Visualization of Large Nested Graphs in 3D: Navigation and Interaction

Greg Parker, Glenn Franck and Colin Ware
Faculty of Computer Science
University of New Brunswick
P.O. Box 4400, Fredericton, NB.
CANADA E3B 5A3
gparker@omg.unb.ca

Abstract

Most systems for visualizing large information structures use 2D graphics to view networks of nodes and arcs that represent data. To understand large structures it is often necessary to show both small-scale and large-scale structure. This has been called the problem of focus and context. Distortion, rapid zooming, elision, and multiple windows are all techniques that have been developed to provide both focus and context within single representations. We review these techniques and argue that 3D visualization has a number of advantages. A system called NestedVision3D (NV3D) will be presented that has been developed to investigate the use of 3D visualization for understanding the structure of large computer programs.

NV3D is a system for visualizing large nested graphs using interactive 3D graphics. It has been tested with graphs containing more than 35,000 nodes and 100,000 relationships. We describe NV3D and its design philosophy. Basic navigation is facilitated by a set of 3D widgets, rapid scaling, and interactive elision. More experimental features include animations called snakes, which are used to trace dynamic software behavior.

Introduction

A common way to visually represent information structures is via a network of nodes and arcs technically called a graph. The nodes represent entities, such as procedures, modules, or objects in an object-oriented computer program, while the arcs represent relationships between entities, perhaps method usage or inheritance. Common graphs of this kind include entity-relationship diagrams used in database modeling, Myers diagrams used to represent module relationships in software, call graphs showing calling relationships between procedures, inheritance graphs showing the pattern of inheritance in object-oriented code, and data flow diagrams used in systems modeling. The shape and the color of the nodes and arcs can be altered to represent different kinds of entities and different kinds of relationships.

Generally, diagrams of this type do not scale easily. Once some limit is reached, perhaps twenty entities and thirty relationships, the diagram becomes a tangled, incomprehensible web. A traditional solution to this problem is to modularize the diagram and create subsidiary diagrams on different pages. Recently, a number of experimental systems have been developed that use interactive graphics to help solve the scale problem without switching pages. The system we present here, NestedVision3D (NV3D), is one such system. Figure 1 shows a sample view taken from this system.

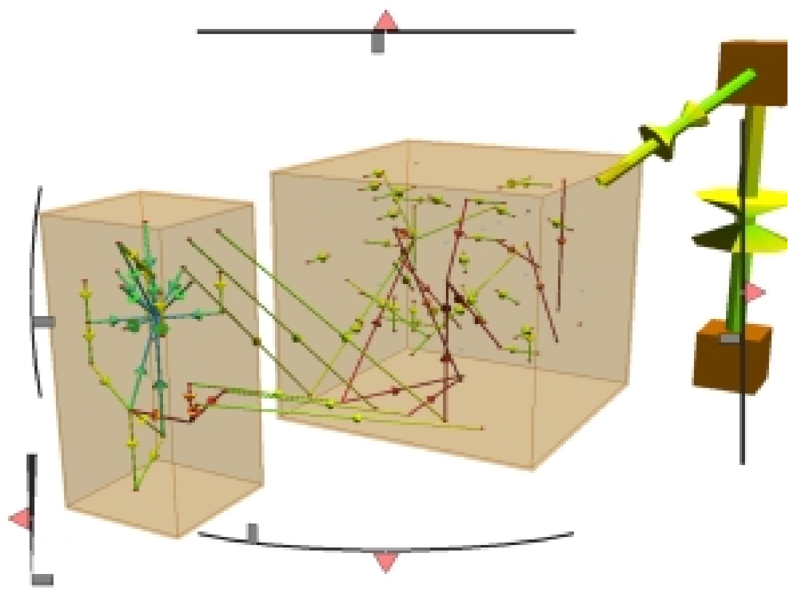

Figure 1. An example graph displayed in NestedVision3D. This data represents part of a software system being analyzed for Year 2000-related date problems.

NV3D uses 3D representation, rapid navigation techniques and nested graphs to help us understand large information structures. It is currently being applied to projects in software reverse engineering, particularly for Year 2000 diagnosis. However, it also has a parallel existence in the research lab where we are investigating techniques that use animation to help us look at the temporal behavior of computer programs superimposed on the software structure. In the present paper we begin by describing the basic features that allow NV3D to help us understand large information structures, and we continue by describing "snakes". Snakes are a method we have developed for understanding multi-threaded applications.

Background

The problem of viewing large graphs is one of focus and context. It is important to provide information about the large-scale structure of the graph, while at the same time allowing users to drill down to an arbitrary level of detail. In the following discussion we consider five techniques to solve the focus-context problem: distortion, rapid zooming, elision, multiple windows and 3D viewing.

Distortion techniques

A number of techniques have been developed that spatially distort a graph. Distortion gives more room to designated points of interest and decreases the space given to those objects away from these points. Some techniques have been designed to work with a single focus, such as the hyperbolic lens [10]. Others allow for multiple foci to be simultaneously expanded [12]. Many of these methods use simple algebraic functions to distort the graph based on the distance from each focus. An alternative method called "intelligent zooming" [5,14] uses techniques of graph layout to dynamically resize and reposition parts of a graph based on selected points of interest (Note that this is not the same as the simple rescaling of the image described in the next section). The basic concept of all distortion techniques is to spatially expand what is currently interesting at the expense of what is not, thus providing both focus and context.

Rapid zooming techniques

In rapid zooming techniques a large information landscape is provided, but only a part of it is visible through the viewing window at any instant. The user is given the ability to rapidly zoom in to and out of points of interest, which means that although focus and context are not simultaneously available, the user can rapidly and smoothly move from focus to context and back. This may allow the user to cognitively integrate the information. The Pad and Pad++ systems are designed around this concept [2].

Note that there are a number of 3D techniques that are loosely referred to as "zooming". These include changing the camera focal length, moving the camera toward an object, and scaling an object up. The result in all cases is that part of the scene occupies a larger area of the viewing window. In this paper, we use the term "zooming" to refer to any technique which results in this effect, regardless of the actual algorithm used.

Mackinlay et al. developed a rapid navigation technique for 3D scenes that they called POI navigation [11]. This method moves a user in towards a point of interest that has been selected on the surface of some object. At the same time, the viewpoint of the viewer is brought to a point that is perpendicular to the surface.

Elision techniques

Elision is a technique where parts of the structure are hidden until they are needed. Typically, this is achieved through collapsing a node that contains a sub-graph into a single node. In the intelligent zoom system [1], when a node is opened, it expands to reveal its contents while simultaneously adjusting the entire graph to make more space for the expanded node. Any number of nodes can be expanded in this way and to an arbitrary depth, all the while the graph is continuously adjusted, creating more space for objects of interest and correspondingly reducing the size of other parts of the graph. Thus rescaling and elision work hand in hand.

The elision idea can be applied to text as well as graphics. In the "Generalized Fisheye" technique for viewing text data [7], less and less detail is shown as the distance from the focus of interest increases. Thus, for example, when viewing programming code, the full text is shown at the focus, while further away only the subroutine headers are made visible and the code internal to the subroutine is elided.

In elision methods, eliding blocks of structure into a single more compact representation provides context. Clearly this kind of technique works well in nested graph structures because entire sub-graphs can be collapsed into representative nodes.

Multiple windows

It is common, especially in mapping systems, to have one window that shows an overview and several others that show expanded details [2]. In Pad++, multiple windows are called portals and each portal can be individually zoomed. However, one problem with multiple window techniques can be that the details are disconnected from the overview; it can then be difficult to see where the zoomed details belong in the overview window. A solution is to use lines to connect the boundaries of the zoom window to the source image in the larger view [16].

3D interactive visualization

3D visualization provides focus and context through the operation of linear perspective. Objects seen in the foreground have more detail than those further away. By changing the viewpoint, the foreground, and hence the focus, can be changed. However, only a small part of the context information is typically visible, and the part that is visible can be somewhat arbitrary and unrelated to the current focus. Also, the 3D layout of the scene is critical in determining what context will be visible.

There is evidence that larger structures can be understood better in 3D than in 2D. Cone Trees allow the display of tree structured graphs showing all the children of a node in the form of a cone of subsidiary nodes. It is claimed that as many as one thousand nodes may be displayable using Cone Trees without visual clutter, and this is clearly more than could be contained in a 2D layout [3]. There is also experimental evidence that shows that substantially larger graph structures can be viewed in 3D. The motion parallax depth cue that allows us to see 3D structures, if they are rotated, appears to be especially important [17].

3D or 2D?

One previous major work done on the 3D visualization of information networks is the SemNet project of Fairchild et al. [6]. This project used 3D representation to allow users to visualize large knowledge bases as nodes and arcs in a three dimensional space. We have heard, informally, that the SemNet project was not regarded as a practical or useful system by its developers, although neither this opinion nor the reasons for it are published. However, the design space of systems for visualizing complex relational structures is very large. To create a usable visualization system requires dozens of design decisions, and any one of them, badly made, may move the system below the threshold of usability. It is also the case that graphics performance has increased dramatically since the SemNet project and interactive performance is critical in the utility of these systems. NV3D is a new 3D-visualization solution that we believe has crossed the threshold into usability. We believe that NV3D is the only system that can deal with tens of thousands of entities and both nested and non-nested relationships.

Problems in navigation, layout, and semiology must all be addressed in order to produce an effective 2D or 3D solution. We briefly consider these issues in broad terms here and then present the NV3D solution.

1) *Spatial Navigation.* In 2D-graph visualization the following operations are available: translation in x and y, scaling about a point of interest and rotation about a point orthogonal to the screen (four degrees of freedom). However, rotations are not commonly used. In Pad++, the main operations are selection of points of interest and zooming (scale changes). In intelligent zooming, navigation consists entirely of opening and closing nodes, while spatial positioning is handled automatically. In 3D space, setting up a viewpoint and view direction is a six degree-of-freedom operation. Scaling and moving the viewpoint toward the point of scale are isomorphic transformations. However, if we constrain the up direction in the virtual environment to be the up direction on the screen, then this is reduced to five degrees-of-freedom.

Moreover, since zooming also works in 3D, we can do most of our navigation by means of selecting points of interest and moving the viewpoint in and out in a manner that results in motion similar to navigation in 2D. This is the basis for the POI navigation method [11].

2) *Layout*. The problem of layout is critical to the visualization of graphs, either in 3D or in 2D. If nodes are randomly placed, then the information will simply appear as a confusing jumble. Layout of 2D graphs may actually be easier to do in 3D. Much work has been expended laying out 2D graphs in ways that minimize arc crossings. In a dynamic 3D environment, arc crossings are much less important because 3D arcs are much less likely to intersect. They will not be *perceived* as intersecting arcs because of motion parallax, shading or stereoscopic disparities that enable us to distinguish arcs that lie at different depths. It is also possible to use the extra spatial dimension to encode semantic information. For example, Koike [9] produced a 2D layout of the modules in a message passing system, but used the third dimension as a time axis to display the message sequencing information.

3) *Semiotics*. Semiotics is the discipline that governs the representation of information by means of symbols. This presents a challenge in 3D because 3D objects must be constructed, unlike 2D icons, and should be equally understandable from any viewpoint. Despite this problem there may be an important advantage in using a 3D solution. The *object* is a central metaphor to modern coding. We are encouraged to think of blocks of code as physical objects that have well defined boundaries. The object concept can be understood in a broad sense that goes beyond object-oriented programming. It seems reasonable to claim that a 3D graphical entity that is shaded and viewed in perspective is more *object-like* than a 2D representation (usually a simple rectangle) and therefore the 3D solution better matches the central metaphor.

Our solution to the problems of navigation layout and semiotics, NV3D, has been developed over the past five years and has now reached (we believe) a state of practical utility. Here we present this system as a full scale working 3D solution.

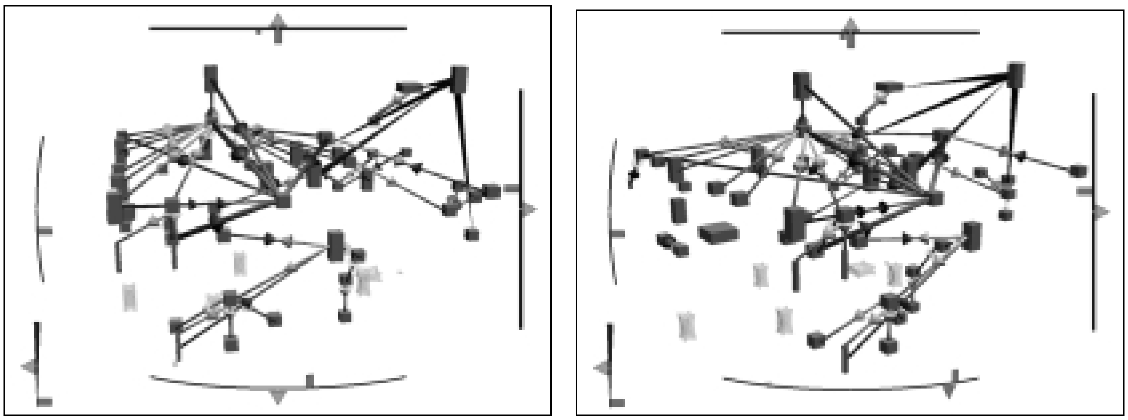

Figure 2. An object-oriented C++ program represented as a 3D graph. The nodes represent files, while the arcs show inheritance, function calls and file inclusion relations differentiated by color. In the first image the rotation widget at the bottom of the scene is in the stationary position. The second image shows the graph after the user has rotated the graph slightly about the x-axis.

NV3D Displays Nested Graphs in 3D

NV3D is a 3D data visualization tool for viewing large relational information structures. It is designed for problems where multiple entity types exist as well as multiple relationships. To consider a specific example, in the case of object-oriented software, the entities can consist of variables, methods, objects, and object libraries (see Figure 2). The relationships can consist of inheritance, method uses, part-of relationships, and the higher level architecture including the formation of object libraries. This information can be represented as a kind of complex graph with both weighted and typed nodes and arcs. Nodes and arcs can be weighted by the size of the things they represent, or by other metrics. Entities are shown as the 3D boxes colored by type and can have 2D graphics drawn onto their surfaces. Relations are 3D lines or "spars" connecting the nodes. Each node can contain an entire sub-graph nested to an arbitrary depth. Nodes can be either open or closed (elided). When they are open, the contents are visible. When they are closed the size is reduced and the sub-graph is hidden. The size of a node is a function of the number and size of the nodes in the internal sub-graph. If two nodes contain sub-graphs that are connected when the nodes are closed, a "fat" arc appears that represents all the nested arcs. The thickness of the arc depends on the number of arcs it represents.

NV3D relies on fast, interactive 3D graphics, and runs well on a Sun ULTRA with Creator3D graphics, or an SGI O2, Indigo2 Extreme, or better.

Navigation and Exploration in NV3D

NV3D is designed around four techniques for navigation.

1) Elision
2) 3D widgets
3) Rapid zooming
4) Non spatial navigation

In this section we describe each of these techniques and how they work together. Early in the development of NV3D, we also explored the use of stereoscopic viewing and Fish Tank VR [4] viewing as navigational aids. We comment on our experiences and give the reasons why we find these viewing modes to be less valuable.

Elision

Information hiding through elision is critical to the practical application of NV3D. The system is only capable of real-time interaction while showing a few hundred nodes and arcs simultaneously. More than this and performance is reduced to the point where interaction is difficult. Thus, very large structures can be visualized only if they are nested. It is also essential that there is some principle that allows us to impose a tree (or forest) structure on the graph for this purpose. Fortunately, we have found that most of the examples we have contain at least one relationship that can be used to nest sub-graphs and thus simplify the 3D diagram.

A nesting structure is usually inherent in the way the software is organized into directories, libraries, modules, etc. For example, Figure 3 shows the structure of a six million line program that represents part of a digital switch that has been nested six levels deep in a hierarchy of modules and sub-modules. At the top level there are only four major entities, and as we move down the structure there are rarely more than fifty entities nested within any given node. This is one of the largest examples we have investigated, but we still find it to be quite manageable.

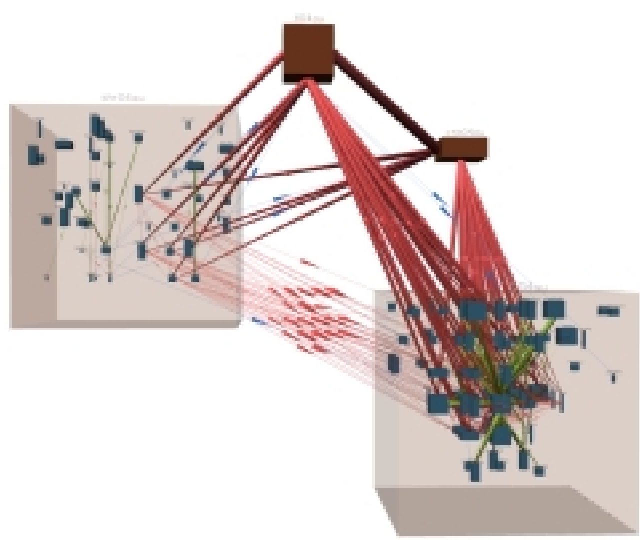

Figure 3. The structure of a six million line-of-code program undergoing major reorganization. The code is broken into four major sections, two of which have been opened in this figure. Some nodes within the sections contain nesting as deep as six levels. The arcs between the sections link duplicated modules from their previous locations to their new ones. They have been duplicated in an effort to clean up the system by placing highly connected entities into the same section.

3D Widgets

Basic spatial navigation in NV3D combines the use of 3D widgets with scaling. The 3D widgets are illustrated in Figure 4. These widgets are 3D objects rendered as part of the scene. For example, the rotation widget at the bottom of the screen curves into the scene at both of its extremes. The widgets allow the scene to be rotated, translated and scaled using either velocity control or position control. There are two translation widgets, two rotation widgets and one scale widget. The translation widgets are used to translate the graph in the X- and Y-directions, however, they are rarely used because most of the navigation is done through rotation and rapid scaling. The two rotation widgets allow for rotation about a vertical axis and for tilting about a horizontal axis. The scale widget scales the entire graph around the center of the screen. Most of the widgets have two "handles". The triangle handle is used for velocity control. When it is selected and dragged, the graph structure moves with a velocity that is proportional to the displacement from the neutral middle position. If the middle mouse button is held down at the same time, then the widget locks to its current velocity. This can be used to maintain a constant rate of rotation, for example. The rectangular handles allow for direct manipulation of the scene. When one of these handles is selected, the scene moves by an amount that is proportional to the displacement. Effectively, these handles provide a way to drag the graph into position.

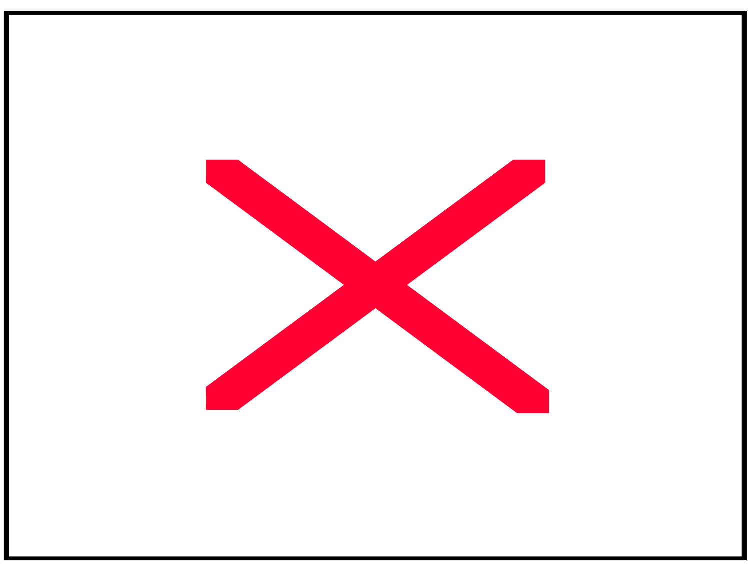

Figure 4. The 3D widgets used to navigate graphs. Each widget has several parts, and each creates one single type of motion in the graph.

Navigation using rapid scaling is achieved by a method similar to POI navigation but based on different principles. The user positions the cursor on an object and depresses the middle mouse button. This causes the scene to scale up as long as the button is held down. A double click followed by a prolonged hold allows the user to scale the scene down again. When the zoom-in operation is being applied, the entire graph structure scales up about the middle of the screen and the scene is simultaneously translated so that the selected object moves to the center of the screen.

The scale and translation algorithm is as follows:

```
while (middle mouse button depressed)
{
      scale (scene by 5% about the center of the screen)
      translate (scene 10% of the distance from the object to the
            center of the screen)
}
```

These parameters are such that (assuming a frame rate of 20 frames per second) the object will have moved 98.5% of the distance to the center of the screen after 2 seconds and the graph will have been scaled up by a factor of 7.

Rapid Zooming

The key to combining widget controlled interaction with rapid zooming is having a common center for both types of operations. In NV3D there is a point in the center of the 3D workspace (also at the center of the screen) that is used to integrate zooming and widget operations (see Figure 4). Once a scale has been made, the object that has been scaled has simultaneously been moved to the central point. Subsequently, the scene will rotate roughly about the center of that object (see Figure 5).

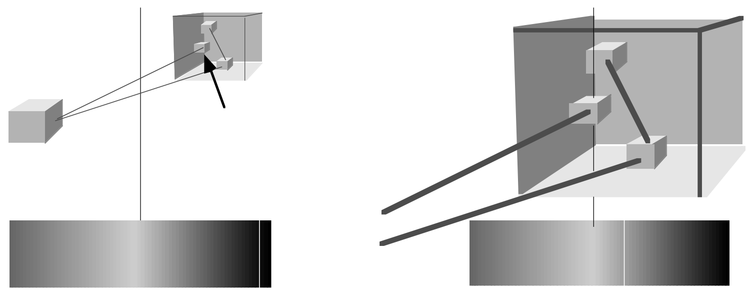

Figure 5. The technique of rapid zooming. The turntable in the center of the scene represents both the vertical axis of rotation and the center (in X and Z) of the scene. All scaling takes place about a point in the middle of this axis, and objects that are being zoomed are translated to this point.

When the user selects an object and holds down the middle mouse button, the scene is scaled about the workspace center and at the same time, the scene is translated so that the center point of the selected object is moved to the midpoint of the scene. The vertical axis of rotation for the 3D widget also passes through this point.

In order to zoom out, the user double clicks the middle mouse button and holds it down. While the button is depressed, the graph is scaled down about the scene's center point. However, in this case the scene does not translate, it simply gets smaller.

Non Spatial Navigation

NV3D also allows the exploration of the data in ways that are not spatial. The two methods available in the basic system are transitive queries and layout.

Queries

One of the key problems in software re-engineering is partitioning the existing legacy code in such a way that it becomes more modular. Part of this task involves finding the domain of influence of various entities such as variables or procedures. NV3D has a dynamic querying capability whereby any node can be selected and relationships can be explored by highlighting incoming or outgoing arcs of any designated type. For example, if we wish to find all the procedures that call a particular procedure, we first pick the node that represents the particular procedure and then we select the 'procedure call' relation. As we move a slider, arcs representing the calling relation and which are connected to the highlighted node are themselves highlighted. Moving the slider further highlights entities that call the designated procedure indirectly (two arcs away). As the slider is moved further, the relation is applied transitively revealing what is, in effect, a breadth first search of the graph for those procedures that might be affected when the selected node is changed in some way.

Layout

Graph layout is not normally thought of as a data exploration technique. But in NV3D it is a powerful tool for understanding information structures. The basic method is to weight a certain kind of relation heavily (for example inheritance in an OO program) before applying layout. When layout is performed, that particular structure will become more visually prominent.

The layout algorithms of NV3D are described in detail elsewhere [18]. Here we briefly summarize the layout methodology and then describe how it can be used as a data exploration tool. NV3D is designed to create a kind of synergy between manual layout and automatic layout. It is also this synergy that makes it a powerful tool for understanding complex structures. The automatic phase of NV3D layout is a two-stage process applied recursively throughout the nested structure. Note that each nested graph is divided into a series of vertical layers, which enforces a tree- or forest-like structure on the graph. For any nested sub-graph the following steps are followed:

1) Assign nodes to layers in the graph. This is done in the following sub-steps:
   a) Discover root nodes.
   b) Find all disjoint clusters in the graph based on these root nodes
   c) Assign the nodes to vertical layers depending on the distance from a root (cycles are broken as part of this process). The clusters are placed together, beginning at varying levels of the graph, so as to even out the number of nodes on any given layer of the graph.
2) Lay the graph out within a layer using a simulated annealing process. Nodes are dragged into position by the sum of the weighted arc forces, as well as forces repelling unconnected nodes.

There are two ways in which this layout algorithm can be tuned to represent different aspects of structure. First, the layering can be done according to any kind of arc. For example, if the structure represented is object-oriented software, then either inheritance or the 'uses' relationship ('function *uses* variable') could be used for layering. These relationships can be ordered, so that inheritance arcs are used primarily, then any nodes not connected by inheritance can be layered based on connected 'uses' arcs.

Secondly, in the annealing process, different kinds of relationships (arc types) can be weighted differently depending on their importance. To continue the above example, if inheritance is considered to be the most important relation, then heavily weighting inheritance will cause the nodes to cluster by inheritance and the inheritance structure will be more clearly revealed.

Figure 6 shows two layouts of a graph representing software code written in the Natural programming language. The layout on the left shows the effects of assigning a lower priority order to the arc type of 'fetch', which predominates the graph; while the graph is still understandable, the 'fetch' arcs wind up being more spread out, since clusters were primarily created with smaller arc groups. The other image shows what results from ordering the ‘fetch’ arcs first; the structure partitions are more cleaner, and the structure is more understandable.

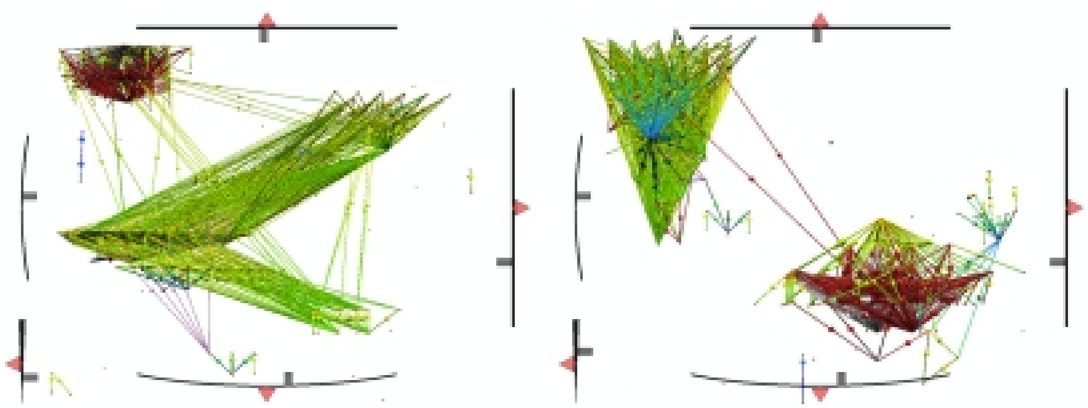

Figure 6. The effects of ordering arc types in layout. In this graph, the ‘fetch’ relation is the most numerous arc type. In the image on the left, the layout is done with ‘fetch’ relations given low priority, which results in it being spread across the graph. In the image on the right, these ‘fetch’ arcs are ordered with a higher priority than the other types, which results in a more coherent structure for the graph.

Fish Tank VR and Stereoscopic Viewing

In the early stages of this project we investigated the use of Fish Tank VR viewing modes in NV3D. This is a method for producing a convincing 3D representation of a virtual object by using stereoscopic viewing glasses and head position tracking.   The head position information is used so that the perspective is geometrically correct from the user’s actual viewpoint. When this is done carefully, a convincing 3D virtual object can be created in the vicinity of the monitor [4].

Although our studies have shown us that more complex graphs can be perceived when fish tank VR viewing is employed [17], we have abandoned this mode of viewing for two reasons:

1) The motion parallax that comes from head movement appears to be the most important component of Fish Tank VR (it allows for a graph 120% larger to be viewed). However, we can also get motion parallax simply by rotating the graph on a virtual turntable, and this has proven to be just as effective.

2) Stereoscopic viewing does allow larger graphs to be examined comprehensibly (about 60% larger). However, this gain is offset by the cost of only being able to display half the number of nodes and arcs at the same update rate and the same resolution. In addition, because of ghosting and other artifacts, the display is of lower quality. There is also the possibility that eye strain may result from extensive stereoscopic viewing.

Ultimately, the balance may change. If the cost of stereo and head tracking were negligible and the graphics fast enough so that all the required nodes and arcs could be displayed with a smooth 60Hz update of the display, then this would be a useful display option.

Imposing Behavior on Nested Structure

The major reason for imposing dynamic behavior on top of NV3D graph structure is to provide an outlet for viewing execution traces. Execution traces contain the calling sequences of the various threads in an application. Many systems currently exist for debugging multi-process software, however, none have attempted to debug software by animating execution traces within the code structure of a nested environment. Many of the complications that had to be overcome to make execution animation possible stem from the nested characteristics of NV3D.

Snakes

One of the major issues surrounding the implementation of dynamic behavior is how to visually represent an executing process. Previous systems, such as XPVM (used to visualize messages of a Parallel Virtual Machine [15]), used simple lines to represent both executing entities as well as the interaction among them. Irrespective of underlying software structure, lines may indicate process activity quite satisfactorily; however, a different approach must be taken when superimposing animation on top of structure. PVMTrace [8] uses 3D pentagons to represent the various nodes of a parallel machine. Messages are shown as small animations, resembling beads, which flow between nodes. Beads such as these could potentially be used to outline execution sequences; however, intuitively they make better message packets than execution threads.

We have developed a novel construct that we call a snake to visualize execution threads. Instead of highlighting an entire arc, a snake (shown in Figure 7) which has a variable number of head shapes and sizes, animates from one end of an arc to the other, leaving a tail stretching out behind it. The purpose of the tail, which can be many arcs in length, is not only to guide the user's attention, but also to present a recent historical account of where the process has been. This allows the user to piece together the recent events that have just unfolded. The various head shapes can be used to represent process states, types of processes, or even messages. For example, a cylindrical head can be used for calls and a cone shaped head can represent returns. Or, perhaps cones can represent processes while cylinders represent messages. Even though snakes were intended to represent processes, a snake without a tail can be made to represent a message. In this way, messages being exchanged between processes can be represented in addition to execution threads.

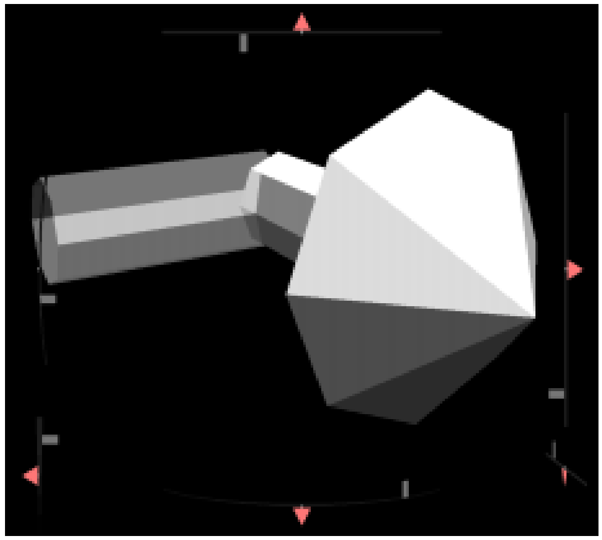

Figure 7. A snake, shown here with a cone shaped head, has the ability to bend around nodes to indicate execution calling sequences.

Multiple Actor Environment

With the introduction of snakes into the NV3D environment comes a significant increase in the complexity of graph interaction management. Consider the real world parallel of a night watchman roaming the barren hallways of an unoccupied office building. With a lone human user controlling an entire scene, interaction is quite simple. As the watchman makes his or her rounds, various doors are opened and light sources are turned on. After patrolling an area the doors are promptly shut and the lights are politely switched off. Whether it is an office building or a nested graph, this type of interaction by one actor in this type of environment is quite obvious and the rules are quite simple.

Snakes are implemented to take advantage of the very same facilities that the human operator manipulates via mouse interactions. Therefore, snakes can be considered automatic versions of the human user and consequently have transformed NV3D graphs into multi-actor interaction networks. Now, consider our office building analogy on a busy workday. People have many more factors to consider when interacting with their environment. For example, human courtesy implies that doors should not be closed in the faces of others, just as lights should not be switched off on a room full of unsuspecting people. Many of these same rules are implemented within the NV3D environment to metaphorically "keep the peace" between the numerous possible actors interacting within a graph. Figure 8 contrasts a single thread simulation against a multi-thread simulation.

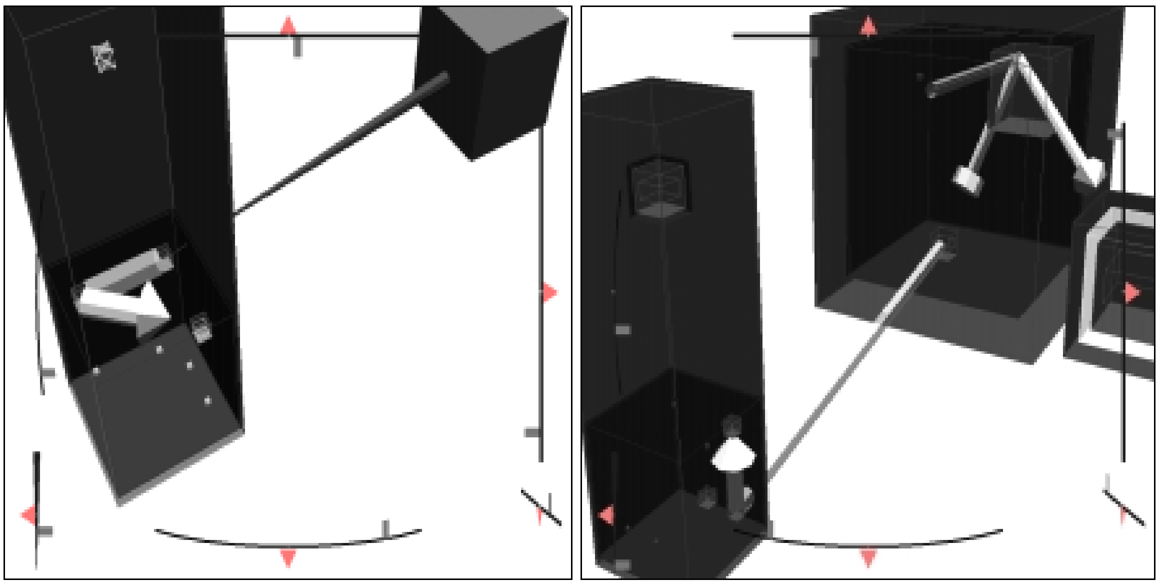

Figure 8. A single thread trace (first image) contrasted against a multi-threaded trace (second image). The second image shows how the amount of activity increases as multiple snakes are introduced into the same environment.

Consider the effect of one snake closing a node that is being used by another actor. If the second actor is the human user, this action may very well result in a certain degree of annoyance and frustration. The user may have intentionally arranged the graph so that certain nodes would remain visible. For example, the user may have opened all of the nodes representing files so that the internal classes are visible. The visibility of the classes may be an essential contextual clue that increases the overall comprehensibility of the trace. If a snake closes the pre-opened nodes, the setup is hopelessly destroyed and context is lost. The interruption of normal trace flow, caused by a trace abort or a simple stoppage, should result in the restoration of the initial graph-state.

If the second actor is another snake, then there is the potential for major disaster. Figure 9 illustrates the potential for such a situation. If the snake to the left closes the large node to the right when its tail reaches the end of the node, the snake inside the node will be trapped. The arcs, on top of which it is trying to animate, will be collapsed. Trying to travel on collapsed arcs would almost necessarily have dire consequences on system stability. To cope with this situation, the snake would be forced to reopen the closed or closing node, which would result in a very disturbing visual hiccup. There is also the possibility that a snake may close a node on top of another snake that has just come alive at the current time index, or that has been patiently awaiting a particular instance in time to continue to its destination. Therefore, snakes not only have to consider the active actors in the scene, but also any actors about to enter the graph or about to awaken.

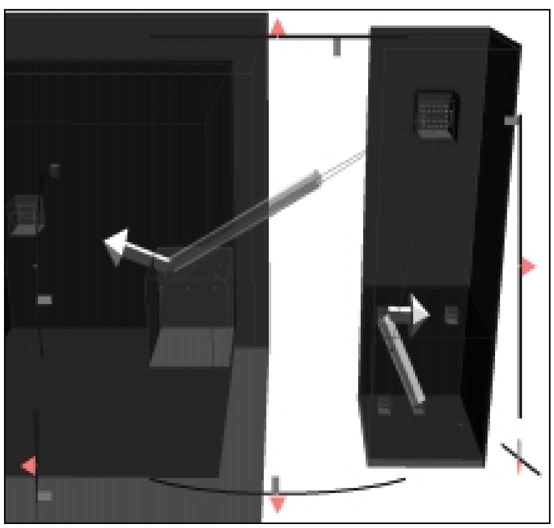

Figure 9. Trapping a snake inside a node. If the snake to the left of the image closes the node to the right when its tail has completely left the adjoining arc, the snake to the right may be trapped inside.

To address the problem of miscommunication between actors resulting in the closure of nodes on one another, each node in the scene is equipped with two actor counters. One counter handles snakes and the other handles the human user. When an actor requests the opening of a node, one of the counters is incremented. Conversely, when the actor requests that the node be closed, the same counter is decremented. The counters act analogously to a semaphore that is used to provide mutual exclusion between competing processes in a multi-process environment. As long as a node's actor counter is greater than zero, the node cannot be closed. As soon as the last actor inside of the node decrements the counter, it becomes zero and close requests are once again granted. This guarantees no actor will be adversely affected by closing nodes.

Human users require special attention. Their requests cannot influence the same counter as the snakes. If they did, then the counter could be incremented or decremented numerous times by the user. Consider a situation where three snakes have entered a node and each has incremented the actor counter. If the human user requests the closure of this node three consecutive times, the node would actually comply. The probability of this situation arising is quite high because it is a bit unnatural to request closure and not have the system obey (the user would probably keep trying to close the node). To address this issue, a second counter is placed at each node. This counter simply acts as a flag to indicate whether or not there is a user examining the contents of the node in question. The node can only close when both the snake counter and the user counter are zero.

Automatic Viewpoint Control

When very large data sets are visualized, or alternatively when the user has scaled up the display, it is possible that a significant portion of an NV3D graph may be occluded by the border of the viewing window. One very undesirable side effect of this inevitability is that animation located within the clipped regions of the graph will almost certainly go unnoticed by the user. The user may realize that activity has commenced within the graph but be completely oblivious as to its whereabouts. To remedy this problem, automatic camera controls, also called automatic viewpoint controls, were implemented. To realize this type of control, the user's viewpoint can be latched to the head of the snake enabling it to direct the user's attention to various locations in the graph.

Essentially the user's viewpoint is now performing an execution trace in synchronism with the snake.

By using the rapid zooming interface developed for navigation in NV3D, the user's viewpoint can be automatically manipulated. However, automatic viewpoint control also must deal with destinations not currently visible in the scene. This problem was first addressed in NV3D when querying support was added to locate nodes that were not easily locatable. First, the scene is zoomed out until all four front facing corners of the destination are visible. Then when both the departure and destination are in view, the scene is zoomed in to the destination node. If it is currently closed, the snake will open the destination node while the scene is being zoomed in.

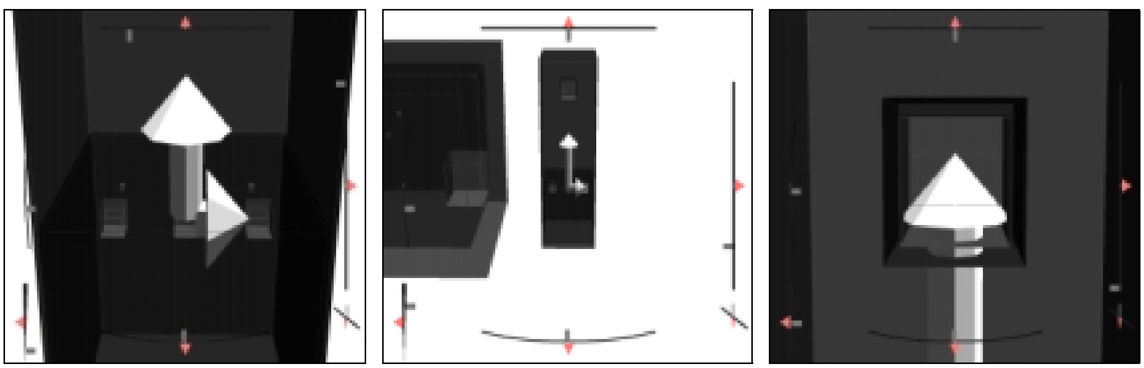

Figure 10. The technique of automatic camera control. The first image shows the user's viewpoint at the departure node. The next image shows the scene zoomed out so that the user can see both the departure and destination nodes. The final image shows the scene zoomed in to the destination node.

The only thing that the automatic controls need to know is the snake's next destination. This method provides the user with global graph context in addition to focus. Figure 10 shows a sequence of camera motions that occur to track a snake from a departure node to a destination node. The first frame shows the departure node. The second frame illustrates the position of the scene after it has been scaled down and the destination is visible. The last frame shows the final position of the scene with the destination at the origin.

Representing State Information

In order to use this implementation as a debugging aid, information must be provided to complement the actual calling sequence. For example, the user may wish to view the calling parameters with which a snake has reached a destination. Or perhaps the user wishes to see the process identification numbers of the sending and receiving nodes in an exchange of PVM packets. Ideally the implementation should provide a generic way to view any type of extra information retrieved through the data extraction process.

One of the very first attempts to provide extra information about the state of an execution thread was through sound. The snake, depending on the state of the execution thread it represented, would emit a different sound. For example, snakes performing calling sequences might emit a buzzing noise, while snakes performing returns might make a humming noise. Unfortunately, sound was eventually abandoned because it was quite annoying. The sound was bearable when one process was active in the graph, but the combined racket of numerous snakes was quite disturbing and confusing.

In order to display execution trace information in NV3D, resizable data probes were incorporated into the system. Snakes contain optional information attached to the nodes in their destination node lists. When a snake arrives at a destination, it informs the node that it has a message for any attached data probes. If there is a probe, the node informs it of the information that the snake wishes to display. The node also passes the data up through the nesting levels to any probes further up in the hierarchy. This allows the user to monitor all of the information being passed around a sub-hierarchy by attaching a probe to its parent. Figure 11 shows a data probe that has just received the calling parameters of the snake seen passing through the node.

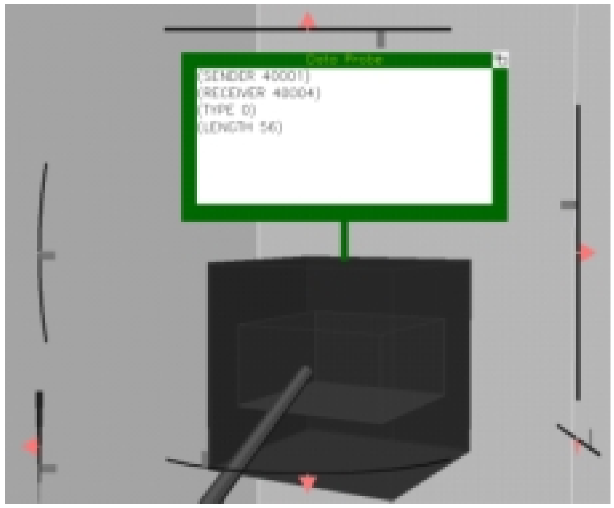

Figure 11. This data probe is attached to a node that has just been passed a message. The probe is displaying the data contained within that message.

Conclusion

We have reviewed a number of techniques available for dealing with the problem of navigating large amounts of data while still maintaining focus and context. In 3D, some techniques make more sense than others, and some additional methods are available. In NV3D we chose not to use any of the distortion methods, such as intelligent zoom, in favor of the added perspective of 3D combined with rapid navigation. This allows us to maintain more scene consistency and to use a single window interface.

To deal with the size of the graphs in NV3D, much of the information is elided. This requires significant scaling up and down of the scene in order to view the data. The main navigation technique that we employ involves rapid zooming, both manual and automatic. All zooming operates through a standard interface provided by a set of 3D widgets. This is important to allow for consistent movement; it is easy to see how automatic zooming affects the scene, for example, since the widgets reflect translation and scale changes. When snakes travel through the graph and guide the control of the camera, it is important to realize how these position changes relate to the graph as a whole.

An inherent understanding of the 3D graphs is essential to their navigation, and to facilitate this we have extensive layout capabilities. These capabilities are complemented by providing flexible control over the appearance of objects in the display. With a varied and meaningful graphical display, the graph becomes more like a distinctive information landscape that becomes familiar to the user over time.

With all the enhanced navigation techniques employed to make the data more comprehensible, we envision a number of practical software engineering applications for NV3D. These include reverse engineering, software partitioning, training and debugging. Currently our major effort is focussed on preparing NV3D for commercial software reverse engineering, particularly year 2000 remediation. A major problem in this area is partitioning legacy code so that subsections can be identified and isolated for remedial action.

We are building an interface to a software repository in partnership with Formal Systems Inc. (FSI). Code structure is reverse-engineered by means of FSI's Refine-based parsing system. As part of this process, it is placed in an Oracle database. NV3D provides a user interface to the very large structures that result from performing various queries on the database. By applying clustering and layout techniques, we can partition the code into structured hierarchies and segregated partitions. In addition, our interactive querying capability makes it possible to trace the sphere of influence of individual variables through the code structure.

In a separate project that is being carried out in collaboration with Nortel we are looking at three distinct applications.

1) Software Training. It can take an individual programmer six months or more to get an approximate mental picture of a large body of software. Because as many as 200 programmers a year may be brought into such projects, the training costs are

substantial. NV3D can be used to help new programmers understand the overall architecture of the code and the scope of their work.

2) Code management. We are working on tools that allow version control information to be superimposed on software structure. This can be especially useful at critical stages, for example, when code is undergoing a major re-organization, or when two distinct bodies of code are being merged. In addition, we are using clone identification software to identify duplicated sections of code. This will allow software engineers to organize software into a more rational structure.

3) Execution trace visualization. Nortel has been interested in using snakes to demonstrate dynamic program behavior to employees. Detailed traces of software execution on a digital switch could aid developers by allowing them to visualize complex interactions between low-level structures without having to examine source code.

Acknowledgments

We would like to acknowledge the assistance of Laval Letoureau, David Drynan and Tim Dudley of Nortel for providing data, advice and support. Authur Ryman of IBM Toronto Labs provided inspiration in the early stages of the project. We are also grateful to Ronald Hunter-Duvar for his help and insights into applications in software reverse engineering.

References

1. Bartram, L, Ovans, R., Dill, J., Dyck, M., Ho, A., and Havens, W.S. (1994) Contextual Assistance in User Interfaces to Complex, Time Critical Systems: The Intelligent Zoom. Proc. Graphics Interfaces'94, 216-224.

2. Bederson, B. and Hollan, J. (1994) Pad++: A Zooming Graphical Interface for Exploring Alternative Interface Physics, ACM UIST Proceedings, ACM Press,17-26

3. Card, S., Robertson, G.G. and Mackinlay, J. (1991) The Information Visualizer, an Information Workspace. CHI 91 proceedings, 181-188.

4. Deering, M., (1992) High resolution virtual reality. Proceedings of SIGGRAPH'92. In Computer Graphics, 26, 2 195-202.

5. Dill, J., Bartram, L., Ho, A., and Henigman, F., (1994) A Continuously Variable Zoom for Navigating Large Hierarchical Networks. IEEE Conference on Systems, Man and Cybernetics, San Antonio, 386-390.

6. Fairchild, K.M., Poltrock, S.E. and Furnas, G.W. (1988) SemNet: Three-Dimensional Graphic Representations of Large Knowledge Bases. In Cognitive Science and Its Applications for Human-Computer Interaction. Ed Raymond Guindon Lawrence Erlbaum. 201-233.

7. Furnas, G.W. (1986) "Generalized Fisheye Views", Proceedings of CHI'86 ACM Press, 203-209.

8. Gobrecht, C. Bhavsar, V. and Ware, C. (1996) PVMTrace: A 3D distributed program visualizer. 10th International Conference on High Performance Computers. HPCS'96 CDROM Proceedings Carleton University Press and IEEE Canada Press. Ottawa, (June).

9. Koike, H. (1993) The role of another spatial dimension in software visualization, ACM Transactions on Information Systems, 11(3), 267-286.

10. Lamping, J., Rao, R. and Pirolli, P. (1995) A focus and Context technique based on hyperbolic geometrics for visualizing large hierarchies. ACM CHI'95, Denver, 401-408.

11. Mackinlay, J.D., Card, S.K and Robertson, G.G. (1990) Rapid Controlled Movement through Virtual 3D Workspace. SIGGRAPH '90 Conference Proceedings. Computer Graphics, 24, 4, 171-176.

12. Noik, E. (1994) A Space of Presentation Emphasis Techniques for Visualizing Graphs. Proc. Graphics Interfaces'94, 225-233.

13. Sarkar, M and Brown, M.H. (1994) Graphical fisheye views of graphs, Proc Graphics Interface'94, 83-91.

14. Schaffer, D., Zuo, Z., Greenberg, S., Bartram, L., Dill, L., Dubs, S., and Roseman, M., (1996) Navigating hierarchically clustered networks through fisheye and full zoom methods. ACM Transactions on CHI 3(2), 162 -188.

15. Sunderam, V.S., Geist, G.A., Dongarra, J. and Manchek, R., (1994). The PVM concurrent computing: Evolution, experiences, and trends. Parallel Computing 20, 531-545.

16. Ware, C. and Lewis, M. (1995) The Drag Mag Image Magnifier. ACM CHI'95 Conference Companion, 407-408. Also CHI'95 videotape

17. Ware, C. and Franck, G. (1996) Evaluating Stereo and Motion Cues for Visualizing Information Nets in Three Dimensions. ACM Transactions on Graphics. 15(2) 121-139.

18. Ware, C., Franck, G., Parkhi, M. Dudley, T. (1997) Layout for visualizing large software structures in 3D. Visual'97 Second International Conference on Visual Information Systems, San Diego, (Nov) 215-225.