A New View of Call Graphs for Visualising Code Structures

Peter Young and Malcolm Munro

Visualisation Research Group

Research Institute in Software Evolution
(formerly The Centre for Software Maintenance)

Department of Computer Science
University of Durham
Durham, DH1 3LE, UK

Computer Science Technical Report 03/97

For more information please http://vrg.dur.ac.uk/
Peter Young left VRG in 1999
PDF created by Claire Knight, 11th March 2002

Keywords: software visualisation, call graphs, virtual reality, program comprehension

Abstract

Software visualisation promises to provide useful techniques for supporting the program comprehension process. One popular view of software structure is the call-graph which displays the relationships between functions in a program. Current software tools visualise these call-graphs as a directed graph. Typical software systems are very large and extremely complex, as a result they generally produce large and complex graphs which often present as much of a comprehension task as they attempt to address. This paper presents a new method for visualising call-graphs which moves away from the conventional node-link depiction. This new visualisation which uses a virtual reality environment shows how some of the conventional problems of call-graph visualisation can be overcome.

A New View of Call Graphs for Visualising Code Structures

Peter Young and Malcolm Munro

Visualisation Research Group
The Centre for Software Maintenance

Department of Computer Science
University of Durham
Durham, DH1 3LE, UK

E-mail : *peter.young@durham.ac.uk*

Keywords: software visualisation, call graphs, virtual reality, program comprehension

Abstract

Software visualisation promises to provide useful techniques for supporting the program comprehension process. One popular view of software structure is the call-graph which displays the relationships between functions in a program. Current software tools visualise these call-graphs as a directed graph. Typical software systems are very large and extremely complex, as a result they generally produce large and complex graphs which often present as much of a comprehension task as they attempt to address. This paper presents a new method for visualising call-graphs which moves away from the conventional node-link depiction. This new visualisation which uses a virtual reality environment shows how some of the conventional problems of call-graph visualisation can be overcome.

1. Introduction

The term software visualisation has almost as many taxonomies as definitions [Price93, Roman93, Myers90], each one offering a slightly different meaning and classification. The most fitting definition is that which encompasses all aspects of visually displaying a software system. This definition includes all possible views of the software and all possible representations, which can range from source code listings through to the most abstract of graphical visualisations. The term software visualisation is generally taken to include the areas of algorithm animation and program visualisation, which is in turn comprised of data visualisation and code visualisation, both either static or dynamic.

It is a well known fact that when attempting to maintain a software system, the maintainer must first gain some understanding of that system. This understanding can come in numerous forms which may range from the system structure at various abstraction levels, to gaining knowledge of the program control flow or message passing mechanisms. It has long been known that when gaining this understanding the maintainer will construct an internal cognitive model [Mayrhauser95] of whatever aspect of the system they are interested in. Typically, to construct this cognitive model the maintainer will extract information directly from the source code and gradually piece together the software puzzle. This method of information gathering is far from efficient and requires the maintainer to continuously shift focus to various sections of the code and actively search for the information they require. What is needed is a method of presenting the necessary information in a readily understandable and intuitive manner which corresponds more closely to the structure of the cognitive model created by the maintainer.

A popular visualisation of software systems is the call-graph. A call-graph displays call or execution relationships between discrete sections of the software. For example, in the C language a call-graph would display each function within the program and the call relations between them. The established visualisation of a call-graph is displayed as a directed graph where the functions are represented as nodes and the call relations as arcs. This paper describes some of the problems associated with this conventional view and introduces a new technique for visualising call-graphs which takes advantage of 3D graphics and virtual reality technology.

2. Conventional representations

A large number of software tools are available to aid the maintainer in understanding a software system, these range from maintenance or comprehension aids to full software development environments, for example McCabe, Logiscope [Meekel88] and SNiFF+[1]. These tools provide graphical representations of some aspect of the software under analysis, typically a function call graph, class inheritance hierarchy or control flow graph. The graphs produced by most tools are unfortunately not very well presented and some use layout criteria which are not suitable for the data being displayed, such as symmetry and regular spacing of nodes. Control flow graphs generally are more aesthetically pleasing due to the inherent tree-like structure of these graphs. Call graphs do not typically possess such an organised structure and result in a complex directed graph which is inherently difficult to layout well. One important feature which is present in most tools is the ability to move from the visualisations to the source code fairly easily, thus allowing the maintainer to switch between the low-level detail and the more abstract visualisations.

The views available in software tools such as those above have the potential to provide valuable structural information but are often badly displayed and unreadable making them more of a hindrance than an aid. A large number of tools appear to have put little effort into the layout or presentation of their graphical representations, apparently adopting the view that any visualisation will help. Unfortunately this is not the case, the layout and presentation of the visualisation could be construed as being of greater importance than the actual content of the visualisation. For instance, a badly presented graph containing key structural information could be extremely hard to read or follow, possibly resulting in the engineer spending valuable time deciphering the information or even gaining no information at all. On the other hand, a well presented and readable graph which contains little useful information could be quickly dismissed allowing time to be spent more profitably elsewhere.

The layout and presentation of visualisations is extremely important and active research is being carried out in creating new views and representations of software which provide an intuitive and useful visualisation. The majority of software views are visualised as directed graphs which are themselves the subject of presentation and layout research, though this research does not cater for the content and meaning of the graphs as they apply to software. Standard 2D call graphs are a popular choice of code visualisation, though for any reasonably large system these become overly complex and unreadable [Storey95]. The problem unfortunately lies with the typically extremely large set of relationships and dependencies between software components such as functions, classes and data.

Using standard 2D graphing techniques even relatively small graphs (when considering software systems) will soon clutter the display making an unreadable layout (Figure 1). Features such as a scrollable window or allowing nesting through multiple windows do make the graph more readable, however only a small area is visible at any one time and it is easy to lose context with regard to the whole graph. Three-dimensional directed graph representations (for example, Figure 2) come some way to alleviating the problems of display size, though at present these views merely move the threshold back a little and still become overly complex for large graphs.

Graph visualisations of software systems suffer greatly from the sheer size and complexity of the information structures which they represent. Various techniques and strategies have been developed for presenting, simplifying and exploring such structures. Graph reduction is concerned with simplifying the graph structure by applying various strategies such as hiding less interesting information, clustering semantically related nodes and nesting nodes and arcs as appropriate [Burd96]. Methods for exploring and recovering these reduced graphs are also very important. Graph presentation techniques generally consist of applying a layout algorithm to the graph structure and driving it with goals such as minimising crossing arcs or grouping related nodes [Di Battista94]. These algorithms are of little use when applied to the complex graphs produced from a typical software system.

---

[1] McCabe by McCabe & Associates,
Logiscope by Verilog
SNiFF+ by TakeFive Software

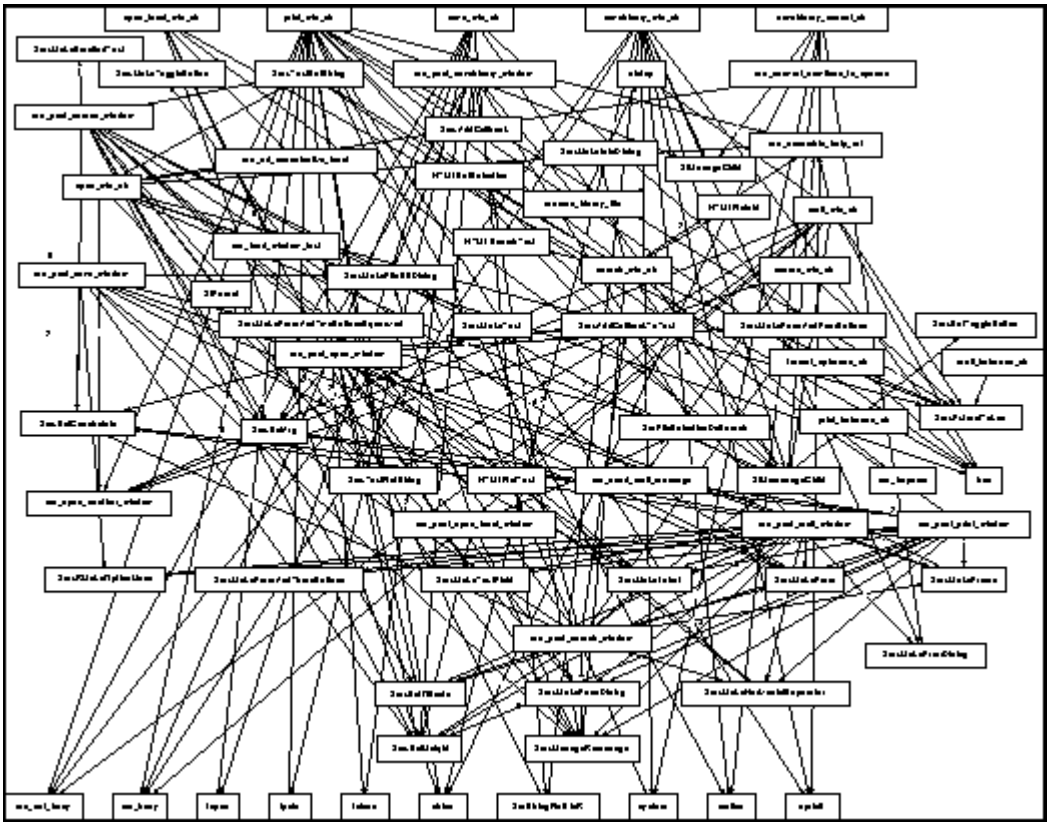

Figure 1. Typical call-graph visualisation for a medium sized module.

Figure 1 shows a call-graph for one module of the Mosaic[2] source code. This graph has already been simplified significantly by combining multiple calls between functions into a single arc.

Visualisation tools have been developed such as the SHriMP views [Storey95] incorporated in Rigi which use fisheye techniques and nested graphs to allow focusing on particular areas of the graph while still retaining a view of the whole structure. The fisheye technique distorts the view on the graph similar to a fish-eye lens, objects near to the centre of the view are magnified greatly with this magnification lessening rapidly with distance from the centre. This view results in objects which are the centre of attention being shown in greatest detail whereas objects on the periphery are shown in lesser detail. Fisheye views allow objects of interest to be studied in detail while still maintaining a view of the context or position with respect to other objects.

3. Alternate representations

The previous section highlighted some of the problems associated with traditional visualisations of software systems and call-graphs in particular. This section will introduce a new technique for visualising code structure information which makes use of the advantages afforded by 3D graphics and virtual environments. We have developed a number of 3D software visualisation prototypes which present different aspects of code structure in a variety of styles and metaphors. One such visualisation which dispenses with the traditional directed graph presentation of call-graphs is described below in section 3.3.

3.1. Virtual reality

Virtual reality has received a lot of attention recently, particularly in the press, and has led to many misconceptions and exaggerated claims [Gigante93]. The goals of Virtual Reality are relatively clear, that of creating the illusion of submersion within a computer generated environment. Although this concept is generally associated with various novel interface and display techniques, these are not seen as a pre-

---

[2] NCSA Mosaic for the X window System,
Copyright (C) 1993, Board of Trustees of the University of Illinois

requisite for virtual reality more as an aid or enhancement. A very high level of involvement or feeling of immersion within a virtual environment can be achieved using standard interfaces and displays (i.e. mice and monitors), the computer games industry being a prime example of this success. In our opinion the virtual reality is not created by the realism of the images or quality of the rendering, it is created by the ability to interact and navigate comfortably through a computer generated environment.

Virtual reality and 3D graphical environments are already becoming commonplace on home computers and are increasingly popular with computer games. Although the methods for interacting with these environments remain relatively simple, they still provide a very immersive experience. Such technology has been exploited greatly in the area of data visualisation with excellent results, and it is being used increasingly in information and software visualisation [Young96]. One of the main advantages of using a VR display for visualising information is the ability to move freely throughout the data and obtain a viewpoint on it from any conceivable angle or position.

3.2. Application of 3D to software visualisation

The application of 3D graphics to software visualisation should not be seen as a cure-all for visualisation problems, it simply provides another presentation medium which allows a greater spatial freedom. It is unlikely that 3D software visualisations will completely replace more traditional 2D methods though they will hopefully provide a useful addition to them. As with 2D software visualisations, 3D visualisations have both advantages and disadvantages. Some of the disadvantages can be easily experienced when first attempting to navigate a 3D visualisation, such as:

- Initial confusion and disorientation combined with the additional complexities of the interface and greater freedom of movement. Navigating with six degrees of freedom *within* an information space as opposed to navigating a set of 2D windows *on* an information space through panning and zoom controls.
- Large processing overheads are evident when a typically fast computer is reduced to a crawl by a complex visualisation.
- Navigating any 3D environment requires a level of spatial awareness from the user. Navigating within an abstract information space demands more from this skill. Such spaces do not typically conform to our preconceptions of a 3D environment and various expectations such as a notion of “up” and “down” can be easily violated.

3D virtual environments also offer a number of advantages for visualising information or software systems. Some of these are:

- A greater working volume for information presentation with focus, context and point of interest managed dynamically by the viewpoint. With standard 2D views you are limited to a variable sized 2D canvas which can be panned, expanded, reduced or linked together. With a 3D environment you have a much larger working volume and can also link worlds together. Additionally, the 3D perspective view means that while investigating items of interest the remainder of the information in the world is still apparent which aids in maintaining a notion of context and location within the visualisation.
- 3D graphics offer a greater level of flexibility for representing, organising and presenting the information.
- Human experience in interacting and exploring within 3D environments is exploited.

The conventional representation of a call-graph can be exported to 3D by for example using the force-directed placement algorithm [Fruchterman91] to perform a simple and effective layout (Figure 2). This generally alleviates crossing lines when used in the 3D environment though because the user effectively has a 2D viewpoint on this environment then there will inevitably be arcs which appear to intersect. The ability to move the viewpoint and utilise motion parallax does aid considerably in comprehending such structures [Ware94]. 3D graph structures offer some improvement but do still suffer from most problems associated with 2D graphs.

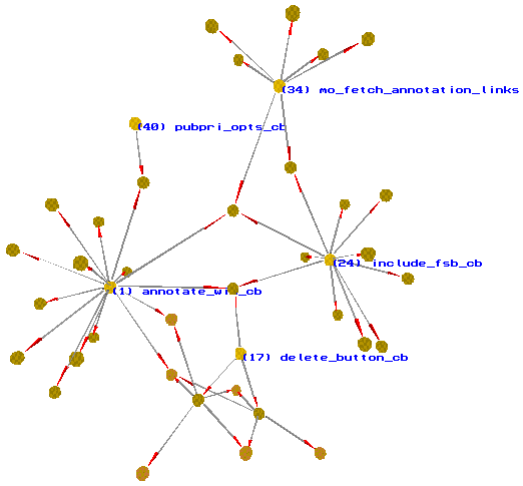

Figure 2. 3D visualisation of a call-graph.

This graph shows a simple extension of the 2D call-graph into 3D space. Layout is performed using the force directed placement algorithm. The advantage of using 3D here is that the graph can be viewed from any position or angle and the user can take advantage of motion parallax to aid comprehension of its structure.

3.3. The CallStax visualisation

CallStax is a visualisation which attempts to move away from the standard visualisations of call-graph structures, i.e. a network consisting of nodes and arcs. CallStax makes full use of the extra dimension afforded by VR to maximise the amount of information available and the flexibility for displaying and interacting with that information.

It is not generally the number of components in the call-graph which complicates the visualisation, rather it is the typically much larger set of relationships between these components. Simple operations which 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 which follow these nodes wherever placed. CallStax attempts to reduce the complexity overhead which these explicit relationships place on the visualisation by making them implicit. This effectively reduces the complexity of the visualisation, but increases the cognitive load of 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. Hopefully, 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 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 is different in that it visualises the *paths* through the graph rather than the graph as a network.

CallStax visualises each possible path through the 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, 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 CallStax visualisation of a typical software system will result in a large number of stacks, each positioned arbitrarily through 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 power of the CallStax visualisation lies in it's flexibility. The stacks are not explicitly connected in any way which affords great freedom in the positioning, grouping, insertion and deletion of stacks. Additionally, the scale of the system under scrutiny does not adversely affect the visualisation to the same extent as in the standard call-graph. Unfortunately this flexibility comes at a price. The implicit relationships between duplicate representations of functions is 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 (see example below) but it becomes very difficult when a large number of distinct functions are present. Within a standard call-graph, it is the position and the node label which 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).

The following example shows the construction of a CallStax visualisation using a simple program as the basis. Figure 3 shows a standard 2D call-graph of the program "`lines.c`". The nodes on the graph have been coloured to show the main functions belonging to the program, whereas the plain 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 4 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.

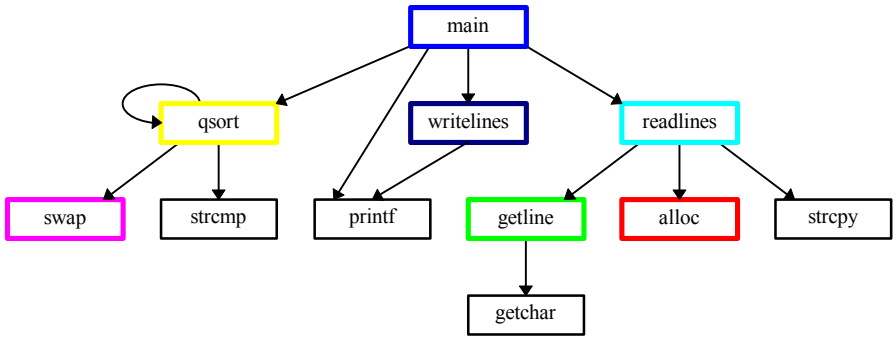

Figure 3. Standard 2D call-graph of a simple program.

Coloured nodes represent functions local to the program while other nodes represent library functions.

As can be seen from the Stacks shown in Figure 4, the CallStax visualisation contains the same information presented in the standard call-graph. 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 4 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[3] shows the same CallStax visualisation but shown in a 3D perspective using the Superscape VR package.

[3] Please note that the colours used in Figures 5, 6 and 7 have been modified to produce a white background as opposed to black. This has been done purely to improve image quality in print.

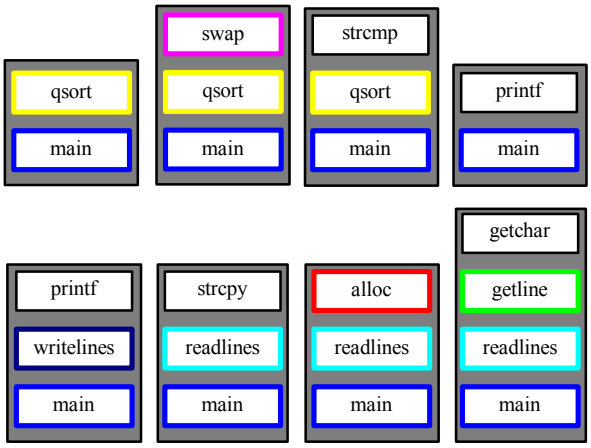

Figure 4. 2D CallStax visualisation of the graph shown in Figure 3.

This figure shows a 2D depiction of the CallStax created from the graph shown in Figure 3. As previously mentioned, each stack represents a path through the graph.

Figure 5 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 indicated by a translucent mesh. All stacks which 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. Similarly, the selection mesh contains holes which correspond 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, it's depth within the call hierarchy can be rapidly found from the stack which extends the furthest down from the selection plane.

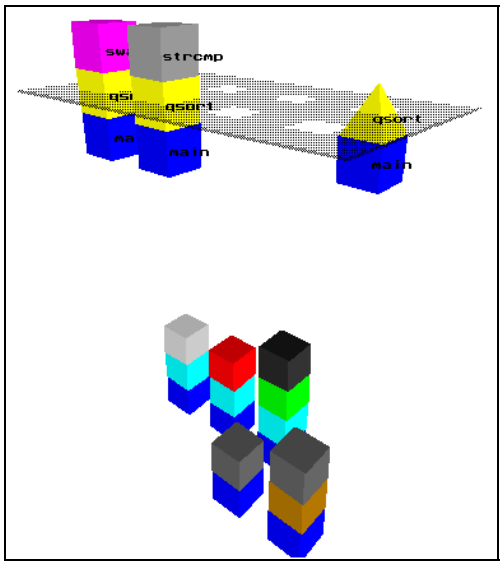

Figure 5. CallStax visualisation shown in 3D using Superscape VR package.

All occurrences of the currently selected function, *qsort*, are aligned in a horizontal plane. Any stacks not containing qsort recede to the bottom of the view.

Function call information only gives 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 the original basic representation of the function will change to a more detailed visualisation of the component and its properties (Figure 6). The representation shown here displays simple structural and metrics 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.

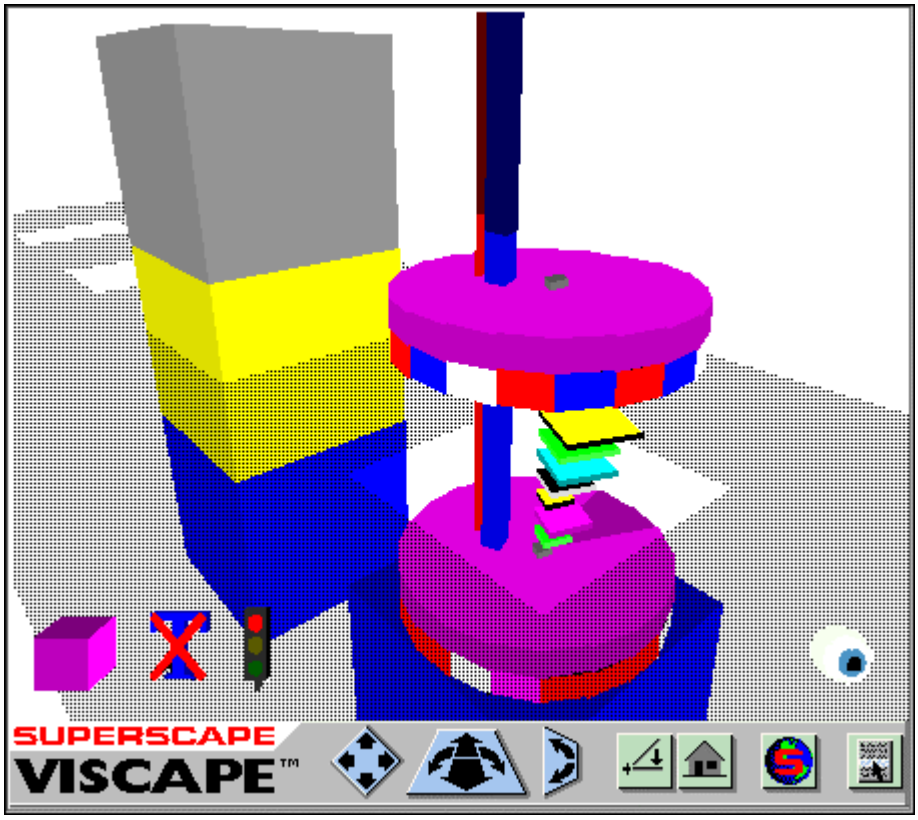

Figure 6. CallStax zoomed in view revealing further detail.

This figure illustrates how the user can move closer to a particular area of interest within the visualisation and obtain more detailed information.

The more detailed function representations shown in Figure 6 are revealed as the user approaches one of the stacks. Each of these representations displays metric details and simple structural information about the function. The metric details are represented by the two vertical bar-charts which display the McCabe complexity and Lines of Code metrics as a percentage of the maximum values in this program. The base of each representation is a pie-chart which shows 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 6, the function *swap* is represented at the top of the stack with *qsort* below it.

Also shown in Figure 6 is a simple widget set which allows the user to control both 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.

As previously mentioned, 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 WWW technology which allows for a rich multimedia presentation. Figure 7 shows the CallStax visualisation integrated into a WWW presentation. In this example a syntax highlighted source code window and 2D call graph are also available. The user can 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.

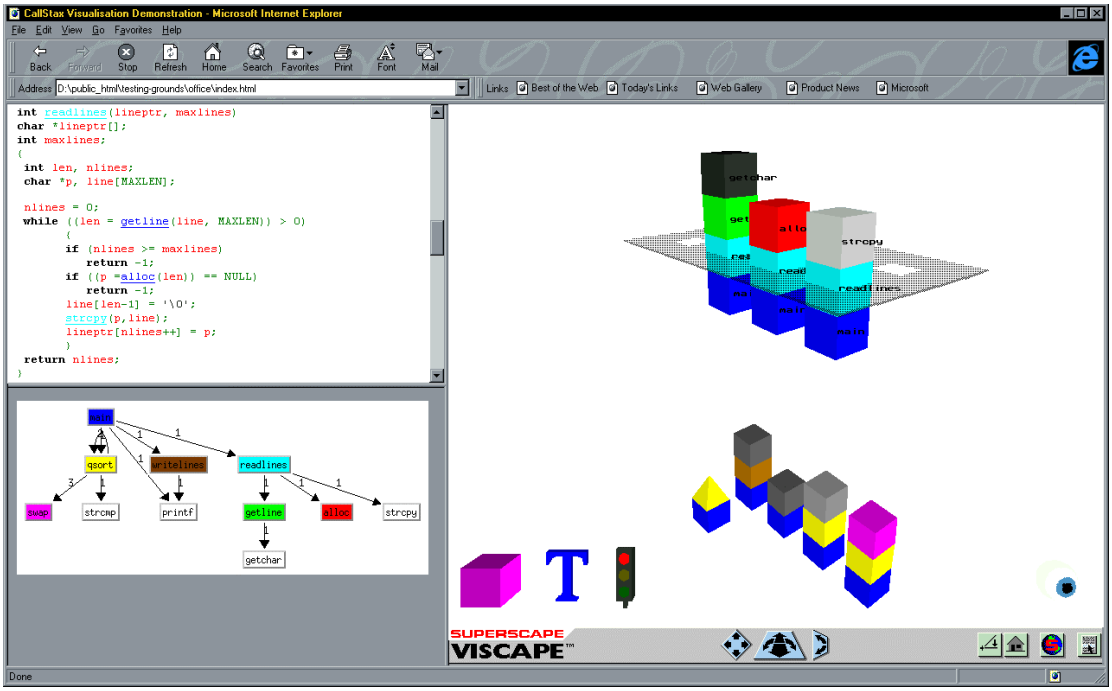

Figure 7. CallStax visualisation integrated with other views.

Figure 7 shows the CallStax visualisation integrated into a WWW presentation and combined with more conventional views of the software system, namely the syntax highlighted source code and a standard 2D call graph.

4. Conclusions

Standard 2D call-graphs are an excellent medium for conveying the relationships between components (i.e. functions) within software. Unfortunately, graphs such as these suffer from a variety of problems mostly associated with the complexity and scale of the information they have to present. For this reason, call-graphs rapidly lose their usefulness as the size and complexity of the software they are presenting increases. The limitations of the 2D graph are highlighted when viewing the relationships within large software systems. The graphs will rapidly become very messy and unreadable, with little or no hope of finding an acceptable layout.

The conclusions of this paper are that graphical representations are important for program comprehension. Software visualisation is complicated greatly by the size and complexity of typical software systems, all visualisations have both their own merits and shortcomings, the problem in hand is to find a suitable and effective compromise. New methods and techniques for visualising software systems, such as the CallStax described here, show potential. The new representation in a 3D virtual world has the advantages of :

- No multiplicity of crossing lines;
- Greater flexibility in grouping functions and removing unwanted information;
- Viewpoint distance gives greater or lesser detail;
- Viewpoint position allows focusing of areas of interest while maintaining context.

There is a need to explore other representations, visual abstractions and appropriate metaphors - particularly the possibilities afforded by 3D graphics and VR technology.

An on-line demonstration of the CallStax visualisation can be found at the following URL. In order to view the visualisation correctly you will require the Viscape plug-in application, details of which can also be found at this location.

http://www.dur.ac.uk/~dcs3py/testing-grounds/demo.html

References

[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,
Department of Computer Science, University of Durham, 1996.
http://www.dur.ac.uk/~dcs3py/pages/work/Documents/tr-10-96

[Di Battista94] **G. Di Battista**, **R. Tamassia**, **P. Eades** and **I.G. Tollis**
Algorithms for Drawing Graphs: an Annotated Bibliography,
June 1994.
ftp://wilma.cs.brown.edu/pub/papers/compgeo/gdbiblio.ps.Z

[Fruchterman91] **T.M.J. Fruchterman** and **E.M. Reingold**,
Graph Drawing by Force-Directed Placement,
Software Practice and Experience,
Vol. 2, No. 11, November 1991.

[Gigante93] **M.A. Gigante**,
Virtual Reality: Definition, History and Applications,
in *Virtual Reality Systems*, Academic Press Ltd.,
pages 3-14, 1993.

[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,
In Proceedings of the ICSM '88 Conference on Software Maintenance,
pages 328-334, 1988.

[Myers90] **B.A. Myers**,
Taxonomies of visual programming and program visualisation.
Journal of Visual Languages and Computing, Vol. 1, pages 97-123, 1990.

[Price93] **B.A. Price**, **R.M. Baecker** and **I.S. Small**,
A Principled Taxonomy of Software Visualisation.
Journal of Visual Languages and Computing Vol. 4, No. 3, pages 211-266, 1993.

[Roman93] **G-C. Roman** and **C. Cox**,
A Taxonomy of Program Visualisation Systems,
IEEE Computer, Vol. 26, No. 12, pages 11-24, December 1993.

[Storey95] **R. Storey** and **H.A. Müller**,
Manipulating and Documenting Software Structures Using SHriMP Views.
International Conference on Software Engineering,
Opio (Nice), France, pages 275-284, October 17-20, 1995.

[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,
St. Louis, Missouri, USA, pages 182-183, October 1994.

[Young96] **P. Young**,
Three Dimensional Information Visualisation,
Technical Report 12/96, Centre for Software Maintenance,
University of Durham, March 1996.
http://www.dur.ac.uk/~dcs3py/pages/work/Documents/index.html