

BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-92-M16

“A Three Dimensional Browser for Visualizing Orthogonal Hierarchies“

(Using Only Two-and-a-Half Dimensions)

by
James Wen

**A Three Dimensional Browser
For Visualizing Orthogonal Hierarchies
(Using Only Two-and-a-Half Dimensions)**

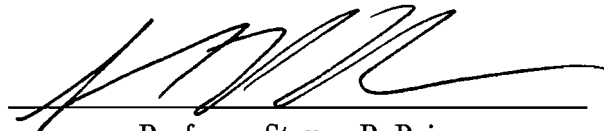
James Wen

Department of Computer Science
Brown University

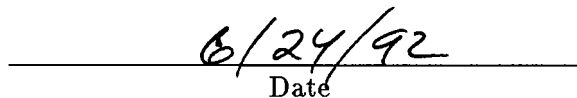
Submitted in partial fulfillment of the requirements for the
Degree of Master of Science in the Department of Computer Science
at Brown University

June 1992

This research project by James Wen is accepted in its present form
by the Department of Computer Science at Brown University
in partial fulfillment of the requirements for the Degree of Master of Science.

A handwritten signature in black ink, consisting of a series of loops and strokes, positioned above a horizontal line.

Professor Steven P. Reiss
Advisor

A handwritten date "6/24/92" in black ink, positioned above a horizontal line.

Date

This paper is accompanied by a 10-minute video which demonstrates
the applications described.

Abstract

Using three dimensional graphics to visualize abstract data presents an interesting challenge because, by definition, there is no *physical* counterpart to abstract data; its extension into three-space is not due to the fact that it is the *natural* thing to do. Prior work in using three dimensional graphics to aid abstract data visualization has concentrated on increasing the quantity of data visualized, using the third dimension as a natural but symmetrical extension to the world of two dimensional layouts. Little has been done, however, to actually use it to capture the semantics of abstract data. This paper presents the notion of using the third dimension asymmetrically so that our familiarity with two dimensional layouts and our abilities to mentally group things confined to a plane are exploited, leaving the third dimension as a means to encode semantic information. A design principle is forwarded whereby the notion of orthogonality in an analytic sense (independence) is captured by using orthogonality in a geometric—and hence, visual—sense. Two systems that illustrate the design principle applied to two very useful areas are described: a visualizer for a set of data with multiple relationships defined over it and a visualizer for program structure and control flow in an object-oriented environment.

1 Introduction

With the ever increasing potential of computer hardware and specialized processors, technologies once considered too expensive computationally to be used as tools for interactive visualizations are becoming available for more general uses. One such technology is interactive three dimensional graphics. Even though it is more readily accessible now, its inclusion into the design stage of various visualization tools still needs to be more fully integrated. While some visualizations, like that of gaseous phenomena, say, almost require three dimensional graphics, others have no natural—with *natural* in this instance meaning *physical*—counterpart. The visualization of program structure, for example, is based upon the need to see things that are fundamentally abstract. There is no *natural* way they should exist. Because most of these things were proposed through books and papers and because people are more apt to write than to sculpt, we have become accustomed to visualizing such things as two dimensional entities.

This familiarity with two dimensional visualizations should not be disregarded as irrelevant, however, now that we can use three dimensional graphics. Instead, it can be exploited to allow the visualization of a semantically richer set of data. Rather than simply using the new dimension as extra room to stuff more of the same information, we can constrain information of one nature to lie in one plane and allow information of another nature that may share common data to break the plane. Thus, instead of destroying an approach to visualization we are already familiar with in order to use the third dimension as a new but symmetrical dimension, we can reuse the methodologies of two dimensional visualization saving the third dimension as something that depicts new semantic information.

1.1 Organization of the Paper

After a review of prior work in the area in section 2, section 3 presents the principle that underlies this paper along with specific ideas that embody the principle. Section 4 describes two systems implemented based upon the ideas outlined demonstrating the validity and applicability of the ideas in two very different areas: a three dimensional graph browser with multiple relationships defined, and a three dimensional program flow visualizer for object-oriented languages. Section 5 comments on future directions and possible

future work on the project, and section 6 offers some concluding remarks. A user's reference guide for using the applications implemented is found in appendix A. Appendix B examines and explores the possibilities of other orthogonal channels of information. Comments regarding the implementation and architecture of the system are given in appendix C. A video accompanies this paper and demonstrates the applications in use.

2 Prior Work

Work in the area of three dimensional visualization has been fairly extensive in the domains of natural phenomena and physically based modeling. However, using three dimensional graphics in the context of visualizing purely abstract data has not been as extensive. This is due primarily to the lack of a natural three dimensional counterpart for such things. There have been successful attempts at *creating* natural settings for abstract data: the two dimensional desk top metaphor becoming a three dimensional office metaphor is one such example [7, 10]. Though the virtues of using three dimensional graphics for aiding in visualization of non-physical information has long been realized, even before the advent of computers [13], the study of actually exploiting interactive three dimensional graphics for visualizing pure abstract data is still very limited.

Three dimensional graphics offers two areas from which techniques can be extracted: those arising from image space (projective geometry) and those arising from object space (three dimensional modeling). As for the former, techniques of projections that are based upon fisheye lenses have been investigated as ways of focusing on various interesting parts of a given data set [4, 12]. But, because they only work in image space, the data itself cannot be manipulated as a three dimensional object that allows the user to gain an intuition for the relationships as they exist spatially.

The manifestation of abstract data in three-space as actual objects are explored in the cone tree [9] and the perspective wall [11]. Both systems are capable of capturing a tremendous amount of information and conveying it through static images. But both systems are dependent on the user's focused attention because they both rely upon the actual movement of the data to convey the origin of the information viewed. As well, both systems use the third dimension simply as a means to increase the quantity of data visualized.

The nature of the data is not addressed as an issue. The third dimension is just a symmetrical generalization of its lower dimensional counterpart, the two dimensional paradigm, and little has been done to allow the existence of a more heterogeneous set of data relationships in a three dimensional environment. So far, only when a very different channel of information is explored does orthogonal information get represented. The use of audio capabilities is one such example, and appendix B briefly examines that in the context of this paper.

3 The Principle

Central to the principle of this paper is the argument that, when moving into the third dimension, keeping around some of the more traditional two dimensional methodologies for visualizations may have its rewards. The familiarity we already have with two dimensional layouts is advantageous to such an approach. But it is not for the sake of such psychological downwards compatibility that the idea is forwarded. Rather, it is the fact that the familiarity with things two dimensional makes it conducive to mentally group all things constrained on a plane to share something in common, e.g. they are related to each other under one relationship. The ease with which things can be grouped into macro structures becomes more important as the complexity or volume of data increases. Just as it is fairly easy for us to accept that things in a two dimensional plane all conform to some particular constraint, it is also easy for us to accept *without having to focus our attention* that things off the plane may be semantically different. The less we have to focus our attention, the freer we are to concentrate on the actual semantics of the situation.

The principle, then, is to attempt to *extend* rather than to replace two dimensional methodologies when exploring visualization techniques in three dimensions. An effort should be made, in a system following this principle, to maintain as much of the two dimensionality of the original structure as possible. The reuse of two dimensional techniques in three space imposes an asymmetry that could be used to encode information whereby things that break the symmetry are seen to break it for a reason and so are easily accepted as being semantically different.

3.1 A Comparison of Browsers

The Cone Tree

An example of a generalization of traditional two dimensional visualization techniques into three dimensions that does *not* observe this principle is found in the *cone tree* [9]. The cone tree is a clever way of encapsulating the triangular two dimensional tree structure into a conical three dimensional tree structure. Navigating through the tree is accomplished by rotating the various cones until the node of interest is visible. In this system, the actual locale of a particular node has no semantic meaning, much like its two dimensional counterpart. In fact, the system is essentially a two dimensional triangular tree topologically warped into a three dimensional cone tree. All three dimensions are treated symmetrically. So while such a system is ideal for visualizing a tremendous quantity of information, the third dimension, in this case, is used simply to extend the storage capacity of the same data structure as its two dimensional counterpart—nothing of a different data type is allowed.

The Orthogonal Graph Browser

This paper presents a three dimensional browser that, when started, appears and acts like a typical two dimensional graph browser with all its functionalities. The browser has a three dimensional structure but, unlike the cone tree, the user never has to know that. Additional information—beyond what is possible with a typical two dimensional browser—is gained when the user changes the viewpoint. Various additional information, stored in the direction parallel to the line of sight—or orthogonal to the viewing plane—reveal themselves as the viewpoint is changed. The orthogonal graph will be examined in more detail in the later sections.

3.2 The Principle Applied

By reusing two dimensional technologies, part of the task of this study becomes recognizing notions that are already well encoded in the two dimensional plane as well as those that may require an aid in visualization. The problem is then transformed into finding clever ways to exploit the third dimension in order to convey and capture those things that require more than

what is available in two dimensions. The following are three examples of approaches based upon the given principle. They are by no means exhaustive but they do represent ways of using an orthogonal channel of information for extending the traditional planar techniques of visualization.

3.2.1 Orthogonal Planes

Consider a binary relation, R_1 , defined over a data set S_1 where $R_1 = \{ \langle s, t \rangle \mid s, t \in S_1 \}$. A traditional two dimensional layout may suffice in visualizing such a structure where S_1 becomes the set of nodes and R_1 becomes the set of edges, say. Now consider another relation, R_2 , defined over the set $S_1 \cup S_2$. The sharing of data makes it useful to view the two relations simultaneously but putting everything on one two dimensional graph can become too dense and confusing.

To illustrate this point, we turn to the world of abstract algebra which provides a natural setting for the point to be made. First, we define a notion of something we shall call *category*. A category is essentially a list of properties. All things that belong to a particular category must have all the properties of the category. A *Ring*, for instance, is a category that requires its members to have the properties of closure under addition, closure under multiplication, and the notion of an additive inverse. Members of a *Ring*—or things that satisfy those properties—include *integers* and *matrices*. A *Field* is a category that requires its members to have all the properties of a *Ring* with the added requirement that there exists a multiplicative inverse. Notice that the *Field* category inherits its properties from the *Ring* category. The set of algebraic categories defines a hierarchy of such inheritance with each child requiring more properties to satisfy for its members [2]. The hierarchy that results is a directed acyclic graph. Figure 1 shows a small example.

The members of categories—those things that have all the required properties of the categories—we shall call *domains* and they correspond to the familiar domains of computation, e.g. *integers*, *reals*, etc. Domains themselves define hierarchies, but in a different manner. *Positive Integers*, *Even Integers* and *Integers mod p* are all *Integers*. See Figure 2.

What we may typically like to do is to scan the graph of categories until we find one we want to inspect and then to examine the domains that belong to that category. As can be seen in Figure 3, this would not be convenient using the traditional two dimensional layout. In an example with just slightly

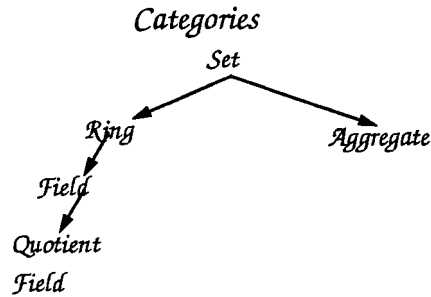


Figure 1: Category hierarchy

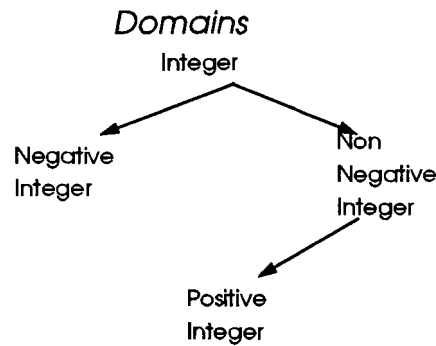


Figure 2: Domain hierarchy

more data, such a mixing of heterogeneous relationships on a planar graph will be very confusing and difficult to read.

There are, in this example, two sets of relationships that overlap in data. We would like to somehow see them together because of the shared data but we do not want to confuse the two relationships. Figure 4 shows how we can, in three space, come up with a fairly compact and elegant way of visualizing both simultaneously. Information orthogonal to one abstraction are placed literally in an orthogonal plane. Illustrations of the concept in an actual application are given in section 4.1.1.

Categories & Domains

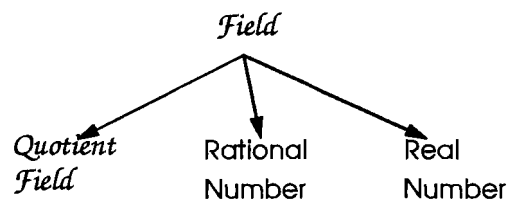


Figure 3: Category/domain relationship

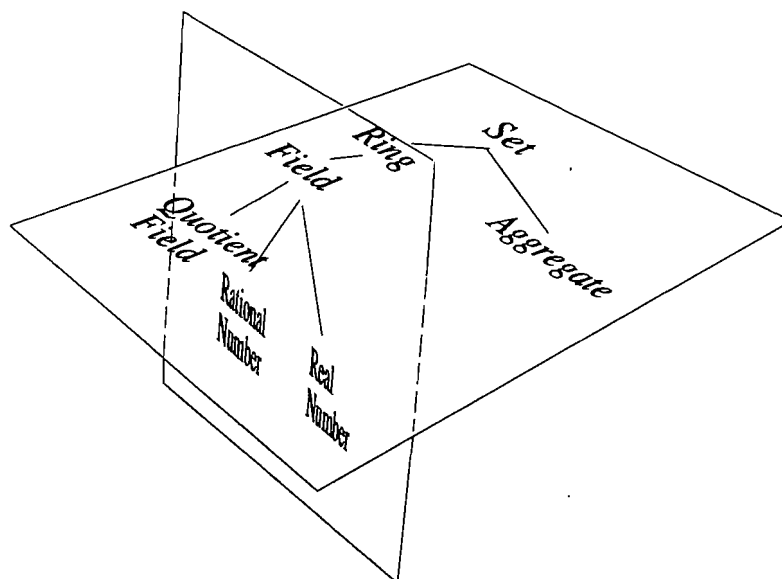


Figure 4: Using orthogonal planes

3.2.2 Depth Cues

Systems that provide a visualization of the flow of control in a program can run into a problem with recursive functions. Consider two mutually recursive functions, A and B. Figure 5 illustrates how simply visualizing the flow of control does not reveal whether the flow is descending a recursion or returning from one. The figure shows a situation where A invokes B which invokes A

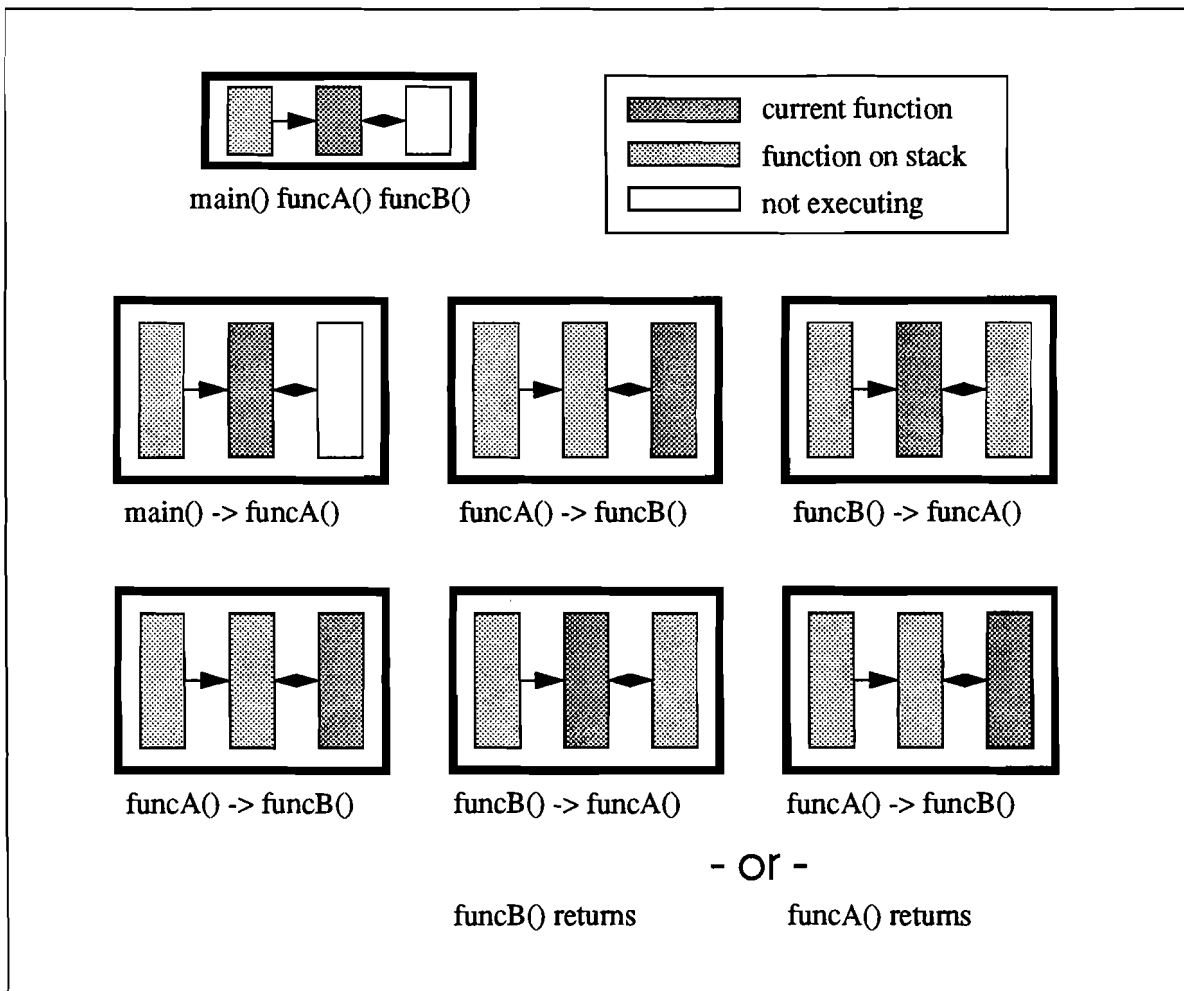


Figure 5: Recursive control flow

which once again invokes B. If, in the next step, the flow of control goes to A, it may be difficult to tell if B returned to A or whether B is invoking

A. Having the layout fixed in two dimensions makes it hard to encode any information in the nodes themselves except with colors. As seen in the figure, colors do not suffice. What is needed is some indication of whether a routine is being pushed onto the run stack or it's being popped off the run stack.

Such information can be hidden in the third dimension. A straight on view of the system shows a two dimensional graph. But we can encode the number of times a node is invoked by changing the thickness, say, of a node. Again, the two dimensional appearance remains undisturbed but, if the graph is rotated, one can get an instant profile of each node. The thickness of the node can encode any scalar function of the node and, in a programming environment, encoding the number of invocations is a useful notion. An instantaneous indication of a program descending recursions or returning from recursions can be had by simply rotating the graph so that the thickness of the nodes could be seen, as show in Figure 6.

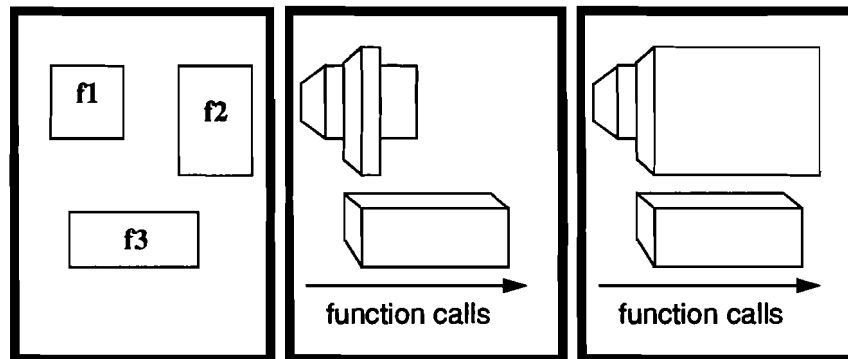


Figure 6: Encoding function calls in node thickness

3.2.3 Grouping by Height

In a browsing environment, we typically work our way from one node to another via the edges. In a graph with many edges, however, extracting even simple parent-child relationships could be somewhat difficult. Extracting all the grandparents/grandchildren is even more difficult. We would like to group nodes related to each other visually but this may be very difficult to do in a complicated graph already laid out in a plane.

Here, we examine how we can use the third dimension to group nodes together. If we define the initial viewer's position as being a some point $(0, 0, z)$ with the initial two dimensional layout constrained to the XY-plane, information can be hidden in the node's position along the Z direction. Nodes of interest could be raised or lowered slightly so that a straight on view is not altered noticeably but a rotated view yields a picture of the grouping. Or, nodes can be raised to various exaggerated heights so that a direct view would, with perspective, make the more interesting and higher nodes closer and hence, bigger, and the less interesting nodes farther away and hence, less prominent. That is, we can *pinch* the node of interest and have its neighbors rise up with it, but to a lesser extent, much like the skin which grows on old milk that's been sitting in the refrigerator for too long. (See Figure 7). This concept will be further explored in section 4.1.2 of the application section.

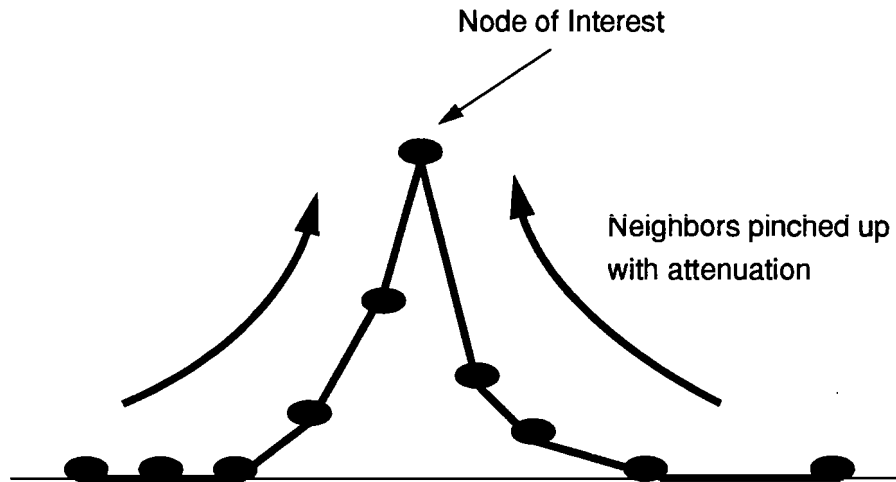


Figure 7: Pinching a node and its neighbors

3.3 The Importance of Interactiveness

It should be pointed out that it is very crucial for such a visualizer to be interactive. By the stated principle, it follows that the three dimensional-

ity of the structure can only become apparent after the user has interacted with it. Otherwise, it appears and functions just like a traditional two dimensional browser. Furthermore, by allowing the user to interact with the system—rotating and moving the three dimensional object around—spatial orientation could be gained. This orientation is not found in two dimensional systems but is part of our innate intuition developed from our natural spatial surroundings. We develop, at a very early age, the ability to perceive the existence of objects even if they are not visible or sensed otherwise [6]. This perception of *object permanence* is based upon the mental facility to account for the object's present state based upon its history and environment. Thus, if we move a structure so that part of it is behind us, we do not need a visual cue to actually *visualize* the part of the structure we cannot see. In the cone tree, for example, it is possible to see an entire file system in one entity—albeit most of the information is not actually visible most of the time as they are rotated out of view. The invisibility is not such a problem, however, due to our ability to perceive existence spatially for things even if we do not actually see them. This notion is critical in designing three dimensional interfaces because it frees up the boundaries imposed by a viewport window: we can mentally account for things we cannot visually account for much easier in the natural setting of three-space than in the more artificial plane of two-space.

4 The Applications

This section presents two applications that demonstrate how the above forwarded principle can be used in visualizing data and program structure. It is in this section that the ideas of section 3 are actually tested for their usefulness as well as usability on a user-interface level. The reader is referred to the accompanying video for a demonstration of the applications. Both applications are implemented on top of a common base system. A description of the available functionalities is given in appendix A. Comments about the base system and its implementation can be found in appendix C.

4.1 A 3D Browser for Multiple Relationships

The browser implemented for this project acts, upon first view, much like a traditional two dimensional browser with the functionalities of the typical planar browser. That, of course, is the intent. It is only after the displayed graph has been rotated does its existence as a three dimensional browsing facility with additional functionalities become revealed. Here we examine some of the additional features it provides as a three dimensional browser.

4.1.1 Use of Orthogonal Planes

If the user clicks on the middle mouse button with the pointer positioned over a node, the subgraph rooted there is expanded if it is hidden or collapsed if it is visible. In this way, the browser acts like a typical two dimensional system. If the node has an asterisk at the end of its name, then there is a collapsed orthogonal graph rooted there. Clicking on the right mouse button expands it. The user can then rotate the graph to a position where it is easy to view how the two relationships relate to each other through the node they share in common. This is illustrated in Plate *I*.

Having the orthogonal subgraph actually be exactly 90 degrees from the plane of the main graph may make a simultaneous view of both somewhat difficult since viewing one plane directly causes the other plane to be edge on. The planes are thus made rotatable about an axis lying along the breadth of the graph and passing through the node that roots the orthogonal subgraph. The user can change the angle the orthogonal plane actually makes with the main plane so that viewing both relationships is made easier. Additionally, the hue of the orthogonal plane is a function of the angle it makes with the main plane so that, when the orthogonal plane is rotated into the main plane, it takes on the color of the nodes on the main plane (this may be modified by the user). See Plate *II*.

4.1.2 Use of Grouping By Height

Traversing a graph entails visiting a node and moving on to its neighbors—either its parents or its children. As mentioned earlier, with a dense graph, this could be somewhat tedious. Two means of grouping by height are given.

In one interface, the parents and the children of the selected node(s) can be raised or lowered by pressing the appropriate keys on the keyboard.

The other interface addresses the fact that, when studying in a graph, we may often find ourselves focusing on a node of interest anchoring our interest relative to that node and then expanding our interest away from it. An interface that addresses that mode of thinking is one where the selected node can be raised dragging its neighbors with it. The further away a neighbor is to the node of interest, the less it is dragged. In this way, we can instantly create a “mountain of locality” where the node of interest is on top and things further and further away are placed further and further down the sides of the mountain. This strategy of focusing on a particular node of interest and having its neighbors recede in prominence creates an effect not unlike that of the work done in information visualization with fisheye perspectives [4, 12]. However, it should be pointed out that this approach is, in fact, very different. The cited papers proposed altering the projected image so as to skew the perspective bringing some nodes forward and making other nodes recede. Such an approach does not allow the user to alter the viewpoint to afford another view of the data since the data does not truly exist in three-space but is only distorted with three dimensional techniques. The ability to interact with the data object as a three dimensional entity, however, provides the user with an intuition of the data and its structure not possible by more passive means. The system presented here creates the focusing effect by actually moving the nodes in object space. The user is then free to rotate the object freely in space to gain a better grasp of the data. This is illustrated in the Plates *III* and *IV*.

4.1.3 Use of User Interactions

A layout program can never hope to always choose the perfect layout for a user simply because the user may be tracking down some very specific thing and sees the data in a way, semantically, that is not inherent in the data itself. It is thus very important to allow the user to group things that are meaningful for whatever particular situation may exist at the time. Such a grouping should be something that could be saved for later retrieval for a given graph.

Nodes can be selected and moved to create a more meaningful layout for the particular session. Selection of nodes can be done on an individual basis or into a group. Subsequent operations (e.g. moving the node) act on all the selected nodes. Selections can also be done on a generation basis: pressing

the key *c* will cause all the children of all the selected nodes to be selected. Parents can be selected by pressing the key *p*.

This system allows interactive manipulation of the graph. Nodes can be selected individually or in groups and then moved with the mouse. The move can affect only the selected nodes or the selected nodes as well as their subgraphs.

Allowing such re-organization of the default layout gives the user the ability to customize the visualization of abstract entities that, in themselves, are not able to capture all the information the user may associate with them.

4.2 A 3D Object-Oriented Program Flow Visualizer

The system described in this paper lends itself well to be used as a visualizer for object-oriented languages. The reason for this is as follows: object-oriented languages inherently embody multiple relationships over overlapping data. Abstract classes in C++, for instance, form a hierarchy describable by a two dimensional graph. Concrete classes derived from abstract classes have the exact relationship as domains have to categories, as described above. Each concrete class can form a hierarchy on its own but may also have its root in an abstract class. Furthermore, methods of these classes form yet another relationship: they are related to their parent classes as well as to each other (if viewing by caller/callee relationships). The application presented here shows the class hierarchy on one plane and the methods of the classes on orthogonal planes.

4.2.1 Current Visualization Tools

Flowview and *cbrowse* are two visualization systems that exist in a general program and data visualization environment *FIELD* [8]. *Flowview* allows the user to visualize the flow of control in a program. In the object oriented environment of C++, a call graph is presented with the nodes representing the methods and the edges representing the caller/callee relationship. What is also desirable is to be able to see the relationship of the parent classes. *Cbrowse* provides this service but it does not show the flow of control. The two systems share some data but are defined over different relationships. Taken together they make a good candidate for being captured by an orthogonal graph.

4.2.2 Using The Orthogonal Browser

The system proposed in this paper is ideally suited for visualizing the program structure of object-oriented languages. The notion of orthogonal planes is used to show the class hierarchy and method call graph simultaneously. Classes are put on the “main” plane and the methods for each class are laid out on an orthogonal plane emanating from the class. Nodes can be moved to afford a more desirable layout if the user is interested in some specific thing. This is illustrated in Plate V.

4.2.3 Features and Functionalities

The browser is enhanced to be able to read in compiled C++ programs and to extract and display the hierarchical relationships between the classes as well as the call graph between the methods. It can also keep track of the progress of the execution of the programs changing the intensity of the nodes representing the methods and classes to indicate which nodes are active, which nodes are lower on the runstack and which nodes are not active. And, finally, as proposed in section 3 the thickness of the nodes profile the number of times the methods they represent are invoked. Refer to Plate VI for an example. The video that accompanies this paper demonstrates the ideas of this section in practice.

5 Future Work

On the implementation level, the applications are meant to be demonstrations of the principle and concepts of section 3: their validity, range of applicability, usefulness and usability. They were not meant to be full-fledged, stand-alone systems and a logical next step would be to make them in robust working systems. The appropriate thing to do, in this case, would be to study the design direction of *FIELD*, and to try to integrate the system into the *FIELD* environment. *FIELD* is a very large system with many tools available. The layout algorithm, for instance, should use the already existing one in *FIELD* that is much more robust. A strategy whereby all the methods can be snapped on to one plane easily (recall each method lies on a plane that emanates from its parent class) can be a useful tool.

For object-oriented languages that are used in an environment where hierarchical structures are more prevalent, allowing recursively defined orthogonal planes (orthogonal planes *of* orthogonal planes) is desirable. Using the example of section 3.2 we can extend the category-domain structure to include, also, the actual operations each domain exports. Integers, for instance, would export addition but not division (since integers are not closed under division). The operations could be laid out on a plane orthogonal to their parent domain which lies in a plane orthogonal to its parent category.

What is forwarded in this paper is more of a design strategy that may have many applications rather than simply an application, by itself. Extensions is thus used rather loosely, here. Almost anything that serves to apply the proposed principle and searches for new ways of infusing semantic information into an environment by using orthogonal channels of information is of interest. *Orthogonal channels*, in this context, appeal to a much more general notion. The use of audio capabilities, briefly demonstrated in appendix B, is one example.

6 Conclusion

This paper presented a design strategy of using three dimensional graphics in a way that extends rather than replaces two dimensional technologies in information visualizations. By treating the orthogonal third dimension asymmetrically from the other two dimensions, we can use it to convey semantic information that would otherwise be difficult to convey clearly in two dimensions. Planar techniques developed for two dimensional visualizations are retained but the planes now reside in a more general three dimensional space. The advantage is that there are an infinite number of two-dimensional planes in three-space and each plane can potentially group data together under a common semantic relation. Additional information not captured by the planar layout can be encoded in a direction orthogonal to the plane of definition so as to, at once, preserve the original two dimensional layout as well as convey new information. Perhaps the approach can be best characterized as a two-and-a-half-dimensional approach to visualization. By recognizing the advantages of, and exploiting our wealth of knowledge in utilizing two dimensional space for visualization, it may be possible to use three dimensional graphics to go beyond just visualizing more abstract data but

to visualize more complex data and data that is much richer in semantics.

Appendices

A The Browser: A User's Reference

A.1 Starting the System

The name of the system is `3dflat`—referring to flat two dimensional planes existing in three-space. A small menu box pops up along with the main viewing window and a camera control slider box. The `FILE` option allows the loading and saving of files, quitting the program, etc. The `CAMERA` option sets the camera to some preset positions in case the interactive movements has gotten out of hand and the graph is nowhere to be seen.

Various options are available as command line arguments as seen in the table on page 17.

-option	argument	effect
-g	filename	read in graph data file
-c	filename	c++ flow debugger mode
-f		use with <i>FIELD</i>
-N		profile nodes in c++ mode
-A		run with audio server on
-S		default with speaker turned on
-M		monochrome mode (white background)
-L		suppress layout recalculations
-t		put up default test graph

If the system is be used as a program flow visualizer, as described in section 4.2 then, in addition to using the `-f -c` program options, `flowview` program and `gddt -f` program should also be running. Refer to their respective man pages for available options.

A.2 Mouse Bindings

Figure 8 gives the mouse bindings for the browser.

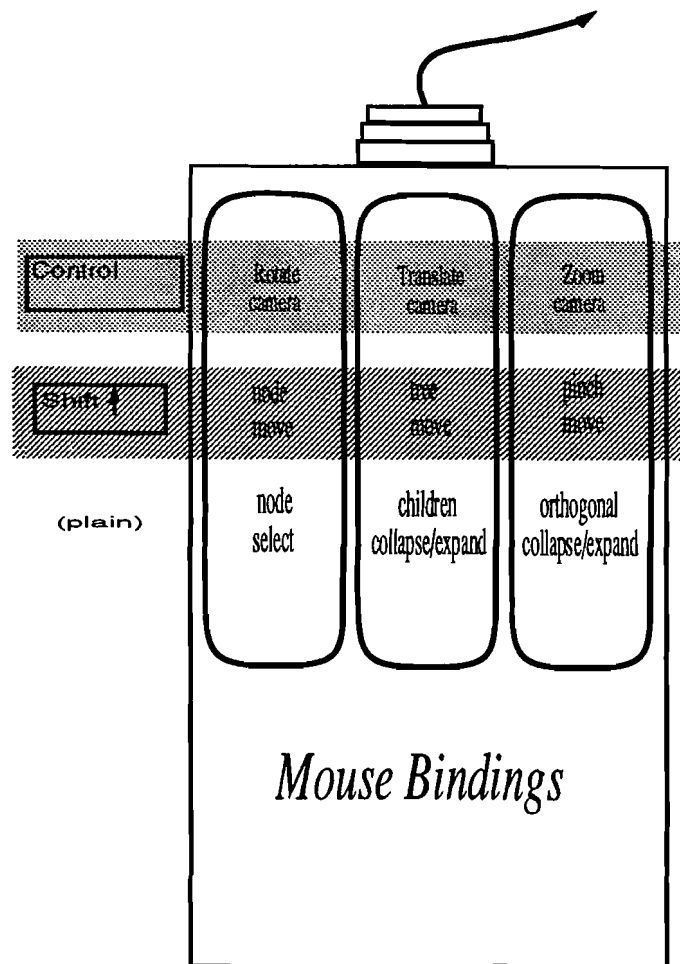


Figure 8: Mouse button bindings

No keys depressed

With no keys held down, the mouse has the following functionalities: The LEFT button *selects* a node (see SHIFT mode for the moving of selected nodes, and section A.3 for key binds that act on selected nodes). Selecting a selected node deselects it. The MIDDLE button expands and collapses the children nodes. The RIGHT button expands and collapses the orthogonal nodes.

Shift key depressed

Holding down the SHIFT key puts the mouse in *node move mode*. The LEFT button moves the selected node to the projected point directly under the mouse; if multiple nodes are selected, their average position is moved. The MIDDLE button moves the selected node(s) and its/their subgraphs *relatively*—that is, the mouse must be moved to create a displacement. The node(s) will not snap to the mouse position, as above, but will be displaced by the amount the mouse moves. The RIGHT button moves the selected node(s) relatively as well. The neighbors (parents and children) are also dragged along. Neighbors further and further away are dragged less and less. This is the “pinching” mode as described in sections 3.2.3 and 4.1.2 that facilitates in the visualization of neighborhoods.

Absolute movement provides for precise fine-tuning but relative movement was chosen for the latter two modes because those two modes both move nodes that are not explicitly selected. By using the displacement resulting from a change in the mouse position, the relative mode allows for the movement of all the nodes even if the selected node to be out of view.

Control key depressed

Holding down the CONTROL key puts the mouse in *camera mode*. By camera, we refer to the user’s view of the scene. The viewpoint can be interactively changed: the LEFT button *rotates* the camera, the MIDDLE button *translates* the camera and the RIGHT button *zooms* the camera in and out.

A.3 Keyboard Bindings

The following table, on page 20, gives the key bindings available. Those that expect a node to be selected have an asterisk (*) in the middle column.

r		Reset camera
l		Recalculate layout
t		Show/Hide Text (text is slow)
.	*	Raise node in Z direction
,	*	Lower node in Z direction
=	*	Pinch node up in Z direction
-	*	Pinch node down in Z direction
>	*	Raise all children in Z direction
<	*	Lower all children in Z direction
c	*	Select all children
p	*	Select all parents
o		Show all orthogonal planes
q	*	Rotate orthogonal plane clockwise
Q	*	Rotate orthogonal plane counterclockwise
←		Rotate camera left
→		Rotate camera right
↑		Rotate camera up
↓		Rotate camera down
a	*	Define sound for node
A	*	Preview (play) sound for node
s	*	Turn sound for node ON
S	*	Turn sound for node OFF
x		Turn browser audio capability ON
X		Turn browser audio capability OFF

A.4 File Format

The applications developed for this paper handles three types of files: compiled *C* or *C++* code, graph definition in ASCII, and audio files. The compiled code is handled through the various facilities of *FIELD* and is transparent to the user. The graph definition is user-defined and is described

below. The audio files follow the currently supported format on the Sun Sparc workstations.

A.4.1 Graph Definition

The file format used to define graphs is given, below. White spaces are ignored and case does not matter. Both **children:** and **orthogonal:** are optional but all entries must be on separate lines. An optional filename for the audio data can follow the name of the node.

NODE:

```
node1 [audioname]
CHILDREN:
child1
child2
child3
ORTHOGONAL:
ortho1
ortho2
```

NODE:

```
node2 [audioname]
.
.
.
```

A.4.2 Audio Definition

The audio field in graph data file described in section A.4.1 refers to the audio data file. The audio data file is a file that contains the sound to be generated for a particular node. Any file with the proper format could be used. Special samples were made for the audio components of this system. Briefly, audio files are stored in *ulaw* format—which is a format that compresses the audio sampling of the sound (as a waveform) logarithmically. It is the standard format used by platforms such as the Sun Sparc's and the NeXT computers. Refer to the man pages for more information on how to create such files.

B Other Orthogonal Channels

While the emphasis of this study is in exploiting the orthogonality of the third dimension to visualize orthogonal information not currently handled by two dimensional means, it is certainly not limited to that. The principle of using new channels of information orthogonally so as to extend existing systems rather than redesigning current systems to account for such things is applicable to other channels as well. Take the audio channel, for example. Section 3.2 proposed that the depth, or thickness, of nodes encode the number of times a node is invoked as an aid to visual the direction of the flow of a program in terms of stack depth. We can, in fact, encode that in another way: we can have a background sound that varies its pitch in proportion to the depth of the call stack.

In this context, we give an example of a functionality that goes well with the C++ program flow browser. It fits well into the framework of visualizing large amount of data—in this case, the aide is in a temporal sense rather than a spatial sense. That is, rather than organizing the data spatially to help us gain a better and more efficient visual understanding of the data, we are given, in the call browser a cue as to when to concentrate on the data. Prior work in this area has included using sound to debug programs in a parallel environment [3] and to visualize program flow [1]. We use it as an extension to the program flow browser because it aides in the assimilation of a large amount of data by allowing the user to save visual concentration for when interesting events actually occur.

B.1 Audio Example: the Audible Trigger

Here, we use the audio channel to further illustrate the principle of orthogonality. The obvious difference between audio capabilities the visual capabilities of two dimensional graphics makes it simple to understand the virtues of using it to carry some other form of information. Certainly, this is using something not found in the traditional layouts taken straight from the books and put on the screen. The extension it provides is not, however, spatial.

One particular problem with a visual system is the focus and concentration it demands from the user. A passive user may easily miss a visual cue in a graphical debugger if attention is compromised for even a little bit. It may be impossible to tell, subsequently, whether anything has been missed or not.

This holds true for both two dimensional and three dimensional systems. A solution would be to allow the user to tag an audible trigger to a node so that when the node is visited, the trigger would go off causing some form of sound to inform the user of the fact that a particular event (or visit) has occurred.

This system allows the user to select a node and assign a sound to it. The sound is played when the node is made active (called and placed on top of the run stack).

Taking this notion one step further, in the environment of orthogonal planes as described above, a node not on the main plane can, when the audio signal is triggered, prefix its signature sound with those of its ancestors. In the case of C++, then, a parent of class C that is manifested in a node $\mathcal{N}(C)$ with the sound $\mathcal{S}(C)$ can have methods $m_{C,i}$ with each having its own signature $\mathcal{S}(m_{C,i})$. The invocation of a method, $C :: m_{C,i}$, would cause the sound $\mathcal{S}(C)$ to be created followed by the sound $\mathcal{S}(m_{C,i})$. The user can then identify a function class and method without even looking at the console.

C System Architecture

The implementation of the system is divided into three parts: (1) the base system which consists of the mechanisms of the browser both graphically and audially, (2) the applications which are built on top of the base system, and (3) the server units which act in concert to create an interface between two very large software projects: *FIELD* [8] and *UGA* [14].

The design of the first two parts are given in the main paper: the base systems is based upon section 3 and appendix B and while the applications are described in section 4. Here, we will comment briefly on the design of the interface that is not discussed elsewhere. Interest in creating such an interface has existed for some time but while the two projects are well suited to be interfaced together they are under the auspices of separate groups and, up to now, few people have been involved with both projects. The differences in the approaches to software engineering in the two projects make for an interesting study in contrast in itself but that will not be dwelt upon here.

C.1 The Interface Units

The interface of the browsing system is realized by creating the framework that sits between and interfaces two very large software projects: *UGA*, the Unified Graphics System under development by the Brown Computer Graphics Group and *FIELD*, a program visualization system under the direction of Professor Steven Reiss. Both systems are very large and each has its own method of communication.

The strategy taken by this project was to create a mini-server that sat between *FIELD* and *UGA*. The mini-server initially creates a child process and sets up the sockets to communicate with it. The parent process is dedicated to the *FIELD* system and, after initialization, it enters into a blocked state listening for messages from the message server of *FIELD*. The child process is dedicated to the *UGA* and, after setting up the browser, it enters an event loop listening for user initiated events. The child is responsible for all the graphical and audio input/output. The child is also hooked into *FIELD*, actually, but only in a synchronized manner within the browser: when it needs to request information on file format, etc. All asynchronous events from the graphical side are handled by the child. All asynchronous events generated by the program being tracked, “flow events”, are caught by the parent which then relays them to the child via the private sockets set up between the parent and the child. In this manner, the two systems are connected into one application. A schematic of the architecture of the system is given in Figure 9.

References

- [1] Marc H. Brown and John Hershberger. Color and sound in algorithm animation. Technical Report 76a, Systems Research Center, DEC, aug 1991.
- [2] John B. Fraleigh. *A First Course in Abstract Algebra*. Addison-Wesley Publishing Company, 1989.
- [3] Joan M. Francioni, Larry Albright, and Jay Alan Jackson. Debugging parallel programs using sound. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, 26(12):68–75, 1991.

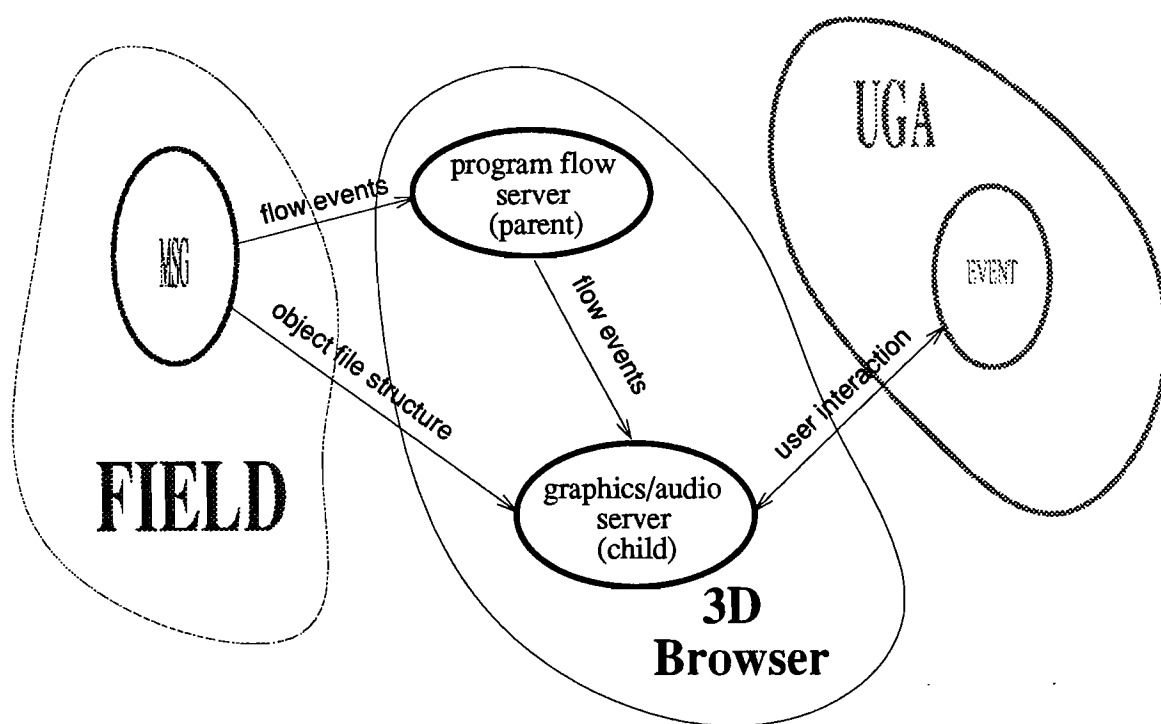


Figure 9: Interfacing *FIELD* and *UGA*

- [4] George W. Furnas. Generalized fisheye views. *CHI '86 Conference Proceedings*, pages 16–23, 1986.
- [5] William W. Gavett. The sonicfinder: An interface that uses auditory icons. *Human-Computer Interaction*, 4:67–94, 1989.
- [6] Ernest R. Hilgard, Richard C. Atkinson, and Rita L. Atkinson. *Introduction to Psychology*. Harcourt Brace Jovanovich, New York, 1971.
- [7] D. A. Henderson Jr. and S. K. Card. Rooms: The use of multiple virtual workspaces to reduce space contention in a window-based graphical user interface. *ACM Transactions on Graphics*, 5:211–243, 1986.
- [8] Steven P. Reiss. Interacting with the field environment. *Software Practice and Experience*, 20(1):89–115, 1990.
- [9] George G. Robertson, Jock D. Mackinlay, and Stuart K. Card. Cone trees: Animated 3d visualizations of hierarchical information. *CHI '91 Conference Proceedings*, pages 198–194, 1991.
- [10] George G. Robertson, Jock D. Mackinlay, and Stuart K. Card. The information visualizer, an information workspace. *CHI '91 Conference Proceedings*, pages 181–188, 1991.
- [11] George G. Robertson, Jock D. Mackinlay, and Stuart K. Card. The perspective wall: Detail and context smoothly integrated. *CHI '91 Conference Proceedings*, pages 174–179, 1991.
- [12] Manojit Sarkar and Marc H. Brown. Graphical fisheye views of graphs. Technical report, Systems Research Center, DEC, 1992.
- [13] Edward R. Tufte. *Envisioning Information*. Graphics Press, Cheshire, Connecticut, 1990.
- [14] Matthias Wloka, Nate Huang, and D. Brookshire Conner. Uga software standards. Technical report, Brown University, 1992.

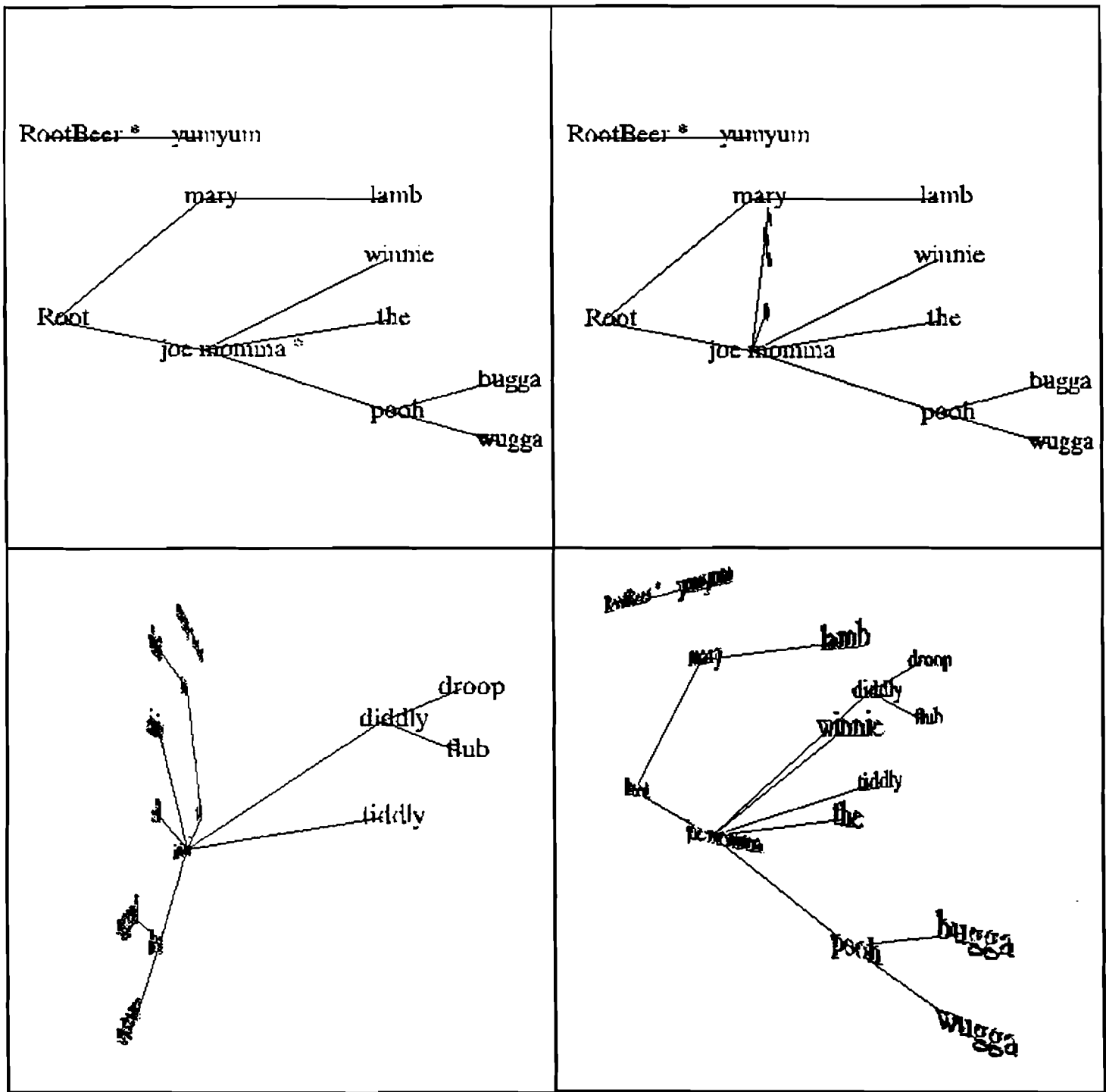
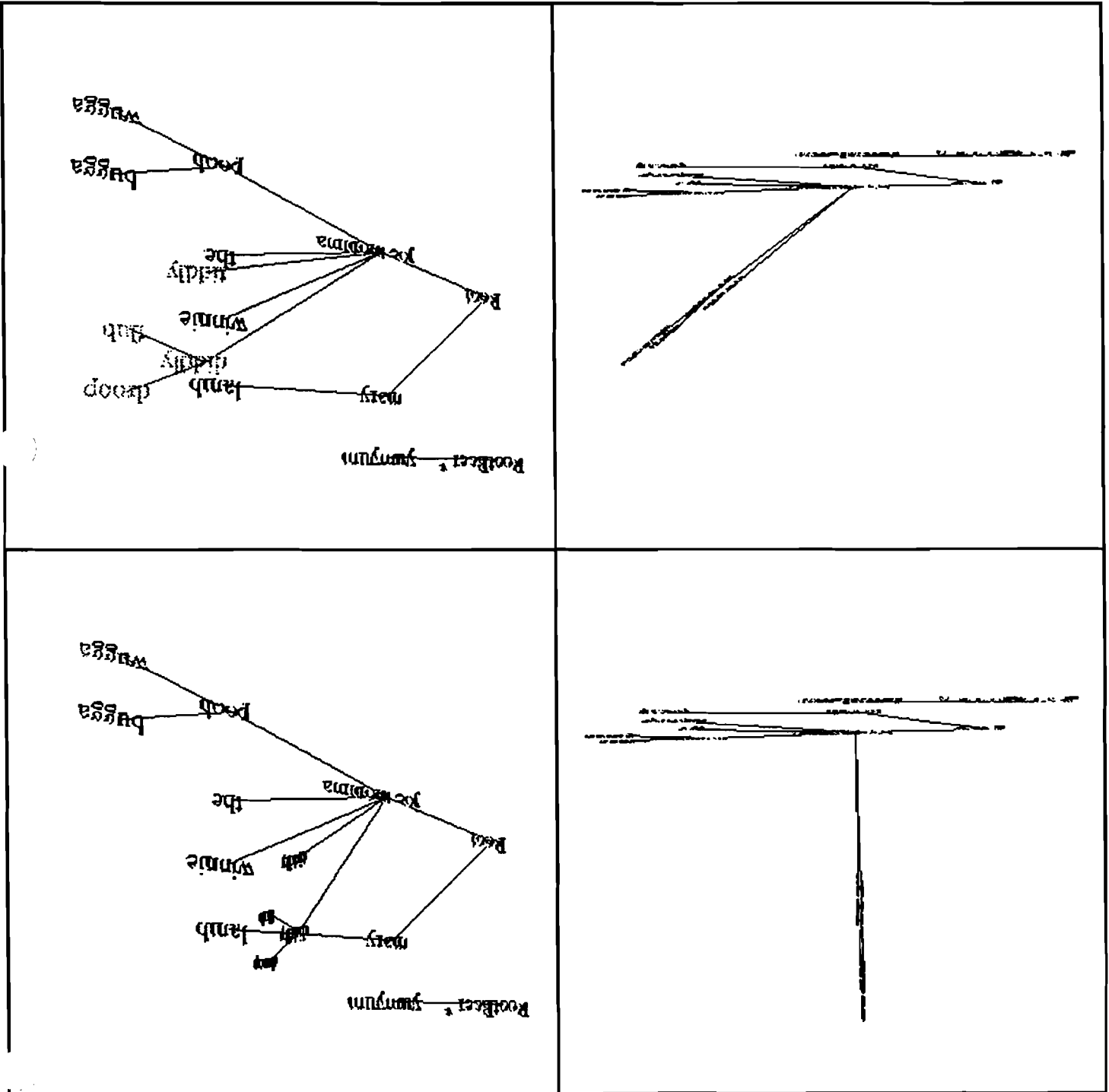


Plate I: An Orthogonal Plane

Plate II: Viewing Both Planes



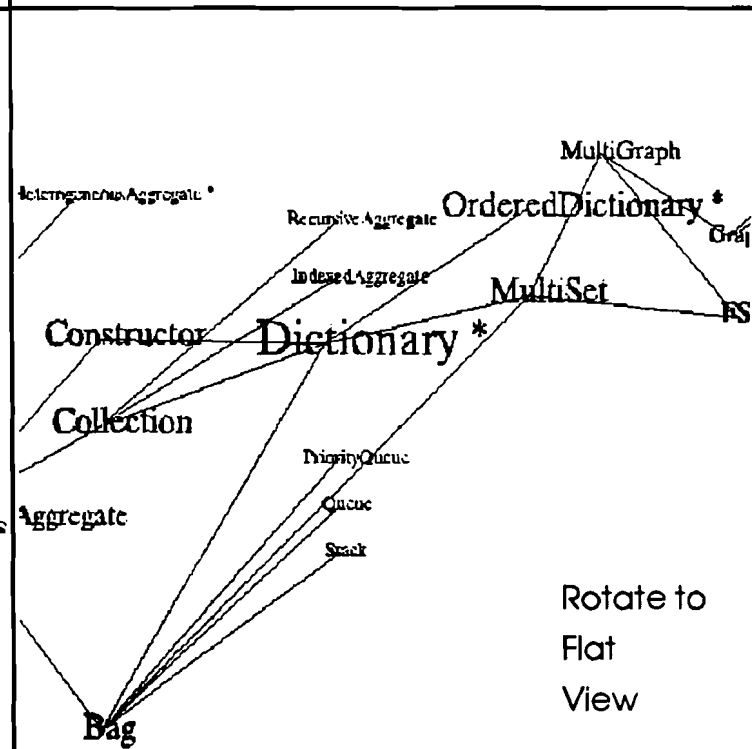
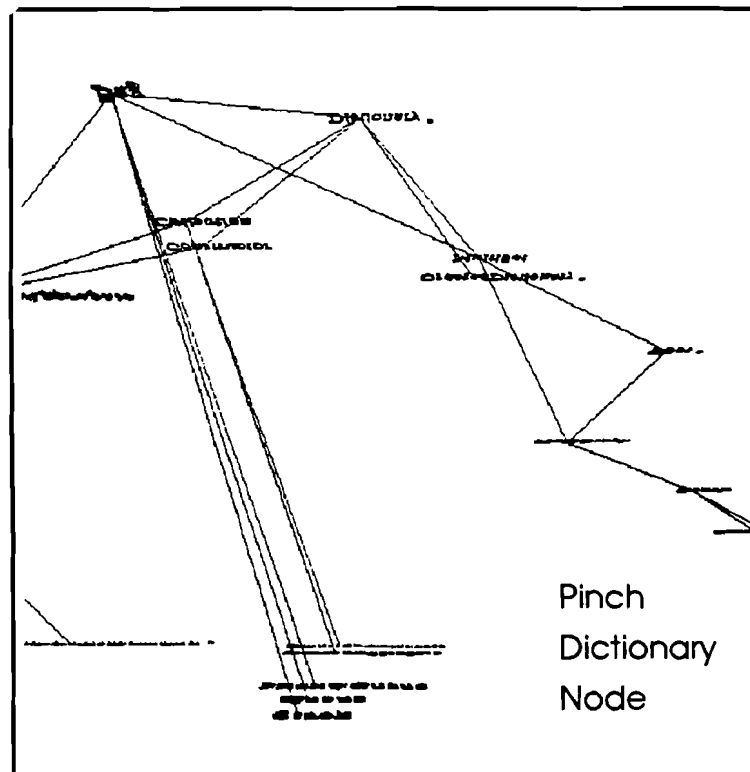
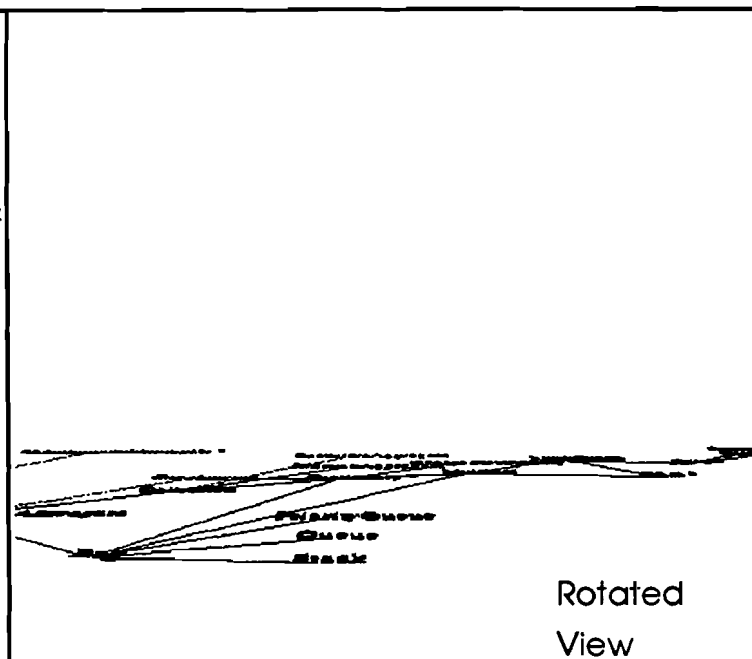
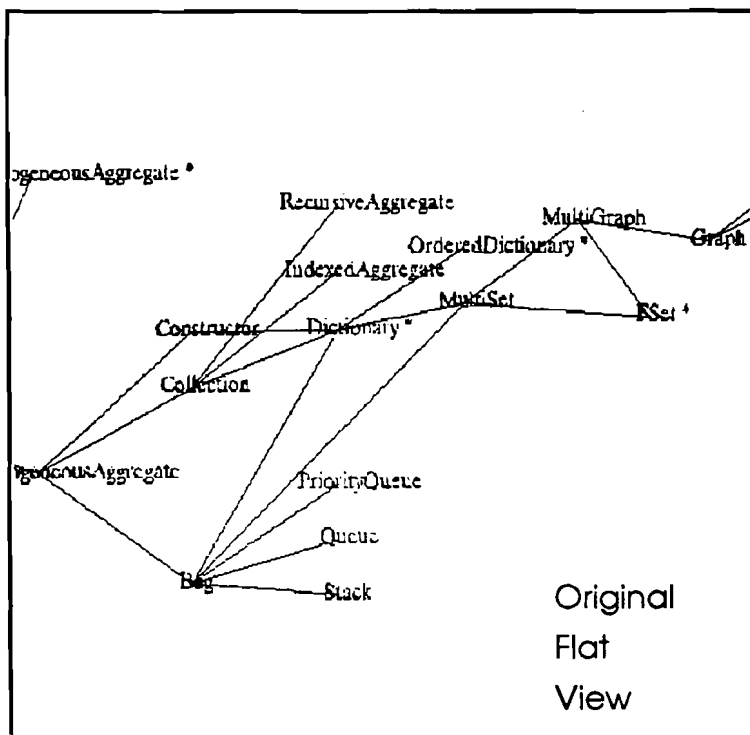


Plate III: Grouping By Height

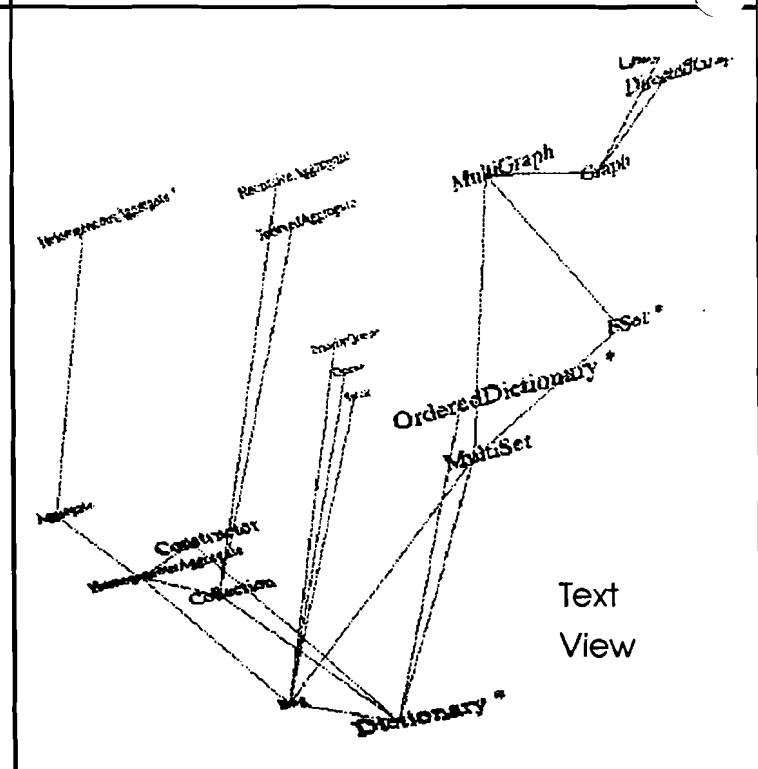
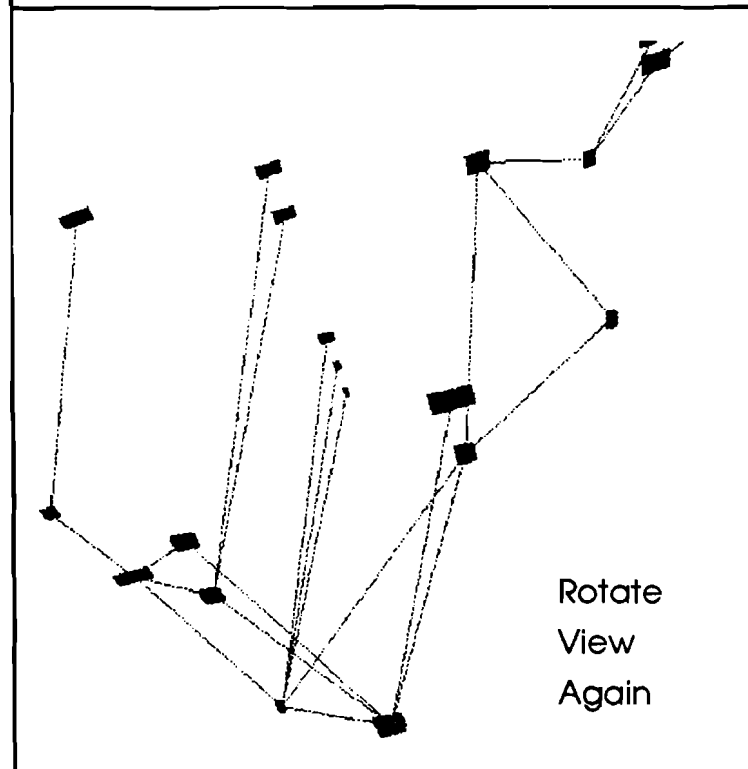
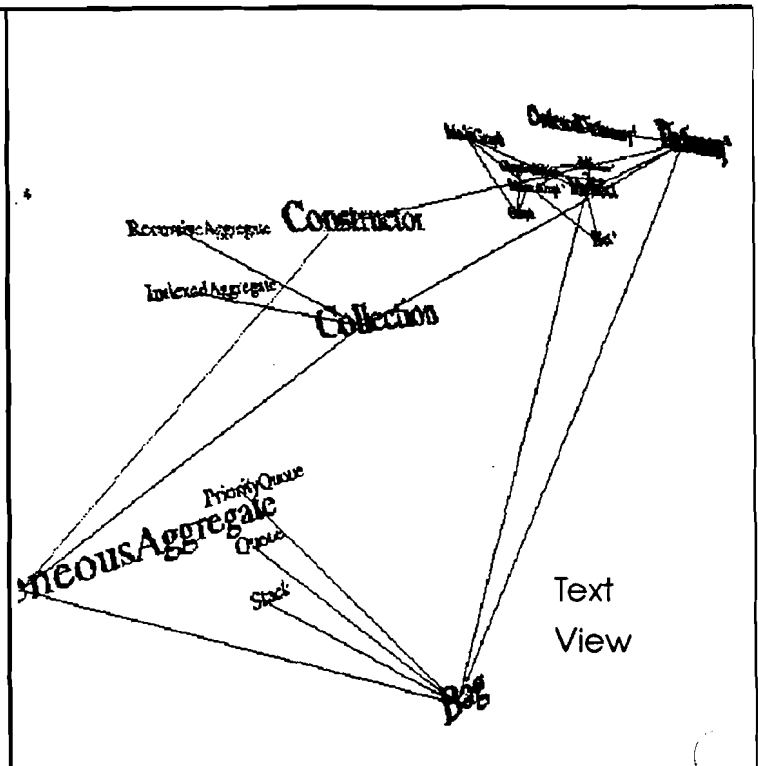
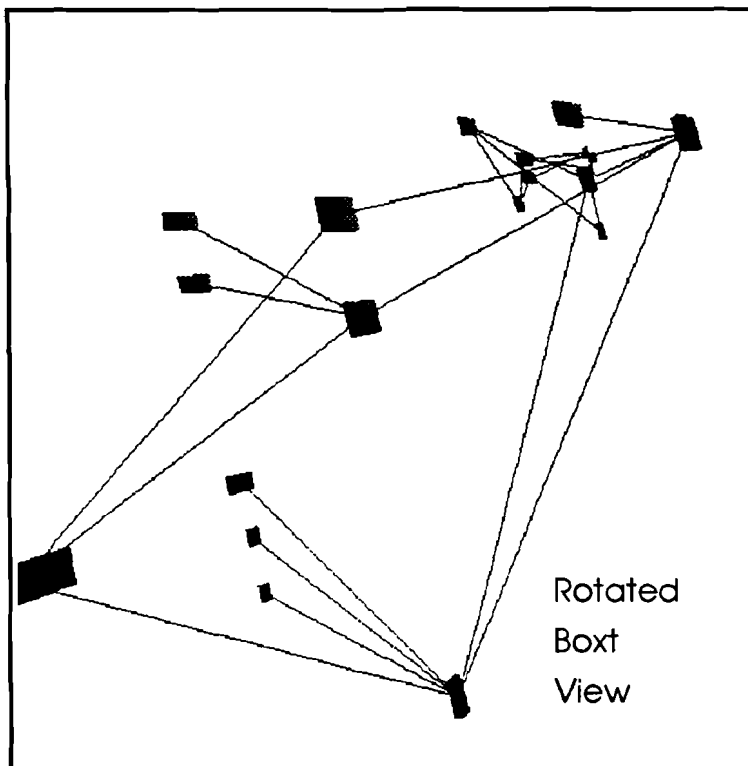


Plate IV: Grouping By Height (cont'd)

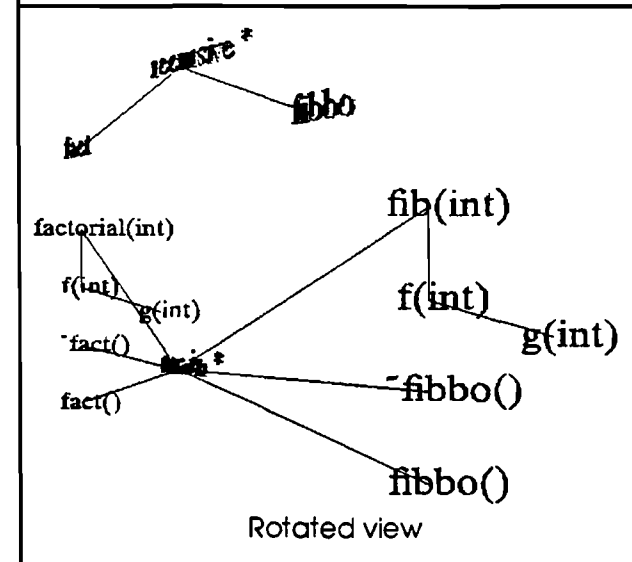
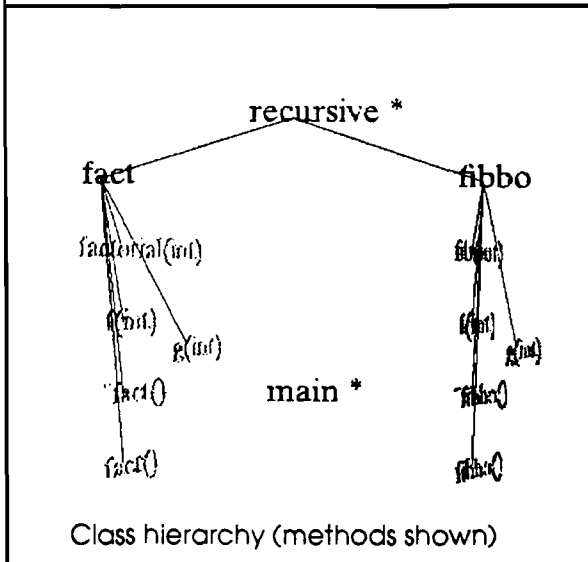
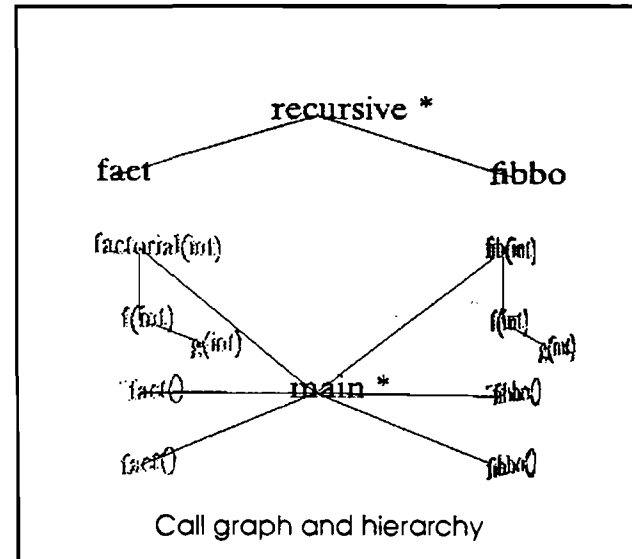
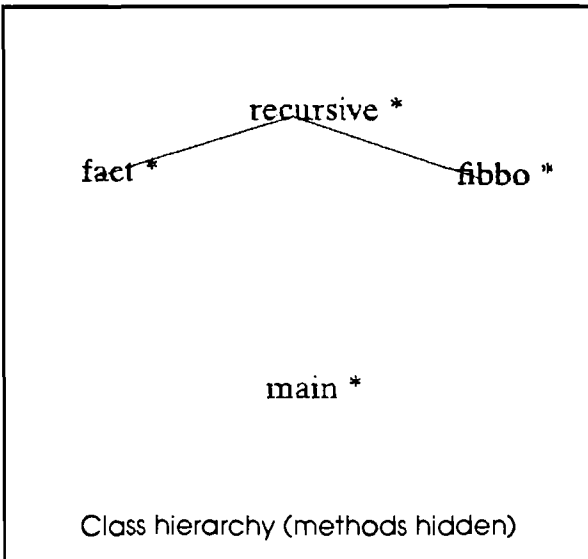
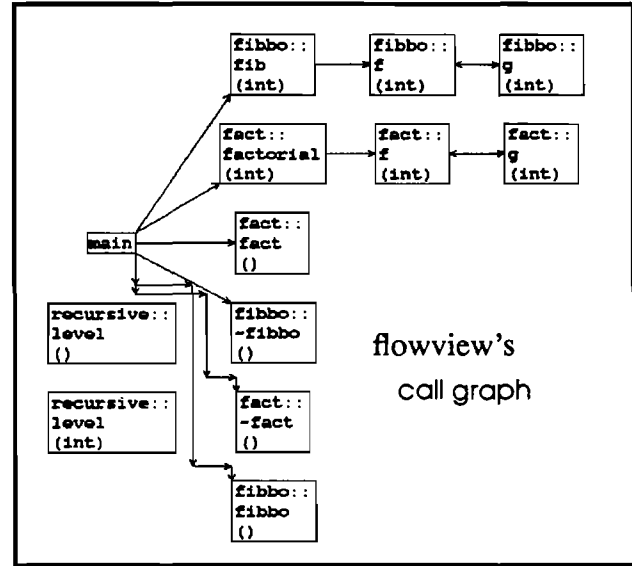
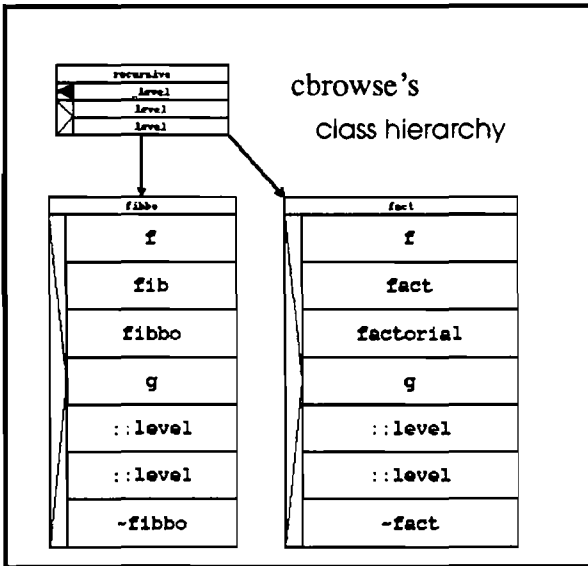


Plate V: Orthogonal Browser in an Object-Oriented Environment

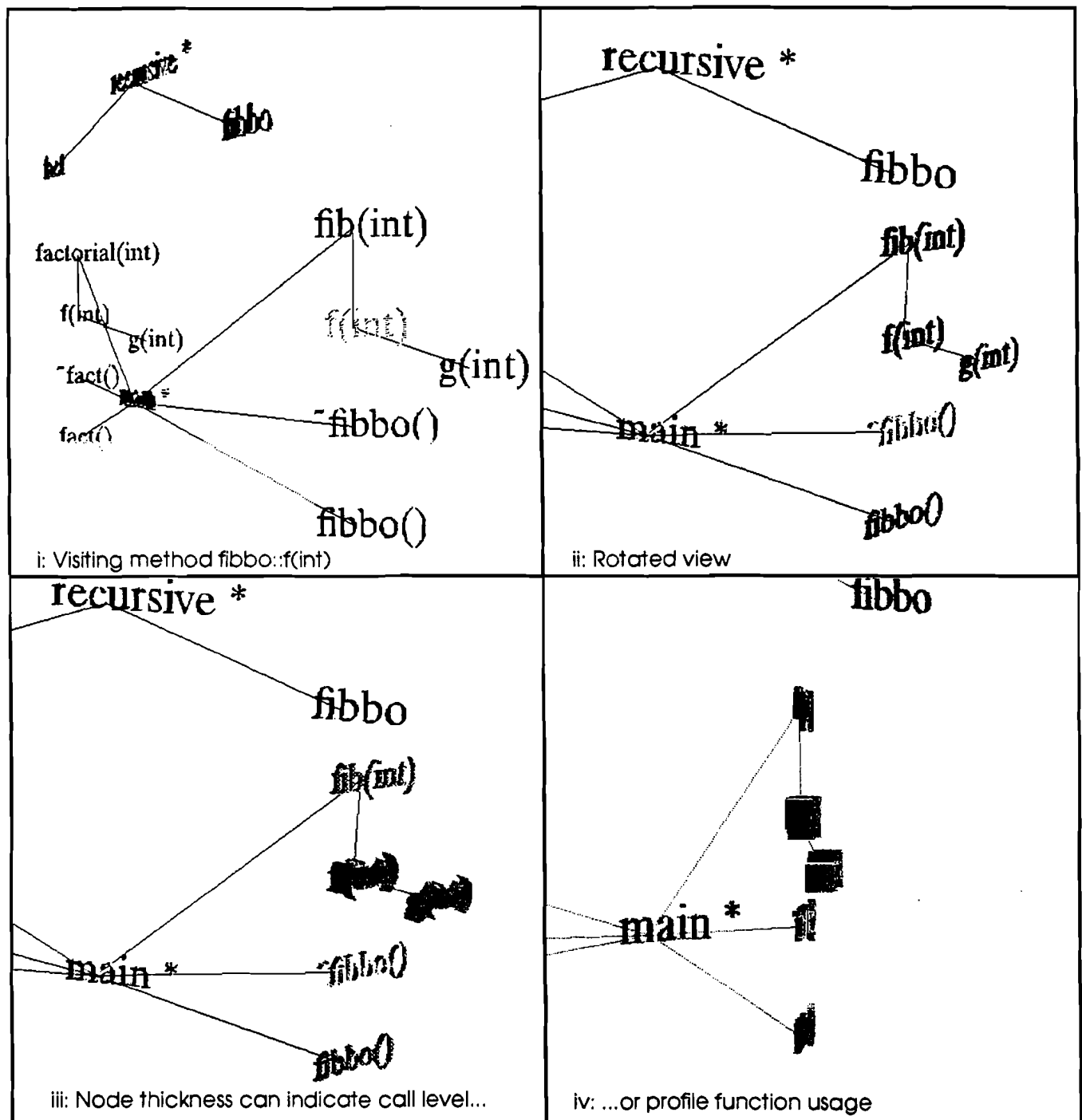


Plate VI: Using node thickness to hold run-time information