal by providing both a theoretical and practical contribution.

Chapter 4 introduced the main theoretical work of this thesis. This described seven key areas of 3D software visualisation. These seven areas identify essential issues that must be considered when contemplating the design or evaluation of a 3D software visualisation. Definition of these areas

provides an important categorisation of the issues involved and allows more meaningful discussion and comparison of visualisations.

A further step was to redefine the notion of a visualisation into two discrete concepts, visualisations and representations, providing a definition of each term. This division is particularly important and aids considerably in the creation, evaluation and discussion of a 3D software visualisation, or in fact any 3D visualisation. The division essentially creates two concepts, the notion of a visualisation, which entails a collection of visualisation objects (representations), and a representation, which corresponds to a single object within the visualisation. This provides a framework for discussing the desirable features of both visualisations and representations, including the context of use for each and the transition between these two states.

Finally, identification of the desirable properties of both visualisations and representations was made, enabling identification of what constitutes good or bad features within a 3D software visualisation. These desirable properties provide a framework for the design of a visualisation but also allow a qualitative evaluation of an existing visualisation. Although the use of these properties does not allow for a definitive assessment of the quality or suitability of a visualisation as a whole, it does provide substantial insight into the strengths and weaknesses of a visualisation and highlights any predisposition towards a particular aspect of the visualisation. As an aid to design, these desirable properties provide a focus on important aspects of the visualisation, enabling the designer to plan a visualisation, which accentuates appropriate aspects while achieving a good compromise with other properties.

Chapter 5 presented the results and practical aspects of this research work by way of a number of visualisation prototypes and concept demonstrators. These prototypes add weight to the more theoretical work described above as well as providing a visible introduction as to what a 3D software visualisation may look like, the various options available and the issues involved. The prototypes presented address a number of issues with 3D software visualisation, including: the need to present strong relational information in an unobtrusive manner; the use of strong or weak metaphorical content; and the creation of an integrated visualisation which incorporates strong correlation to detailed documentation.

Chapter 6 provided an evaluation of these prototype visualisations using a variety of different approaches. This evaluation included an informal discussion of the visualisations presented and a more detailed evaluation of one particular visualisation. This detailed evaluation was performed using the desirable properties identified in Chapter 4 followed by evaluation against a framework of cognitive design elements. The evaluation illustrated the relative merits and deficiencies of each visualisation and identified that there are substantial benefits in the use of virtual reality for software visualisation.

The results of this thesis can be summarised as follows:

- Illustration of the possibilities afforded by 3D graphics for software visualisation and program comprehension.

- Definition of seven key areas of 3D software visualisation, namely: representation, abstraction, navigation, correlation, automation, interaction and scalability.

- Definition of the two terms, visualisation and representation, providing a distinct division within the concept of a software visualisation and enabling improved discussion of inherent properties.

- Identification of desirable properties for both visualisations and representations, providing support for the design and evaluation of a 3D software visualisation system.

- Development of a variety of prototype visualisations, each providing a different approach to the visualisation of software information. These prototypes also demonstrate the practicalities and feasibility of 3D software visualisation.

- Evaluation of these prototypes using a variety of evaluation techniques, thus providing a wide picture as to the merits and possibilities afforded by 3D software visualisation.

The following section will evaluate this research as a whole against the criteria for success defined in Chapter 1.

7.4 Evaluation against the criteria for success

The criteria for success, defined in Chapter 1, are reiterated here in addition to a brief evaluation of how this research has addressed each criterion. Criteria A and B are concerned primarily with the theoretical aspects of this research, while criteria C through F are concerned more with the practical aspects and the prototype visualisations produced.

A. *Identification of the key aspects of 3D software visualisation.*

The research work described in Chapter 4 explicitly defines seven key areas of 3D software visualisation. These areas have intentionally high-level descriptions but provide focus on aspects of 3D software visualisation, which are important when designing such systems. Further to this, definition of the terms visualisation and representation are presented, providing a conceptual division between the components of a visualisation. This division aids more focused discussion of the properties and features of a 3D visualisation, and helps to identify properties that are important in the creation of a successful 3D software visualisation.

B. *Development of a framework to aid in both the design and evaluation of a 3D software visualisation.*

The distinction between the concepts of a visualisation and a representation allow for a more detailed analysis of what constitutes a good or bad visualisation. This research has identified a number of desirable properties, which are conducive to a good visualisation. These desirable properties can be used as an aid to the design of new visualisations, but also as a framework for

evaluation of an existing visualisation. As an evaluation framework, the desirable properties cannot provide a definitive statement of whether a visualisation is good or not. They can, however, provide an objective view as to the strengths and weaknesses of a particular visualisation and identification of any compromise made between properties.

C. *Demonstration of the possibility for a move away from traditional graph-based visualisations of software systems, providing a more intuitive visualisation, which makes full use of the additional flexibility afforded by VR.*

This research has identified a need for visualisations that make full use of a virtual environment, and has demonstrated the practicality and feasibility of such an approach by way of a number of concept demonstrators. These prototype visualisations address a number of possible views on software information, ranging from an alternative and more flexible visualisation of graph structures, through to the construction of a full software landscape within which the user is submerged. Further investigation has also addressed the use of strong, real-world metaphors within these visualisations and identified issues involved with such an approach. The prototypes have demonstrated a definite possibility for dedicated 3D software visualisations, as well as highlighting many of the gains to be made through such an approach. Evaluation of FileVis against the cognitive design elements, E1 to E15, has shown that it is possible to create a viable tool to aid program comprehension using 3D graphics and virtual reality.

D. *The representations or abstractions used must be intuitive and easily understandable (with practice).*

The visualisations demonstrated within Chapter 5 all incorporate simple and often familiar concepts. Each visualisation is relatively easy to understand and use after only a short time spent with interface familiarisation (between different visualisations). The amount of information presented within each visualisation varies greatly. For example, FileVis provides an abstract, information-rich view of a software system, whereas SoftCities provides a more realistic metaphor-based approach with consequently lower information content. The full visualisation system demonstrated in FileVis makes use of a web-browser interface, providing a well known and intuitive front-end to the visualisation system. The use of a virtual reality system does, however, present a significant problem with interface familiarisation in general. As mentioned above, once familiar with the VR system, the interface varies little between visualisations. A new user to the VR system will, however, require some time learning the new interface and movement techniques, and will undoubtedly suffer frequent disorientation until proficient with the system.[2]

[2] This, however, is a problem associated with most VR systems.

E. *The visualisations should scale up satisfactorily, allowing visualisation of both small programs and large software systems.*

The prototype visualisations described in this thesis are predominantly handcrafted concept demonstrations. As such, it has not been possible to physically test the visualisations with large-scale software systems. The exception to this being Zebedee and SoftCities, which are both automatically generated visualisations. These have each been tested with medium-size software systems, up to approximately 25,000 lines of code, containing over 400 functions. Zebedee performed poorly, with the complexity of the visualisation increasing rapidly with the size of the software system or graph structure. SoftCities performed well, providing a visual indication to software evolution and areas of change between revisions. FileVis, although handcrafted, was designed specifically to view large-scale software systems and should cope reasonably well.

F. *A suitable level of automation, or the possibility for automation, must be demonstrated.*

As previously mentioned, the prototype visualisations demonstrated in Chapter 5 are predominantly handcrafted, however, each was designed expressly with capabilities and features to aid automation of the process. Zebedee and SoftCities are visualisations in which this automatic generation is actually implemented and provides a suitable demonstration of the feasibility of such an approach. Other visualisations include features, such as regularly sized and placed representations, which are conducive to automatic generation. One of the desirable properties identified in Chapter 4 highlights the issue of automation; each of the visualisations presented here addresses this property well.

From the above evaluation, it can be seen that this research has successfully addressed all of the criteria that were defined at the beginning of the project. The only weak areas which did not receive as much attention as originally planned are the issues of scale and automation, defined in criteria E and F. These two issues are closely related, because, to fully investigate the issues of scale it is necessary to visualise large-scale software systems. Unfortunately, this is not feasible unless the visualisations are generated through some automatic process. Creating a prototype visualisation that is fully automated, such as Software Cities, is a time consuming and limiting task. It was decided that the research effort was more profitably spent by investigating a wider variety of visualisation techniques, necessitating the manual creation of a number of prototype visualisations. The author believes that this approach has provided a broader insight into the possibilities of 3D software visualisation, while still having consideration for the issues of automation and scale.

7.5 Insights into 3D software visualisation

Throughout the course of this research, and in particular through the identification of the desirable properties and development of the concept demonstrators, various lessons have been learnt and observations noted. While many of these are not of direct relevance to the focus of this thesis, they are nevertheless useful insights into aspects of 3D software visualisation. This section highlights a number of these insights, referring, where possible, to the relevant pages of this thesis where these observations have been mentioned.

To reiterate, many of the statements listed here are simply observations and insights, they have not been experimentally or formally derived. The majority of these insights have been identified elsewhere within this thesis, this section simply serves as an amalgamation and focus of these points. The points listed below have been loosely grouped into three categories: general, automation and metaphor content.

7.5.1 General

- Within software systems there is a large variety of information which can often only be displayed within a two-dimensional view. It is inevitable that for some aspects of a software system it will be inappropriate to radically change the way in which the information is presented (Page 40).

- There are many possible visualisations for the numerous different aspects of a software system. There will inevitably be a high level of diversity, which will make it difficult to integrate these disparate views into a single, general-purpose visualisation of a software system (Page 80).

- Virtual environments are currently not very effective at displaying textual information (Page 109).

- 3D positional audio cues would provide a very powerful location mechanism for objects that may be out of immediate sight. These would aid considerably in directing the user's attention (Page 117).

- Software visualisation provides software objects with a visual, and often characteristic, appearance. The ability to identify such characteristics enables users to focus their attention quickly on areas of potential interest. This ability is one of the most powerful assets which software visualisation can provide (Page 120).

- The use of motion or rotation, if used sparingly, is particularly useful for highlighting objects and for commanding the user's attention (Page 133).

- Altering the dimensions of spatially distributed representations is not recommended unless accompanied by some static reference point, such as a ground plane. Doing so presents additional problems with depth perception and proves detrimental to understanding the structure of a

visualisation. It becomes extremely difficult to differentiate between the distance of a representation, and its size, thereby destroying any instinctive assumptions on the 3D structure and inducing disorientation (Page 133).

- The ability to manoeuvre within the 3D environment and to examine it from different angles contributes greatly to the comprehension of the structure (Page 100). Static images and screenshots of 3D visualisations cannot convey this ability.

7.5.2 Automation

- Allowing the user to 'build' the visualisation while investigating areas of the software system may prove more intellectually profitable than fully automating the process with the user then gaining their understanding from the completed visualisation (Page 85).

- The goal should be to use automation to provide a rough framework for the visualisation, and then allow the user to manipulate this as he or she explores, providing a deeper semantic meaning which is extremely hard to encapsulate automatically (Page 143).

- A lack of explicit connections between representations enables the user to position or group them freely, as they see fit, allowing them to impart some semantic structure to the visualisation. (Page 137).

- The layout algorithm used provides the dominant characteristics of the visualisation and contributes greatly to the ease or difficulty with which the structure can be understood (Page 134).

- Automatic clustering is often a very subjective approach due to the number of possible interpretations of 'close', both semantically within the information and also graphically within the visualisation (Page 47).

- When changes are made to the underlying software system, it is imperative that the new visualisation changes as little as necessary in order to reflect this (Page 85).

7.5.3 Metaphor content

- Metaphors are very effective tools for visualisation, but only remain so as long as the user's expectations are not violated. Breaking a metaphor can destroy any benefits gained from their use, adversely affecting comprehension of the visualisation. This restriction makes incorporation of additional information particularly difficult (Page 139).

- Metaphors are useful for encouraging comprehension of the environment and the information contained within it, whereas abstract constructs are useful for actually encoding information and offering maximum flexibility for the display of that information (Page 48). Using a strong real-

world metaphor will constrain the visualisation to displaying certain information in a particular way and may be over-restrictive for adding additional features (Page 91).

- Regardless of whether the visualisation is abstract or metaphor-based, there will always be a necessity to explain how the information is mapped onto the features of the visualisation. It is unlikely that a visualisation can be created which requires no initial instruction, regardless of the metaphor used.

- There may be many cases in which the mapping from software features to a metaphor will present problematic decisions. For example, the choice of viewing maintenance as a constructive or destructive activity. Different maintainers will have different views and expectations, making one choice of metaphor less appropriate than another (Page 123). There are inevitably a large number of metaphors which should be avoided due to conflicting individual interpretations of those metaphors, identification of such metaphors will not be easy.

- Ideally, a visualisation should aim to strike a careful balance between the use of metaphor and the use of more abstract constructs (Page 48).

7.6 Further work

The theoretical and practical work presented within this thesis has provided a framework for further investigation into the issues of 3D software visualisation. This research work has illustrated a definite need for further investigation into this potentially profitable area (in terms of program comprehension and software maintenance tasks), and has demonstrated the potential of such an approach. There are a number of future directions which could be investigated, this research providing a basis for such investigation. Several possibilities for further work will be highlighted here.

This research has essentially provided a first-step into the field of 3D software visualisation, that is, where the user is effectively immersed within the structure of a software system. At present the work here has concentrated purely on static, structural views of a software system and primarily considered single-user visualisations. Further work could investigate the issues involved with presenting more detailed or dynamic views of software operation. Details such as data usage, memory allocation, concurrent process interaction, and simple cross-referencing are all candidates for inclusion in the visualisation model.

There are many issues involved in the use of multi-user virtual environments. Such environments have obvious advantages within software maintenance, where it is typical for teams of software engineers to be closely involved in a project. Often the individuals within a project are geographically dispersed. Allowing shared use and interaction with a software visualisation and providing sophisticated communication features between users has the potential to be of great value. In the case of shared virtual environments, tools should be created to allow communication between users and other, more passive features such as identification of users. Even simple information is useful, such as the location

of a particular user, what they are currently looking at, and what action they are performing. There are many other issues involved with shared visualisations, the most difficult of which will be the provision of consistency throughout the visualisation, while still allowing users to exert some control over the structure and nature of the visualisation.

A further analysis of potential user tasks within a 3D software visualisation should be performed. Identification of these tasks, the requirements of the users and the possibilities for providing support within the visualisation are all important issues. During the course of this research a number of user tasks were identified. Further investigation and support for these tasks would be a logical progression of this work. The tasks identified were browsing, searching, interrogation, communication, modification, manipulation, customisation and annotation.

The visualisations demonstrated within this thesis all present relatively small software systems. There are many issues involved with the visualisation of significantly larger software systems; the seven key areas identified in Chapter 4 still being relevant here. The prototypes discussed in Chapter 5 address the areas of representation, abstraction, correlation, automation and scaling to a large degree, however the areas of navigation and interaction have received considerably less attention. These areas should be addressed in future work. Navigation is particularly important for the visualisation of large software systems, while good interaction is necessary in the development of virtual tools and usable visualisation systems.

Finally, in order to create usable 3D software visualisations, a comprehensive range of tools for use within this environment must be provided. It is foolish to assume that all standard views of software systems can be completely replaced using virtual reality. As mentioned earlier, software visualisations will not replace traditional views of software, they will work best as a supplement to those views. The goal should be to produce systems that require as little diverted attention from the user as possible. In other words, the fewer occasions in which the user must disengage from the 3D environment to consult information within another view, the better. In order to support this desire we must develop virtual tools for use within cyberspace which allow the user to access information, navigate within the visualisation, or even communicate with other users of a shared environment. Identification of the user's needs and methods for supporting these needs within the virtual environment would provide a valuable extension to this work.

7.7 Closing statement

This research has provided a valuable first-step towards the visualisation of software in cyberspace. Several essential issues in the design and evaluation of 3D software visualisations have been identified, from both a theoretical and a practical perspective. This research has shown that the application of 3D graphics and VR technology to software visualisation is not only possible, but also worthy of further study.

This research has succeeded in its goal, which was to investigate the application of 3D graphics and VR technology to the visualisation of software systems, and to identify new representations, visual abstractions and appropriate metaphors for use within the 3D environment.

Glossary of abbreviations

Abbrev-iation	Meaning	Description or relevant section / page (where applicable)
2D	Two-dimensional	
3D	Three-dimensional	
API	Application program interface	The interface (calling conventions) by which an application program accesses operating system and other services.
AR	Augmented reality	Section 3.2.4, "Types of VR systems", page 31.
CAD	Computer aided design	Section 3.2.1, "Virtual reality systems", page 26.
CD	Compact disc	An optically read, digital storage medium.
CSCW	Computer supported co-operative working	Software tools and technology to support groups of people working together on a project, often geographically distributed.
FDP	Force-directed placement	Section 3.3.1, "Self-organising graphs", page 43.
GUI	Graphical user interface	The use of pictures rather than just words to represent the input and output of a program.
HMD	Head-mounted display	Section 3.2.4, "Immersive VR", page 31.
HTML	Hyper-text mark-up language	A hypertext document format used upon the World-wide web.
IV	Information visualisation	Section 3.3, "Information visualisation", page 32.
LCD	Liquid-crystal display	An electro-optical device used to display digits, characters or images. Commonly used within head-mounted display interfaces for VR systems.
LEADS	Legibility for abstract data spaces	Section 3.3.2, "Legibility enhancement", page 60.
LOC	Lines of code	A software metric often used as an indication to the size of a software system.
OOI	Object of interest	Section 3.3.1, "Sphere visualisation", page 41.
PIT	Populated information terrain	Section 3.3.2, "Populated information terrains (PITs)", page 51.
POI	Point of interest	Section 3.3.2, "VR-VIBE", page 55.

UI	User interface	The aspects of a computer system or program which can be seen (or heard or otherwise perceived) by the human user, and the commands and mechanisms the user uses to control its operation and input data.
VE	Virtual environment	Section 3.2, "Virtual environments and VR systems", page 26.
VR	Virtual reality	Section 3.2, "Virtual environments and VR systems", page 26.
VRML	Virtual reality modelling language	A specification for the design and implementation of a platform-independent language for virtual reality scene description.
WoW	Window on world	Section 3.2.4, "Window on World (WoW) or Desktop VR", page 31.
WWW	World-wide web	An Internet, client-server, hypertext distributed information retrieval system which originated from the CERN High-Energy Physics laboratories in Geneva, Switzerland.

References

[Adam93] **J.A. Adam**, *Virtual Reality Is For Real*, IEEE Spectrum, Vol. 30, No. 10, pages 22-29, October 1993.

[Ball96] **T. Ball** and **S.G. Eick**, *Software Visualisation in the Large*, IEEE Computer, Vol. 29, No. 4, pages 33-43, April 1996.

[Basili82] **V.R. Basili** and **H.D. Mills**, *Understanding and Documenting Programs*, IEEE Transactions on Software Engineering, Vol. SE-8, No. 3, pages 270-283, March 1982.

[Benedikt91] **M. Benedikt**, *Cyberspace: Some Proposals*, In *Cyberspace: First Steps*, MIT Press, pages 273-302, 1991.

[Benford94a] **S. Benford** et al, *Experience of using 3D graphics in database visualisation*, Computing Department, Lancaster University, October 1994.

[Benford94b] **S. Benford** and **J. Mariani**, *Populated Information Terrains: Virtual Environments for Sharing Data*, Research report CSCW/4/1994, Centre for Research in CSCW, Lancaster University, 1994.

[Benford95] **S. Benford**, **D. Snowdon**, **C. Greenhalgh**, **R. Ingram**, **I. Knox** and **C. Brown**, *VR-VIBE: A Virtual Environment for Co-operative Information Retrieval*, Eurographics'95, 30th August - 1st September, Maastricht, The Netherlands, pages 349-360, 1995.

[Boyle93] **J. Boyle**, **S. Leishman**, **J. Fothergill** and **P. Gray**, *Development of a Visual Query Language*, Aberdeen University, 1993.

[Brooks77] **R. Brooks**, *Towards a theory of the cognitive processes in computer programming*, International Journal of Man-Machine Studies, Vol. 9, No. 6, pages 737-742, 1977.

[Brooks83] **R. Brooks**, *Towards a Theory of the Comprehension of Computer Programs*, International Journal of Man-Machine Studies, Vol. 18, No. 6, pages 543-554, 1983.

[Brooks87] **F.P. Brooks Jr.**, *No Silver Bullet: Essence and Accidents of Software Engineering*, IEEE Computer, No. 20, pages 10-19, 1987.

[Brown93] **M.H. Brown** and **M.A. Najork**, *Algorithm Animation Using 3D Interactive Graphics*, DIGITAL, Systems Research Center, September 1993.

[Burd96] **E.L. Burd**, **P.S. Chan**, **I.M.M. Duncan**, **M. Munro** and **P. Young**, *Improving Visual Representations of Code*, Technical Report 10/96, Centre for Software Maintenance, University of Durham, February 1996.

[Burkwald98] **S.K. Burkwald**, **S.G. Eick**, **K.D. Rivard** and **J.D. Pyrce**, *Visualizing Year 2000 Program Changes*, 6th International Workshop on Program Comprehension (IWPC'98), pages 13-18, Ischia, Italy, June 1998.

[Card87] **S.K. Card** and **A.H. Henderson Jr.**, *A multiple virtual workspace interface to support user task switching*, Proceedings of the CHI+GI 1987 (Toronto, April 5-7). ACM, New York, pages 53-59, 1987.

[Card91] **S.K. Card**, **G.G. Robertson** and **J.D. Mackinlay**, *The Information Visualizer: An Information Workspace*, Xerox Palo Alto Research Center, Palo Alto, California, 1991.

[Chalmers92] **M. Chalmers** and **P. Chitson**, *Bead: Explorations in Information Visualisation*, Proceedings of SIGIR'92, published as a special issue of SIGIR forum, ACM Press, pages 330-337, June 1992.

[Chalmers95] **M. Chalmers**, *Design perspectives in visualising complex information*, In Proceedings of IFIP 3rd Visual Databases Conference, Lausanne, Switerland, March 1995.

[Chan97] **P.S. Chan** and **M. Munro**, *PUI: A Tool to Support Program Understanding*, In Proceedings of the IEEE 5th International Workshop on Program Comprehension, Dearborn, Michigan, IEEE Computer Society Press, pages 192-198, May 28-30, 1997.

[Chan98] **P.S. Chan**, *Support for an Integrated Approach to program Understanding: An Application of Software Visualisation*, Ph.D. Thesis, Centre for Software Maintenance, Department of Computer Science, University of Durham, May 1998.

[Clarkson91] **M.A. Clarkson**, *An Easier Interface*, BYTE, February 1991.

[Colebourne94] **A. Colebourne** et al, *Populated Information Terrains: supporting the cooperative browsing of on-line information*, Research report CSCW/13/1994, Centre for Research in CSCW, Lancaster University, 1994.

[Consens92] **M. Consens**, **A. Mendelzon** and **A. Ryman**, *Visualizing and Querying Software Structures*, 14th International Conference of Software Engineering ICSE'92, Melbourne, Australia, May 11-15, 1992.

[Dieberger94] **A. Dieberger**, *Navigation in Textual Virtual Environments using a City Metaphor*, PhD Thesis, Vienna University of Technology, November 1994.

[Domingue92] **J. Domingue**, **B.A. Price** and **M. Eisenstadt**, *A Framework for Describing and Implementing Software Visualization Systems*, In Proceedings of Graphics Interface'92, Vancouver, Canada, pages 53-60, May 1992.

[Eades84] **P. Eades**, *A hueristic for graph drawing*, Congressus Numerantium, 42, pages 149-160, 1984.

[Fairchild88] **K.M. Fairchild**, **S.E. Poltrock** and **G.W. Furnas**, *Semnet: Three-dimensional Graphic Representations of Large Knowledge Bases*, in "Cognitive Science and Its Applications for Human-Computer Interaction", Fairlawn, NJ: Lawrence Erlbaum Associates, 1988.

[Fairchild93] **K.M. Fairchild**, *Information Management Using Virtual Reality-Based Visualizations*, in "Virtual Reality: Applications and Explorations", Alan Wexelblat (ed.), Academic Press Professional, Cambridge, MA, ISBN 0-12-745045-9, pages 45-74, 1993.

[Fenton91] **N.E. Fenton**, *Software metrics: A rigorous approach*, Chapman and Hall Publishers, 1991.

[Fenton97] **N.E. Fenton** and **S.L. Pfleeger**, *Software metrics: A rigorous and practical approach*, Thomson Computer Press, 1997.

[Fruchterman91] **T.M.J. Fruchterman and E.M. Reingold**, *Graph Drawing by Force-Directed Placement*, Software Practice and Experience, Vol 2, Part 11, November 1991.

[Fry95] **C. Fry** and **H. Lieberman**, *Programming as Driving : Unsafe at any Speed?*, Demonstration, ACM Conference on Computers and Human Interface [CHI-95], Denver, April 1995.

[Furnas86] **G.W. Furnas**, *Generalized fisheye views*, In Proceedings of the ACM SIGCHI '86 Conference on Human Factors in Computing Systems, pages 16-23, 1986.

[Gibson93] **W. Gibson**, *Burning Chrome*, reprinted in *Burning Chrome*, HarperCollins Science Fiction & Fantasy, ISBN 0-586-07461-9, 1993.

[Gigante93] **M.A. Gigante**, *Virtual Reality: Definition, History and Applications*, in Virtual Reality Systems, Academic Press Ltd., pages 3-14, 1993.

[Graesser81] **A.C. Graesser**, *Prose Comprehension Beyond the Word*, New York: Springer-Verlag, 1981.

[GV3D99] *The GraphVisualizer3D Project*, On-Line project overview, University of New Brunswick, Last visited July 1999. http://www.omg.unb.ca/hci/projects/gv3d/

[Henderson86] **D.A. Henderson** and **S.K. Card**, *Rooms: The use of multiple virtual workspaces to reduce spatial contention in a window-based graphical user interface*, ACM Transactions on Graphics 5, 3 July, 1986.

[Harel88] **D. Harel**, *On visual formalisms*, Communications of the ACM, 31(5), May 1988.

[Hemmje93] **M. Hemmje**, *A 3D Based User Interface for Information Retrieval Systems*, In: Proceedings of IEEE Visualization '93, Workshop on Database Issues for Data Visualization, San Jose, California, October 25-29, 1993.

[Hemmje94] **M. Hemmje**, **C. Kunkel** and **A. Willet**, *LyberWorld - A Visualization User Interface Supporting Fulltext Retrieval*, In: Proceedings of ACM SIGIR '94, Dublin, July 3-6, 1994.

[Hendley95a] **R. Hendley** and **N. Drew**, *Visualisation of complex systems*, School of Computer Science, University of Birmingham, 1995.

[Hendley95b] **R.J. Hendley**, **N.S. Drew**, **A.M. Wood** and **R. Beale**, *Narcissus: Visualising Information*, University of Birmingham, 1995.

[Hill93] **W.C. Hill** and **J.D. Hollan**, *History-Enriched Source Code*, Computer Graphics and Interactive Media Research Group, Bell Communications Research, Submitted to ACM UIST '93, 1993.

[Ingram95a] **R. Ingram** and **S. Benford**, *Legibility Enhancement for Information Visualisation*, Proceedings of Visualization '95, Atlanta, Georgia, October 30 - November 3, 1995.

[Ingram95b] **R.J. Ingram**, *Legibility Enhancement For Information Visualisation*, PhD Thesis, Department of Computer Science, University of Nottingham, September 1995.

[Jerding94] **D.F. Jerding** and **J.T. Stasko**, *Using Visualisation to Foster Object-Orientated Program Understanding*, GVU Center, College of Computing, Georgia Institute of Technology, Technical Report GIT-GVU-94-33, July 1994.

[Johnson91] **B. Johnson** and **B. Shneiderman**, *Treemaps: A space-filling approach to the visualization of hierarchical information structures*, In Proceedings of the 2nd International IEEE Visualization Conference, San Diego, pages 284-291, October 1994.

[Kennedy96] **J.B. Kennedy, K.J. Mitchell, P.J. Barclay**, *Describing and Characterising Visualisations*, in proceedings of 3rd FADIVA Workshop, Gubbio, Italy, 1996.

[Kings95] **N.J. Kings**, *Software Visualisation*, Report for the Corporate Research Programme, British Telecommunications, 1995.

[Knight98] **C.R. Knight** and **M. Munro**, *Using An Existing Game Engine to Facilite Multi-User Software Visualisation*, Workshop on System Aspects of Sharing a Virtual Environment, Collaborative Virtual Environments (CVE'98), Manchester, UK, June 1998.

[Knight99] **C.R. Knight** and **M. Munro**. *Comprehension with[in] Virtual Environment Visualisations*. To appear in Proceedings of the IEEE 7th International Workshop on Program Comprehension, Pittsburgh, PA, May 1999.

[Koike99] **Koike Labs**, *VisuaLinda: 3D Visualisation of Parallel Linda Programs*, On-Line Project Abstract, Koike Labs, University of Electro-Communications in Tokyo, Last visited July 1999. http://www.vogue.is.uec.ac.jp/

[Kraemer93] **E. Kraemer** and **J.T. Stasko**, *The Visualisation of Parallel Systems: An Overview*, Journal of Parallel and Distributed Computing (18), pages 105-117, 1993.

[Krohn96a] **U. Krohn**, *VINETA: Navigation Through Virtual Information Spaces*, in "AVI: Advanced Visual Interfaces", Ed. T. Cartaci, Gubbio, Italy, ACM, 1996.

[Krohn96b] **U. Krohn**, *Visualization for Retrieval of Scientific and Technical Information*, PhD. Dissertation, 1996.

[Levialdi95] **S. Levialdi, A. Massari** and **L. Saladini**, *VIRGILIO – Virtual Metaphors for Database Exploration*, FADIVA 2 Workshop, Glasgow, UK, July 1995.

[Letovsky86a] **S. Letovsky**, *Cognitive Processes in Program Comprehension*, Empirical Studies of Programmers, Albex, Norwood NJ, pages 58-79, 1986.

[Letovsky86b] **S. Letovsky** and **E. Soloway**, *Delocalized Plans and Program Comprehension*, IEEE Software, Vol. 19, No. 3, pages 41-48, May 1986.

[Lieberman95] **H. Lieberman** and **C. Fry**, *Bridging the Gulf Between Code and Behaviour in Programming*, ACM Conference on Computers and Human Interface [CHI-95], Denver, April 1995.

[Littman86] **D.C. Littman**, **J. Pinto**, **S. Letovsky** and **E. Soloway**, *Mental Models and Software Maintenance*, Empirical Studies of Programmers, Albex, Norwood NJ, pages 80-98, 1986.

[Lynch60] **K. Lynch**, *The Image of the City*, M.I.T. Press 1960.

[Mackinlay91] **J.D. Mackinlay**, **S. Card** and **G.G. Robertson**, *Perspective Wall: Detail and Context Smoothly Integrated*, In Proceedings of the ACM SIGCHI '91 Conference on Human Factors in Computing Systems, pages 173-179, New Orleans, LA, USA, April 1991.

[Mayrhauser94] **A. von Mayrhauser** and **A.M. Vans**, *Dynamic Code Cognitive Behaviours For Large Scale Code*, Proceedings of the IEEE 3rd International Workshop on Program Comprehension, Berlin, Germany, March 29-31, IEEE Computer Society Press, pages 9-18, 1994.

[Mayrhauser95] **A. von Mayrhauser** and **A.M. Vans**, *Program Comprehension During Software Maintenance and Evolution*, IEEE Computer, Vol. 28, No. 8, pages 44-55, August 1995.

[Meekel88] **J. Meekel** and **M. Viala**, *Logiscope : A Tool For Maintenance*, Proceedings of ICSM'88, pages 328-334, 1988.

[Miller93] **B.P. Miller**, *What to Draw? When to Draw? An Essay on Parallel Program Visualisation*, Journal of Parallel and Distributed Computing (18), pages 258-264, 1993.

[Muthukumarasamy95] **J. Muthukumarasamy** and **J.T. Stasko**, *Visualising Program Executions on Large Data Sets Using Semantic Zooming*, GVU Center, College of Computing, Georgia Institute of Technology, Technical Report GIT-GVU-95-02, 1995.

[Myers90] **B.A. Myers**, *Taxonomies of Visual Programming and Program Visualisation*, Journal of Visual Languages and Computing, (1), pages 97-123, 1990.

[Najork95] **M. Najork**, *Visual Programming in 3D*, Dr. Dobb's Journal, No. 237, pages 18-31, December 1995.

[Olsen93] **K.A. Olsen**, **R.R. Korfhage**, **K.M. Sochats**, **M.B. Spring** and **J.G. Williams**, *Visualisation of a Document Collection: The VIBE System*, Information Processing and Management, Vol. 29, No. 1, pages 69-81, Pergamon Press Ltd, 1993.

[Oman90] **P. Oman**, *Maintenance Tools*, IEEE Software, Vol. 23, No. 3, pages 59-65, May 1990.

[Price93] **B.A. Price**, **R.M. Baeker** and **I.S. Small**, *A Principled Taxonomy of Software Visualisation*, Journal of Visual Languages and Computing, No. 4, pages 211-266, 1993.

[PVMTrace99] *The PVMTrace project*, On-Line thesis proposal, University of New Brunswick, Last visited July 1999. http://www.omg.unb.ca/hci/projects/hci-pvmtrace.html

[Quinn79] **N. Quinn** and **M. Breur**, *A force directed component placement procedure for printed circuit boards*, IEEE Transactions on Circuits and Systems, CAS-26, (6), pages 377-388, 1979.

[Reiss94] **S.P. Reiss**, *3-D Visualization of Program Information*, In proceedings of GD'94: DIMACS international workshop on graph drawing, pages 12-24, Princeton, New Jersey, USA, October 1994.

[Rekimoto93] **J. Rekimoto** and **M. Green**, *The Information Cube: Using Transparencty in 3D Information Visualization*, Proceedings of the Third Annual Workshop on Information Technologies & Systems (WITS'93), pages 125-132, 1993.

[Robertson91] **G.G. Robertson, J.D. Mackinlay** and **S. Card**, *Cone Trees: Animated 3D Visualizations of Hierarchical Information*, Proceedings of the ACM SIGCHI '91 Conference on Human Factors in Computing Systems, pages 189-194, New Orleans, LA, USA, April 1991.

[Robson88] **D.J. Robson, K.H. Bennett, B.J. Cornelius** and **M. Munro**, *Approaches to Program Comprehension*, Computer Science Technical Report 11/88, University of Durham, 1988.

[Roman92] **G. Roman** and **K.C. Cox**, *Program Visualization: The Art of Mapping Programs to Pictures*, Department of Computer Science, Washington University, 1992.

[Roman93] **G. Roman** and **K.C. Cox**, *A Taxonomy of Program Visualisation Systems*, IEEE Computer, December 1993.

[Sakar92] **M. Sakar** and **M.H. Brown**, *Graphical fisheye views of graphs*, In proceedings of the ACM CHI '92, pages 83-91, May 3-7, 1992.

[Shneiderman79] **B. Shneiderman** and **R. Mayer**, *Syntactic / Semantic Interactions in Programmer Behaviour: A Model and Experimental Results*, International Journal of Computer and Information Sciences, Vol. 8, No. 3, pages 219-238, 1979.

[Shneiderman80] **B. Shneiderman**, *Software Psychology*, Cambridge MA: Winthrop Publishers Inc., 1980.

[Shneiderman91] **B. Shneiderman**, *Tree Visualization with Tree-maps: A 2-d space-filling approach*, Department of Computer Science and HCI Laboratory, University of Maryland, June 1991.

[Shu88] **N.C. Shu**, *Visual Programming*, Van Nostrand Reinhold, New York, 1988.

[Soloway84] **E. Soloway** and **K. Ehrlich**, *Empirical Studies of Programming Knowledge*, IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, pages 595-609, September 1984.

[Soloway88] **E. Soloway**, **B. Adelson** and **K Ehrlich**, *Knowledge and Processes in the Comprehension of Computer Programs*, in "The Nature of Expertise", M. Chi, R. Glaser, and M. Farr (eds.),A. Lawrence Erlbaum Associates, Hillside, NJ, pages 129-152, 1988.

[Sommerville92] **I. Sommerville**, *Software Engineering*, Fourth Edition, Addison-Wesley publishers, 1992.

[Standish84] **T.A. Standish**, *An essay on Software Reuse*, IEEE transactions on Software Engineering, Vol. SE-10, No. 5, pages 494-497, September 1984.

[Stasko92a] **J.T. Stasko**, *Three-Dimensional Computation Visualisation*, GVU Center, College of Computing, Georgia Institute of Technology, Technical Report GIT-GVU-92-20, 1992.

[Stasko92b] **J.T. Stasko** and **C. Patterson**, *Understanding and characterizing software visualisation systems*, Proceedings of the IEEE 1992 workshop on Visual Languages, Seattle, Washington, pages 3-10, 1992.

[Storey95] **M-A.D. Storey** and **H.A. Müller**, *Manipulating and Documenting Software Structures Using SHriMP Views*, In Proceedings of the ICSM '95 conference on Software Maintenance, Opio (Nice), France, October 17-20, pages 275-284, 1995.

[Storey97] **M-A.D. Storey**, **F.D. Rracchia** and **H.A. Müller**, *Cognitive Design Elements to Support the Construction of a Mental Model during Software Visualisation*, Proceedings of the IEEE 5th International Workshop on Program Comprehension, Dearborn, Michigan, May 28-30, IEEE Computer Society Press, pages 17-28, 1997.

[Sutherland65] **I. Sutherland**, *The Ultimate Display*, Proceedings of the International Federation of Information Processing Congress, pages 506-508, 1965.

[Vince95] **J. Vince**, *Virtual Reality Systems*, Addison-Wesley publishers, 1995.

[Walker93] **G.R. Walker**, **P.A. Rea**, **S. Whalley**, **M. Hinds** and **N.J. Kings**, *Visualisation of telecommunications network data*, BT Technology Journal, Vol. 11, No. 4, pages 54-63, October 1993.

[Walker95] **G. Walker**, *Challenges in Information Visualisation*, British Telecommunications Engineering Journal, Vol. 14, pages 17-25, April 1995.

[Ware93] **C. Ware**, **D. Hui** and **G. Franck**, *Visualizing Object Oriented Software in Three Dimensions*, In proceedings of CASCON '93 (IBM Centre for Advanced Studies), pages 612-620, Toronto, Ontario, Canada, October 1993.

[Ware94] **C. Ware** and **G. Franck**, *Viewing a Graph in a Virtual Reality Display is Three Times as Good as a 2D Diagram*, In 1994 IEEE Conference on Visual Languages, pages 182-183, St. Louis, Missouri, USA, October 1994.

[Watson95] **A.B. Watson** and **J.T. Buchanan**, *Towards Supporting Software Maintenance with Visualisation Techniques*, Technical Report R5, University of Strathclyde, 1995.

[Wiedenbeck86] **S. Wiedenbeck**, *Processes in Computer Program Comprehension*, Empirical Studies of Programmers, Albex, Norwood NJ, pages 48-57, 1986.

[Wiedenbeck91] **S. Wiedenbeck**, *The Initial Stage of Program Comprehension*, International Journal of Man-Machine Studies, Vol. 35, No. 4, pages 517-540, 1991.

[Wiss98a] **U. Wiss** and **D.A. Carr**, *An Empirical Study of Task Support in 3D Information Visualizations*, Technical report 1998:31, Luleå University of Technology, 1998.

[Wiss98b] **U. Wiss**, **D.A. Carr** and **H. Jonsson**, *Evaluating Three-Dimensional Information Visualization Designs: A Case Study of Threee Designs*, in proceedings of IEEE Conference on Information Visualization, IV'98, London, England, July 1998.

[Wiss98c] **U. Wiss** and **D.A. Carr**, *A Cognitive Classification Framework for 3-Dimensional Information Visualization*, Technical report 1998:04, Luleå University of Technology, 1998.

[Wood95] **A.M. Wood**, **N.S. Drew**, **R. Beale** and **R.J. Hendley**, *Hyperspace: Web Browsing with Visualisation*, In Third International World-Wide Web Conference Poster Proceedings, pages 21-25, Darmstadt, Germany, April, 1995.

[Young97] **P. Young** and **M. Munro**, *A New View of Call-Graphs for Visualising Code Structure*, Technical Report 03/97, Centre for Software Maintenance, University of Durham, January 1997.

[Young98a] **P. Young** and **M. Munro**, *Visualising Software in Virtual Reality*, 6th International Workshop on Program Comprehension (IWPC'98), pages 19-26, Ischia, Italy, June 1998.

[Young98b] **P. Young** and **M. Munro**, *3D Software Visualisation*, 1st International Conference on Visual Representations and Interpretations (VRI'98), Liverpool, UK, September 1998.