Handbook of Software Engineering and Knowledge Engineering
Vol. 0, No. 0 (2000) 000-000


# SYSTEM AND SOFTWARE VISUALISATION

CLAIRE KNIGHT

*Visualisation Research Group, Department of Computer Science,*
*University of Durham, Durham, DH1 3LE, UK.*
*E-mail: C.R.Knight@durham.ac.uk*

The progress of software visualisation, beyond that of nodes and arcs in various forms, has lagged behind the developments made in such fields as information visualisation. That is now starting to change and recent innovations are reviewed, based on a historical view of software visualisation, and followed by some of the many outstanding research issues. The use of three-dimensions is almost as new as the more recent research directions and the focus is on the newer visualisations that make use of the extra dimension for the display of information. Whilst this leaves many unanswered questions and unsolved areas, it is a field where much fruitful work has been done and is ripe for further research.

*Keywords:* Software Visualisation, System Visualisation, Intelligence Amplification, Program Comprehension, Software Maintenance

## 1. Introduction

If the entire of what many consider to be the software visualisation field is reviewed, then this article would be much larger in size and also consist mainly of variations on the nodes and arcs theme. Because the use of the third dimension for system and software visualisation is emerging as a viable alternative for the representation of complex artefacts then it was considered much better to focus on this form of software visualisation. The previous techniques had various identified shortcomings and the space and freedom afforded by the extra dimension has the potential to be usefully employed to overcome some of these problems.

Software visualisation can be seen as a specialised subset of information visualisation. This is because information visualisation is the process of creating a graphical representation of abstract, generally non-numerical, data. This is exactly what is required when trying to visualise software. The term software visualisation has many meanings depending on the author. For the purposes of this thesis software visualisation can be taken to mean any form of program visualisation that is used after the software has been written as an aid to understanding (i.e. it does not mean visual programming). More formally software visualisation can be defined as [1]:

> ``*Software visualisation is a discipline that makes use of various forms of imagery to provide insight and understanding and to reduce complexity of the existing software system under consideration.*"

The goal of software visualisation is included in the above definition. To create a visualisation for no real purpose would be a pointless exercise. It has long been known that understanding software is a complex and hard task because of the complexity of the software itself. Therefore techniques that aid the programmer in his comprehension of an existing software system deserve research focus. Software visualisation aims to aid the programmer by providing insight and understanding through the graphical displays and views, and to reduce the perceived complexity through the use of suitable abstractions and metaphors.

In presenting spatially arranged and located information, i.e. visually, there is the benefit that the human perception skills can be used for part of the comprehension. This moves some of the comprehension load away from the conscious cognitive processing. The double navigation and orientation load of virtual reality visualisations could be seen as a burden but it can be useful. The two mapping processes, that of virtual space movement and of the data comprehension, can feed each other and help to build up a solid mental representation of both. Using three dimensions for visualisation adds an element of familiarity and realism into systems. The world is a three-dimensional experience and by making the visualisation more like that world means there is less cognitive strain on the user. This in turn makes the system easier and more comfortable to use because of all the experience and knowledge the user has built up elsewhere. In using three dimensions the depth cues that make the world, and the visualisation, appear three-dimensional can be used as part of the visualisation. This means that the aim of the visualisation to aid the comprehension of complex phenomena can be achieved without adding unnecessary complications because of the visualisation used.

One of the main problems for software visualisation (and other forms of information visualisation) is of trying to create a tangible representation of something that has no inherent form. Therefore the aim is to visualise the intangible in an effective and useful way. Effective and useful here refer to the visualisation being able to increase the understanding of the user whilst reducing the perceived complexity. Whilst this article does not focus particularly on providing complete overviews of the preceding tools and techniques used for program comprehension and more recently that of information visualisation it does present some of these are the background for later work. It is also useful to provide a yardstick to see where improvements have been made; or even where newer representations have not lived up to expectation. As with much emerging research, when it relies on certain popular technologies it tends to become part of the wider hype. This has, unfortunately, been the case with much of the visualisations in three-dimensions because of their reliance on technologies such as Virtual Reality (VR) for implementation.

## 2. Historical Perspective

This section traces the origins of software visualisation through program comprehension, features identified as important for program comprehension tools and techniques (such as visualisation), taxonomies, and then finally the nodes and arcs usually associated with the term software visualisation.

### 2.1 *Program Comprehension*

Program comprehension is an important part of not only software maintenance, but also the entire software engineering process. It is required for many tasks that are carried out under these broad headings but for all the aim is the same. Program comprehension is carried out with the aim of understanding an existing piece of code. It is a gradual process of building up the necessary understanding by examining sections of the source code. Using the knowledge gained from the source code explanations and understanding can be built and refined. According to Biggerstaff et al. [2] this process of discovery and refinement is known as the *Concept Assignment Problem,* whilst several other program comprehension strategies have different terms or processes to describe the same activities (such as bottom up comprehension). An overview of many types of program comprehension can be found in Robson et al. [3] and Von Mayrhauser and Vans [4]. Many authors have documented the ways in which studies have shown that programmers understand code such as Corbi [5], Oman and Cook [6] and Chan and Munro [7]. These documents provide brief overviews of the strategies.

Program comprehension is very much a gradual process where the maintainer gathers information through studying various aspects of the code at different times, and possibly by returning to previously examined pieces of code. Wiedenbeck [8] supports this view of program comprehension. This process is true regardless of the strategy employed to examine the various pieces of code that constitute the system. Despite the importance of program comprehension to such a diverse and wide range of other software maintenance activities there still remains much work to be done to improve the tools and refine the techniques that exist today. Software continues to increase in size and complexity and whilst the program comprehension theories may support this growth, the tools and techniques developed for helping maintainers do not keep up with the speed, and size of change.

### 2.2 *Features of Program Comprehension Tools*

Storey et al. [9] identified a hierarchy of cognitive issues that are important when considering what facilities a program comprehension tool should include. They identify the fact that software exploration tools can be likened to hypermedia document browsers. Because of this a hierarchy of hypermedia cognitive issues has been adapted to form program comprehension guidelines. Also identified is the lack of support in existing systems for the integrated and top-down models of comprehension and the inability to switch between different mental model information. Navigation and orientation cues were also identified as an area for future research.

The work done by Chan and Munro [7] identifies the need to provide different viewpoints for maintainers. This allows them to choose the most appropriate view for the current task, and also to be able to switch between views to gain a higher or lower level understanding of some piece of information.

As long ago as 1987 authors had realised the benefit of cross-referencing information but few tools have actually implemented such functionality. Munro and Robson [10] implemented an interactive cross-reference tool that solved a problem at the time of cross referencers only producing large listings of textual information. Foster and Munro [11] produced a cross-reference tool that allowed maintainers to cross-reference the source code, and included documentation. Fletton and Munro's [12] work is also related to this cross-referencing idea. An alternative view is that of Landis et al. [13], who identified the use of cross-referencing but suggested it is only useful for variable information and determining where to make changes of that nature. In terms of dataflow it was considered to be of less use, and a criticism of the method was that variables with the same names in different scopes could be listed and confuse the programmer. Von Mayrhauser et al. [14] suggest that cross referencing of related areas of code would make identification of areas where changes need to be made easier. These cross-reference links should be, where possible, hypertext and also link to algorithm and/or domain information. They also identify the need to provide orientation cues in the documentation and propose the use of some form of browser history with on-line sticky notes to make this effective. They also think that documentation of the system (which could be included in any tool that was used to aid comprehension) should have a high-level *road map of the system structure*.

### 2.3 *Visualisation Taxonomies*

Early work often contained animation, or was thought to have to contain animation. This is reflected in the detail of the various classification and evaluation taxonomies developed in the early 1990s. Whilst such detail is beyond the scope of this article it is worth reviewing the major points of the three main taxonomies. Myers [15] identifies one of the first program visualisation taxonomies. In this taxonomy he makes the distinction that all the included systems use graphics to illustrate some part of the program after it has been written. This distinction is important because it makes clear that the taxonomy covers only program visualisation and not visual programming. Other authors are not so clear and even confuse the two terms.

Other authors have produced taxonomies of program visualisation and classified things in a different way to Myers [15]. Price et al. [16] produced a taxonomy of software visualisation systems that is based on six categories. In creating such a taxonomy the authors aimed to create a ``road map" of the research to the point when the taxonomy was created. The taxonomy created by Price is more detailed than that of Myers' and can be arranged into a hierarchical structure. This structuring was a deliberate move by the authors to allow for the taxonomy to be extended and revised as software visualisation (the term they use instead of program visualisation) systems evolved and matured. Another taxonomy is the one defined by Roman and Cox [17] which is derived from their earlier work [18]. This is closer to the taxonomy of Price

et al. [16] than the one of Myers [15] and some parallels can be drawn between the two.

### 2.4 *Early Representations; Nodes and Arcs*

For many years basic visualisation, based around simple boxes and lines, has been done in an attempt to be able to ease some of the cognitive overload caused by program comprehension. The problems with such visualisations is that they can very easily become incomprehensible by trying to force large amounts of information into a small space, relying solely on two-dimensions for the representations. Baker and Eick [19] acknowledge the problems of such approaches:

> ``*When applied to production-sized systems, routines for producing flow charts, function call graphs and structure diagrams often break because the diagram is too complicated. Or they produce displays that contain too much information and are completely illegible.*''

Much effort has been spent on visualising programs in two-dimensions, with graph structures such as call-graphs being prominent. It is acknowledged that these forms of visualisation suffer when the number of, and relationships between, information is complex. The representations themselves can even become more complicated than the code itself. An example of this can be seen in Figure 1. If this is the only thing that can be used to aid the comprehension of a well maintained medium sized commercial system, then it is obvious that something more is needed.

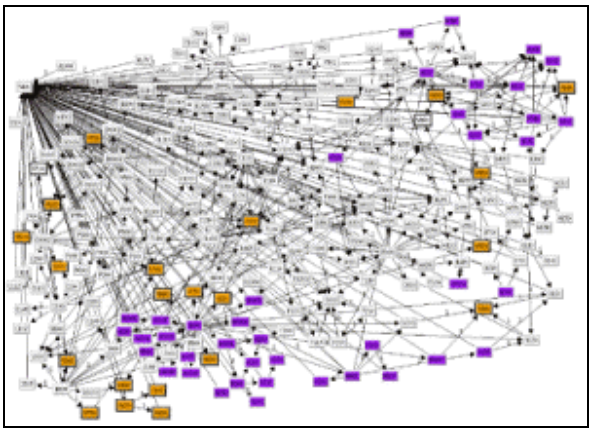

**Fig. 1.** Call graph of a medium sized system

In attempts to address the obvious problems with the existing representations several systems and layout algorithms have been developed. The following sections provide more detailed information on a representative sample of these. Another technique that has been applied, with little success due to the same problems still being prevalent, is the representation of node and arc structures such as call graphs in three-

dimensions [43, 44, 30]. Much of this work was done without thought or design being applied to the virtual space in which the graphics were located which destroyed the possible usefulness of such an environment.

## 3. Examples

The examples in the following four sections are representative of the advances made in software visualisation in the last ten years. The first three are refinements and enhancements of node and arc technology with improved browsing and node management, and the final one is one of the early software visualisations to move away from total reliance on nodes and arcs. Whilst these four tools show slightly different aspects of the program code, they are all intended for the tasks of program comprehension and software maintenance hence their inclusion.

### 3.1 *SHriMP*

SHriMP (**S**imple **H**iera**r**ch**i**cal **M**ulti-**P**erspective) [20] is a visualisation that combines the nodes and arcs representation so cherished by software visualisers with the fisheye filtering technique. As the name implies, the fisheye technique emulates the behaviour of a fisheye lens. The information at the centre of the view is magnified, whilst that at the periphery is reduced in size. SHriMP also incorporates the use of nested graphs for the display of software structures. The use of these two techniques provides, according to the authors, the ability to create multiple views at different levels of abstraction and perspective. To be able to interact with the data in this way is a powerful way of dealing with large amounts of information as selective filtering and viewing can take place. Such extensions are a step towards being better able to deal with the large and complex software that is so pervasive today, but the user of two-dimensions and the standard representation as the basis may limit the applicability and use of such tools.

### 3.2 *VIFOR*

VIFOR stands for **V**isual **I**nteractive **FOR**tran and is a software tool that is geared towards maintaining Fortran 77 code [21]. This system works by using a database of code detail from the Fortran code and then allowing it to be viewed and queried in either the textual form of the code or a graph layout based visualisation. This layout mechanism was developed for the VIFOR tool and attempts to merge the standard call graph and data dependency graphs that are more commonly used.

Early work on a C maintenance and understanding tool is also documented in this paper; VIC. An extension of this work was the development of VIFOR 2 [22]. This improved the browsing system to allow the integration of incremental recording and retrieval of documentation.

### 3.3 *CARE*

CARE (**C**omputer **A**ided **R**e-**E**ngineering) is an understanding tool that works with C source code [23]. This understanding tool makes use of two-dimensional visualisations in windows and browsers (as with the previous tools) to show graphs of some of the code relations. As with VIFOR, CARE displays the data flow and call

structure of the program using an extension of the VIFOR layout algorithm. In order to do this, a repository of the structural and functional dependencies in the code is generated, and the presentation part of the tool uses this information when displaying the visualisations. The tool also supports the creation and use of both graphical and textual slices through the information.

As with VIFOR, an extension of the tool was developed to deal with other languages. OO!CARE [24] is an extension of CARE that deals with C++ code, hence the **O**bject **O**riented addition to the name. It is also able to deal with C code because of the syntactical similarities of the language notwithstanding the object oriented part of C++.

### 3.4 *SeeSys and SeeSoft*

In an attempt to address some of the shortcomings of relying solely on nodes and arcs for data representation, the tools SeeSys [19] and SeeSoft [25] were developed. These tools are part of a research effort that produced similar displays for several underlying data types. The visualisation technique used by these systems is based on the idea of decomposition of the information to be visualised into its component form. Colour and interaction are incorporated into the systems, and the displays make much use of colour scales to visualise extra information about the underlying data. The system also uses the overlay of additional information onto the display to provide yet more facts for the user.

SeeSys is a visualisation system for software metrics whilst SeeSoft visualises the program code and the constituent files. These visualisations are based on three principles:

1. The individual components can be assembled to form the whole. This allows the user to easily see the relationships between them.
2. Pairs of components can be compared to understand how they differ.
3. The components can be disassembled into smaller components. This important feature of the components allows the structure of the display to reflect the structure of the software.

The individual components are visible whilst maintaining a view of the whole system. This sort of technique has also been applied in the Information Mural visualisations of Jerding and Stasko [26], whilst the concept of encoding wear as a coloured property of source code was originally documented by Hill and Hollan [27].

### 3.5 *Summary*

In his paper *No Silver Bullet* [28], Brooks wrote

> ``*Software is invisible and unvisualizable.*
>
> …
>
> *…software is very difficult to visualize. Whether one diagrams control flow, variable-scope nesting, variable cross-references, dataflow, hierarchical data structures, or whatever, one feels only one dimension of the intricately interlocked software elephant. If one superimposes all the*

> *diagrams generated by the many relevant views, it is difficult to extract any global overview."*

At the time Brooks wrote this, visualising software meant displaying some information about (or some aspect of) the software in a graph structure. From what can be seen in Figure 1, he has a point. This need not now be the case with the advances in computer hardware and graphics technology.

Two-dimensional techniques have shortcomings, but this is not to trivialise the issues that still remain with the nodes and arcs techniques. Layout, for example, is a hard problem and one that is not appreciated by many [50]. The systems presented in the previous sections provide a representative sample of the sorts of program comprehension tools that have been developed to aid understanding and are early attempts at visualisation. The last of these sections shows that the reliance on nodes and arcs and solely investigating layout algorithms and clustering is not necessarily the only way forward, even with two-dimensions. This work is a stepping stone to moving onto three-dimensional, coloured, and non-arc reliant representations.

## 4. Recent Research

Despite the identified problems with the traditional forms of software visualisation, it is not simply enough to throw the techniques of information visualisation at the source code and expect useful and usable visualisations to emerge. Information visualisation influences have produced many hierarchical and tree oriented visualisations, and some information landscape style visualisations [45, 46, 47, 48, 49] and these could have application for software visualisation. Indeed Hendley and Drew [43, 44] produced springs layout algorithms for nodes and arcs in three-dimensions, which bears a similarity to such research.

Notwithstanding these advances, there are general unsolved problems with three dimensional visualisations, such as scalability and evolution, which are important to all visualisations but paramount to software visualisation due to the complexity and changeability of the underlying data; the source code. There may also be issues that are pertinent to software visualisations that do not have much effect on information visualisations; database visualisation is a subset of information visualisation because of the nature of the data source being visualised. For a more complete review of information visualisation in three-dimensions, the interested reader is referred to the comprehensive survey by Young [29].

Recently more attempts at utilising three dimensions for software visualisation have been carried out by Feijs and De Jong [30] where they visualised node and arc structures using variously coloured lego blocks for the nodes. These visualisations showed architectural structures and relationships present in the code and used various colours to indicate different pieces of information about that structure to the viewer. Whilst this paper acknowledged that they have more research to do the work does not seem to address many of the issues and limitations of the use of three dimensions. The creation of their visualisations relies again on nodes and arcs but using an extra dimension. This does provide a greater degree of flexibility than the two-dimensional form of such structures but again does not scale or evolve well; two very important issues.

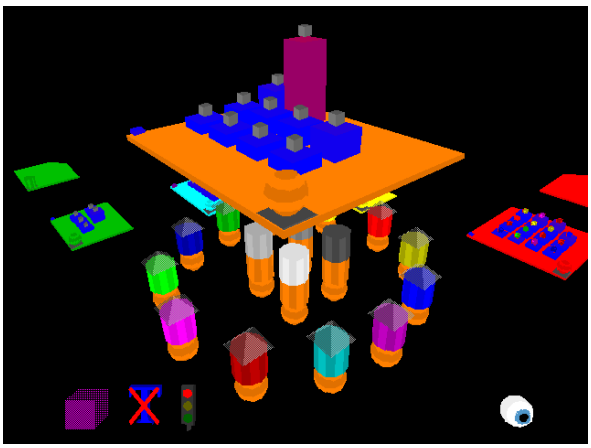

**Fig. 2.** CallStax visualisation underneath part of a FileVis display

In an attempt to move away from the obvious connectivity displayed in software visualisations Young and Munro [31] and Young [32] produced visualisations based around abstract three-dimensional geometrical shapes. The first, known as *CallStax*, showed the visualisation of the calling structure of C code (essentially the same information as a call graph) with coloured stacks of blocks showing the routes through the graph. The second, *FileVis*, showed a view of the software system with the code files represented individually as floating platforms around a central point which represented the connectivity of the source code files. A view of these visualisations can be seen in Figure 2. This visualisation combines both in order to show two aspects of the C code at the same time.

**Fig. 3.** View over a software district showing many, possibly complex, methods

Further advancing the three-dimensional space aspect of the visualisations, work by Knight and Munro [1, 33, 34] moved to considering the use of virtual reality

environments for software visualisation. *Software World* was created to show that three-dimensions (in this case also showing the viability of real world metaphors) could be used to create automatable and scalable software visualisations. Buildings, cities, and also at the highest level atlas views, were used to represent Java source code. An example view of this visualisation can be seen in Figure 3.

In order to show some of the method level views that can be created with this visualisation over 17,000 lines of Java code (which together composed a package) written by others was parsed, and district visualisations automatically generated. The code was split across 70 classes, which provided different district views. Two of the images that came out of this can be seen in Figure 3 and Figure 4.

The use of environments allowed extensions of this work to consider the benefits and challenges of creating virtually inhabitable spaces where the visualisation provides a view of the data under consideration and facilitates communication and collaboration within that data by those working on the same tasks or having similar knowledge. Some of these further issues can be found in [35].

### *4.1 Example of Use: Impression*

The condition of any piece of source code can affect the effort required for future comprehension and maintenance activities. There are many measures that can be utilised to provide a prediction of the condition of many aspects of the source code in numerical form and these may be used to justify a preventative maintenance process taking place. It is also useful in these situations, especially if the numbers that come out of the measured analysis of the code are averaged, to be able to ``see" possible problem areas of the code and to know where to direct efforts.

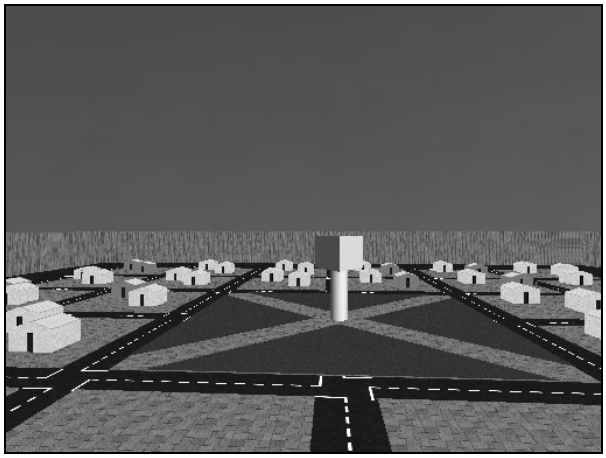

**Fig. 4.** A district representing a class containing only very small methods

Such a process can be achieved through the use of the visualisation tools for browsing and a more exploratory form of comprehension. These overviews immediately make obvious anything that is extreme and can draw the user to that area. It may be that these apparently anomalous areas of the code are fine considering the

use to which that piece of code is put; i.e. if the task is complex the code to achieve it may well be unavoidably complex.

In order to find this information using *Software World* the following stages are necessary:

- Use the world and county views to gain an overview of the split and top-down structure of the system and packages.
- Query to locate class (for each class) which is highlighted from the country view down. If the package is not specified then several could be highlighted. This is done from the user's office in the visualisation environment.
- The class of interest is selected and the district that represents this class is then made the focus of the virtual environment. This district is shown in the context of the file in which the class is located so that the file information and any related classes can be seen.
- The district can then be explored, the heights of buildings surveyed and the colours of buildings used to provide an indication of the accessibility of the code. Differing visual examples of this can be seen in Figures 3 and 4.

Once methods of interest have been identified then they can be explored in more detail by entering the appropriate building. Source code (actual text) and also other extracted information can be viewed and analysed from this point in the visualisation. This will then lead any further exploration considered necessary for this method and/or class.

These recent research initiatives have shown that such software visualisations are viable, and have the potential to become powerful additions for aiding understanding and providing a forum in which knowledge about the data is also managed, viewed, and interpreted.

## 5. Moving Forward

Unfortunately it is not simple a case of following a recipe book set of simple steps when creating visualisations because of the very purpose of visualisation data sources. The underlying data can be sizeable, and it can be complex. These facts alone preclude the use in many situations of what could be traditionally termed ``off-the-shelf" visualisations, but there are a wider range of issues as to why this is not necessarily the best way of visualising software and systems.

One of the main problems for software visualisation (and other forms of information visualisation) is of trying to create a tangible representation of something that has no inherent form. Therefore the aim is to visualise the intangible in an effective and useful way. Effective and useful here refers to the visualisation being able to increase the understanding of the user whilst reducing the perceived complexity. Many authors have identified the intangible nature of software including Walker [36], Ball and Eick [37] and Chapin and Lau [38].

### 5.1 *Intelligence Amplification*

Intelligence amplification (IA) is the use of computers to aid and enhance human intelligence rather than the artificial intelligence (AI) aim of trying to substitute humans with computers. Intelligence amplification builds on the skills that humans

already have, and tries to augment the areas that are lacking in some way. Frederick Brooks (documented in by Rheingold [39]) describes his beliefs about intelligence amplification in the following way:

> *``I believe the use of computer systems for intelligence amplification is much more powerful today, and will be at any given point in the future, than the use of computers for artificial intelligence (AI). In the AI community, the objective is to replace the human mind by the machine and its program and its data base. In the IA community, the objective is to build systems that amplify the human mind by providing it with computer-based auxiliaries that do the things that the mind has trouble doing."*

Brooks identifies three areas in which humans are more skilled than computers. The first is *pattern recognition* (aural or visual). The second is in performing *evaluations*, and the third is the *overall sense of context* that allows previously unrelated pieces of information to become related and useful in a new situation.

Walker [36] also touches on the subject of intelligence amplification in his discussion on the challenges of visualisation.

> *``A natural and intuitive visual interface can retain the critical contribution from human perceptual skills, ensuring that opportunities for lateral thinking or perhaps an unexpected leap of imagination are not lost. Programming a computer to ``look for something interesting" in a database is a major undertaking, but given appropriate tools, it is a task for which humans are well equipped."*

The first sentence can be seen to the similar to the third skill identified by Brooks, that of a sense of context. The second sentence by Walker is essentially talking about the pattern recognition skill specified by Brooks (in [39]).

Intelligence amplification is of importance to software visualisation (and any other form of visualisation) because in representing large and complex data sets graphically the aim is to help the user to get a better understanding of content of the data sets. By aiding the user in this way visualisation tools are acting also as intelligence amplification tools. Reading through many thousands of pieces of information and then summarising them in a finite graphical space would be an immense, complex and possibly tedious task. For a computer with the right ``instructions", it is a simple data processing exercise.

Hubbold et al. [40] make a similar connection with the field of VR (and therefore visualisations that make use of VR as an enabling technology). They also identify the pattern recognition and contextual abilities of humans.

> *``In our everyday existence we cope with, and filter out, tremendous amounts of information almost effortlessly and with very little conscious thought. Indeed, if the same information, in all its detail, were to be presented in a form that we had to think about consciously, then we would be overwhelmed quite easily. Spatial awareness, pattern recognition, information filtering, coordination of multiple information streams are things we take for granted. Rather than look for a solution in AI, part of the VR thesis is that information presented in a suitable way can be processed far more effectively and directly by people."*

The role of a visualisation system as an intelligence amplification tool rather than as a system that tries to second guess the information the user requires is emphasised by Crossley et al. [41]:

> ``*…the role of the system is not to select documents similar to a user-supplied query but to organise and display information about many documents in such a way as to assist users to select useful documents on their own.*"

This shows that the important challenges and research issues for visualisations are to be able to handle such tasks well and provide the necessary support as transparently as possible. Changing the query mechanism in order to improve performance (for example in the situation above) is not going to help in another situation or be widely applicable to other visualisations.

### 5.2 *Important Factors*

There are many factors that are important for visualisation. The main ones in the context of software visualisation [42], and indeed for all information visualisation, are:

- Scalability
- Evolution
- Automation
- Navigation, Interaction, and Orientation

Navigation, interaction, and orientation are presented together because they are often highly interconnected. Orientation features are also very often useful for navigation and vice-versa. Interaction mechanisms are also affected by the navigation available to users. The final three bullet points cover issues that should be considered vital for all visualisations, including those that create virtual environments. Obviously there may have to be trade-offs for a particular system with respect to some of these features but striving to achieve all of these goes some way towards reliable creating useful and usable visualisations.

The first of these points is to do with the scalability of the visualisation. It is all very well for a visualisation system to work with a small source code sample but unless it works with large systems, with little human intervention then it is of no use to programmers and maintainers. The visualisation needs to scale well, and because of this the metaphors and abstractions used should be carefully considered. There is also a need for automation. If each facet of the visualisation needs to be created by hand there is little point in using the visualisation with real systems. Some human intervention can be considered to be acceptable, but on the whole the visualisation needs to be created with as little as possible. Navigation and orientation are the final points to consider. It does not matter how well drawn the graphics of a visualisation are, or how suitable the mapping and metaphor chosen, if the resulting visualisation cannot be navigated adequately. It is important to be able to explore the information presented and to be able to locate the desired pieces of information. It is also vital, especially when considering three-dimensional displays, that the visualisation itself is navigable and able to be comprehended in some way by users. Without either of these

two then the visualisation is immediately unusable and the effort on representations etc. wasted.

### 5.3 *Summary*

Much research still needs to be done in many areas both in and around software visualisation and this section has identified some of the more pressing concerns, and presented a rationale for the use of visualisation techniques. It should not be considered a failing of the field to have not answered all such questions at this moment in time; it provides a rich forum in which further research can be done to make the necessary future advances. It also provides a basis on which the technique of visualisation for understanding can start to be applied to larger projects and data sets and usefully used by those outside of research.

## 6. Conclusion

This article has introduced and discussed many facets of the three-dimensional research agenda and provided rationale for the creation of software visualisations that exploit the extra dimension. Software visualisation is still a hard problem and purely by using the extra dimension the field cannot hope to solve the identified deficiencies of the two dimensional representations. What this information has tried to show is that there is a way forward from nodes and arcs, and that the complex software of today is capable of being (and should be) visualised. As Myers [15] concludes so succinctly:

> ``*The success of spreadsheets demonstrates that if we find the appropriate paradigms, graphical techniques can* revolutionize *the way people interact with computers."*

Visualisation for the sake of it may well produce pretty pictures but the visualisations developed for program and system comprehension should have two aims; to reduce the complexity of the perceived view of the software and to increase the user's understanding of the software.

## Acknowledgements

Much of this work has been financed by an EPSRC ROPA grant: VVSRE; **V**isualising **S**oftware in **V**irtual **R**eality **E**nvironments.

## References

1. C. Knight and M.Munro, *Comprehension with[in] Virtual Environment Visualisations*, Proceedings of the IEEE 7th International Workshop on Program Comprehension, pp4-11, May 5-7, 1999.
2. T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster, *Program Understanding and the Concept Assignment Problem*, Communications of the ACM, Vol. 37, No. 5, pp72-82, May 1994.
3. D. J. Robson, K. H. Bennett., B. J. Cornelius, and M. Munro, *Approaches to Program Comprehension*, Journal of Systems and Software 14, pp79-84, 1991.

4. A. Von Mayrhauser and A. M. Vans, *Program Comprehension During Software Maintenance and Evolution*, IEEE Computer, pp44-55, August 1995.
5. T. A. Corbi, *Program Understanding: Challenge for the 1990s*, IBM Systems Journal, Vol. 28, No. 2, pp294-306, 1989.
6. P. W. Oman and C. R. Cook, *Typographic Style is More than Cosmetic*, Communications of the ACM, Vol. 33, No. 5, pp506-520, May 1990.
7. P. S. Chan and M. Munro, *PUI: A Tool to Support Program Understanding*, Proceedings of the IEEE 5th International Workshop on Program Comprehension, pp192-198, May 28-30, 1997.
8. S. Wiedenbeck, *The Initial Stage of Program Comprehension*, International Journal of Man-Machine Studies, Vol. 35, pp517-540, 1991.
9. M.-A. D. Storey, F. D. Fracchia, and H. A. Müller, *Cognitive Design Elements to Support the Construction of a Mental Model During Software Visualization*, Proceedings of the 5th IEEE International Workshop on Program Comprehension, pp17-28, May 28-30, 1997.
10. M. Munro and D. J. Robson, *An Interactive Cross Reference Tool For Use in Software Maintenance*, Proceedings of the 20th Annual Hawaii International Conference on System Sciences, pp64-70, 1987.
11. J. R. Foster and M. Munro, *A Documentation Method Based on Cross-Referencing*, Proceedings of the IEEE Conference on Software Maintenance, pp181-185, 1987.
12. N. T. Fletton and M. Munro, *Redocumenting Software Systems using Hypertext Technology*, IEEE Conference on Software Maintenance, pp54-59, 1988.
13. L. D. Landis, P. M. Hyland, A. L. Gilbert, and A. J. Fine, *Documentation in a Software Maintenance Environment*, Proceedings of the IEEE Conference on Software Maintenance, pp66-73, 1988.
14. A. Von Mayrhauser, A. M. Vans, and A. E. Howe, *Program Understanding Behaviour during Enhancement of Large-scale Software*, Journal of Software Maintenance: Research and Practice, Vol. 9, pp299-327, 1997.
15. B. A. Myers, *Taxonomies of Visual Programming and Program Visualization*, Journal of Visual Languages and Computing, Vol. 1, pp97-123, 1990.
16. B. A. Price, R. M. Baecker, and I. S. Small, *A Principled Taxonomy of Software Visualization*, Journal of Visual Languages and Computing, Vol. 4, No. 3, pp211-266, 1992.
17. G. C. Roman and K. C. Cox, *A Taxonomy of Program Visualization Systems*, IEEE Computer, pp11-24, December 1993.
18. G-C Roman and K. C. Cox, *Program Visualization: The Art of Mapping Programs to Pictures*, Proceedings of the 14th International Conference on Software Engineering, pp412-420, May 1992.
19. M. J. Baker and S. G. Eick, *Space Filling Software Visualization*, Journal of Visual Languages and Computing, Vol. 6, pp 119-133, 1995.
20. M-A. D. Storey and H. A. Müller, *Manipulating and Documenting Software Structures Using SHriMP Views*, Proceedings of ICSM '95, pp 275-284, October 17 - 20, 1995.
21. V. Rajlich, N. Damaskinos, and P. Linos, *VIFOR: A Tool for Software Maintenance*, Software Practice and Experience, Vol. 20, No. 1, pp67-77, January 1990.
22. V. Rajlich and S. R. Adnapally, *VIFOR 2: A Tool for Browsing and Documentation*, IEEE International Conference of Software Maintenance, pp 296-300, 1996.
23. P. Linos, P. Aubet, . Dumas, Y. Helleboid, P. Lejeune, and P. Tulula, *Facilitating the Comprehension of C Programs: An Experimental Study*, Proceedings of the 2nd IEEE Workshop on Program Comprehension, pp55-63, July 8-9, 1993.

24. P. K. Linos and V. Courtois, *A Tool for Understanding Object-Oriented Program Dependencies*, Proceedings of the 3rd IEEE Workshop on Program Comprehension, pp20-27, November 14-15, 1994.
25. S. G. Eick, *Engineering Perceptually Effective Visualizations for Abstract Data*, In Scientific Visualization Overviews, Methodologies and Techniques, IEEE Computer Science Press, pp191-210, February 1997.
26. D. F. Jerding and J. T. Stasko, *Using Information Murals in Visualization Applications*, Proceedings of UIST '95 (ACM), November 14-17, 1995.
27. 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.
28. F. P. Brooks, *No Silver Bullet*, IEEE Computer, pp10-19, April 1987.
29. P. Young, *Three Dimensional Information Visualisation*, University of Durham, Computer Science Technical Report 12/96, 1996.
30. L. Feijs and R. de Jong, *3D Visualization of Software Architectures*, Communications of the ACM, Vol. 41, No. 12, pp72-78, December 1998.
31. P. Young and M. Munro, *Visualising Software in Virtual Reality*, Proceedings of the IEEE 6th International Workshop on Program Comprehension, pp19-26, June 24-26, 1998.
32. P. Young, *Visualising Software in Cyberspace*, PhD Thesis, University of Durham, October 1999.
33. C. Knight and M. Munro, *Visualising Software – A Key Research Area*, Proceedings of the IEEE International Conference on Software Maintenance, August 30 – September 3, 1999. (Short paper.)
34. C. Knight and M. Munro, *Virtual but Visible Software*, Proceedings of the IEEE International Conference on Information Visualisation, July 2000, (in publication).
35. C. Knight and M. Munro, *Should Users Inhabit Visualisations?*, to appear in the post proceedings of Knowledge Media Networking workshop of IEEE WETICE, June 2000.
36. G. Walker, *Challenges in Information Visualisation*, British Telecommunications Engineering Journal, Vol. 14, pp17-25, April 1995.
37. T. Ball and S. G. Eick, *Software Visualization in the Large*, IEEE Computer, pp33-43, April 1996.
38. N. Chapin and T. S. Lau, *Effective Size: An Example of Use from Legacy Systems*, Journal of Software Maintenance: Research and Practice, Vol. 8, pp101-116, 1996.
39. H. Rheingold, *Virtual Reality*, Mandarin Science, 1992.
40. R. Hubbold, A. Murta, A. West, and T. Howard, *Design Issues for Virtual Reality Systems* Presented at the First Eurographics Workshop on Virtual Environments, 7 September 1993.
41. M. Crossley, N. J. Davies, R. J. Taylor-Hendry, and A. J. McGrath, *Three-dimensional Internet Developments*, BT Technology Journal, Vol. 15, No. 2, pp179-193, April 1997.
42. C. Knight, *Virtual Software in Reality*, PhD Thesis, University of Durham, June 2000
43. B. Hendley and N. Drew, *Visualisation of complex systems*, University of Birmingham (UK) School of Computer Science, 1995.
44. R. J. Hendley, N. S. Drew, A. M. Wood, and R. Beale, *Narcissus: Visualising Information*, University of Birmingham, Advanced Interaction Group Technical Report, 1995.
45. J.D. Mackinlay, S. Card and G.G. Robertson, *Perspective Wall: Detail and Context Smoothly Integrated*, Proceedings of the ACM SIGCHI '91 Conference on Human Factors in Computing Systems, pp. 173-179, April 1991.
46. 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, pp. 189 - 194, April 1991.

47. K.M. Fairchild, *Information Management Using Virtual Reality-Based Visualizations*, in ``Virtual Reality: Applications and Explorations", Alan Wexelblat (ed.), Academic Press Professional,
Cambridge, MA, pp. 45-74, 1993
48. M. Hemmje, *A 3D Based User Interface for Information Retrieval Systems*, Proceedings of IEEE Visualization '93, Workshop on Database Issues for Data Visualization, October 25-29, 1993
49. M. Hemmje, C. Kunkel and A. Willet, *LyberWorld - A Visualization User Interface Supporting Fulltext Retrieval*, Proceedings of ACM SIGIR '94, July 3-6, 1994
50. R. Tamassia, G. D. Battista, and C. Battini, *Automatic Graph Drawing and Readability of Diagrams*, IEEE Transactions on Systems, Man, and Cybernetics, 18(1), pp61-79, January/February 1988.