A Framework for Abstract 3D Visualization

Steven P. Reiss[1]

Department of Computer Science, Brown University, Providence, RI 02912
(401)-863-7641, spr@cs.brown.edu

Abstract

This paper describes a package, PLUM, we have developed for visualizing abstract data in three dimensions. We are particularly interested in visualizing information about programs, both static and dynamic, but the package should have a more general applicability. The package provides a framework to support a wide variety of different 3D visualization techniques, many of which have been implemented. The package also provides support for 3D graph layout using a variety of different layout heuristics.

1.0 Introduction

This paper describes a package for visualizing abstract data, particularly information about programs. Our goal is to provide a system where the programmer can specify what information should be displayed and how it should be displayed with a minimal amount of work. We are designing a framework to accomplish this, complete with declarative visual and textual languages. This framework is based on a package for abstract 3D visualization, PLUM, that is the topic of this paper. This package encompasses a wide variety of different strategies for presenting program visualizations and provides an extensible backbone for these strategies.

Program visualization is the graphical display of information about a program. While there have been many program visualization efforts, these have been limited in both scope and application because the amount of information to be included is far more than can be displayed. Practical program visualization must provide tools to select and display just the information of interest. It must provide quality visual displays that look "nice" and offer a user-friendly interface for browsing and querying. Finally, practical program visualization must make use of the capabilities of modern workstations including 3D graphics to provide as much information as possible in a small display.

A practical program visualization system can be achieved by focusing on abstractions [19]. Abstractions can be specified as queries on a heterogeneous object-oriented database. The basic idea of looking at programs through a database was explored by Linton [14]. This work assumed a single relational database of program information. We start with program information from various sources. These are united using an extensible object-oriented database schema rather than a full database. Both textual and visual query languages are provided for this schema. The result of the query is a set of objects. These are stored in an in-core object-oriented database as the source for visualization.

The visualization of program information can be viewed as the definition of appropriate graphical output for a set of abstraction objects. This is done in two steps. The first step is to map the abstraction objects into objects that represent an abstract graphical structure such as a layout containing objects representing nodes and arcs. At this level, no information about position, layout, routing, etc. needs to be provided. To make these mappings declarative, the target space of graphical objects must be well defined. One of the tasks that we have undertaken in PLUM is to develop a catalog of approaches to 3D presentation of structured data to explore this space. PLUM also provides the mechanism to integrate these approaches and to add new approaches easily.

The second step is generating a display from the abstract graphical structure that results. This involves automatic layout and constraint satisfaction for each of the graphical objects. This also involves supporting incremental modification of the display through animation.

In the next section we describe related work. Section 3 looks at the design space for 3D structured presentations, considering the various strategies that have been implemented already in PLUM as well as some that we have considered but not yet implemented. The fourth section looks at the structure of PLUM itself to understand how it serves as a backbone for these presentations. The fifth section looks at the problem of layout and routing in 3-space. We conclude by discussing how we have been using PLUM and the open problems.

2.0 Background

While there has been substantial work on program visualization, most of this work has been directed toward providing specific visualizations such as a call graph or a class browser, and little has been directed toward a generic

[1]Support for this research was provided by the NSF under grants CCR9111507 and CCR9113226, by DARPA order 8225 and by ONR grant N00014-91-J-4052

framework. The work that is closest to our approach includes our earlier efforts on data and program visualization, work related to the display of user data structures, work directed at graphical editing, work on systems for algorithm animation, and visualization efforts that attempt to use a single paradigm for a variety of applications.

Our previous work addressed the issue of 2-D visualization of abstract data [15,17]. This work supported our work on visual languages in the GARDEN system [16]. It was used to display a variety of different visual languages including Petri nets, statecharts, finite automata, flow charts, and data flow diagrams, as well as arbitrary user data structures. The package was later used in the FIELD environment to support browsers for call graphs, class hierarchies, and make dependencies [18]. The package had three parts. The first, GELO, provided a framework for abstract 2-D displays. The application created displays by building a hierarchy of GELO objects. There were four basic flavors of objects, data objects to represent simple nodes, tiled objects to represent structured composites, arc objects to represent connections, and layout objects to represent graphs. By providing a variety of different shapes and forms of data objects and by providing flexible heuristics for layout objects, GELO could compose these objects to form the different visual languages and most of the desired data structure displays. The second package, APPLE, provided an automatic mapping facility from user data structures into GELO graphic objects. The mapping could be specified by the user declaratively using a visual editor. APPLE also provided reasonable default displays for most structures. The final component, PEAR, provided graphical editing capabilities. It offered a user interface for manipulating the resultant diagrams, and provided callbacks to the application for editing operations.

GELO was not the first system that attempted to display user data structures. Early work in this area by Brad Myers allowed the user to program a display using a graphics library to code the display for each type [12]. Later work by Baskerville attempted to integrate simple displays into a debugger [1]. Recent efforts along these line include VIPS [9], and the commercial data structure display facilities provided by Centerline's C environment and by SGI's Codevision. This work is all fairly specialized in that it attempts to provide standard displays of data structures. Myers' efforts allowed the user to design the data structure display, but required the user to do this design procedurally.

PEAR demonstrated for us the utility of providing a general purpose graphical editor as part of a user interface toolkit. This has also been recognized by a number of other groups and there have been a variety of generic graphical editors that can display abstract program data. One of the earliest such editors was Unidraw developed as part of Interviews [10]. This editor used object-orientation to provide an extensible framework for editing somewhat similar to that we provide in PLUM. Later examples include Go [5] and a variety of graph drawing widgets for Motif. The Garnet environment provides a slightly different basis for editing [13], a powerful low-level environment based on constraints that can be used to build a higher level graphical editor.

Another area in which a general display mechanism supports a variety of applications is algorithm animation. The Balsa system provided a high-level graphics library where different animations could be easily coded [2]. The TANGO system followed this up by providing a formal framework consisting of an animation algebra where the animations could either be coded procedurally or by demonstration [22]. More recent work on Zeus added color and sound and is now incorporating 3D visualizations [3,4]. While these efforts are suitable for a variety of different animations, they concentrate on providing high-quality displays of smaller amounts of information and generally expect the developer to do a substantial amount of work in implementing the animation.

There have been other efforts aimed at providing generic display facilities for a variety of applications. Flynn and Maier have worked on the specification of displays for objects from an object-oriented database [8]. While this work is related to abstraction visualization, their graphical displays are quite limited. Work at Bell laboratories has applied a single file visualization technique to a variety of different applications [7]. We have incorporated their ideas on file display into our system as one of the presentation mechanisms we provide.

3.0 Abstract graphical objects

Abstract graphical objects provide the foundation of our approach to visualization. They are the target of the mappings from abstraction objects and the starting point for generating a concrete 3D presentation. As such they must satisfy several criteria:

- **Completeness**. They must be able to represent all the different target visualizations that we want to include.
- **Extensibility**. Since we cannot anticipate all visualizations initially, it should be easy to add new graphical objects to the system.
- **Hierarchy**. Each visualization should be broken down into its basic components which in turn should be abstract graphical objects. This makes for simpler objects and more logical mappings from abstraction objects to graphical objects. It also makes combining visualizations easy.
- **Parameterization**. All the properties of each graphical object should be parameterized. This allows the mapping from graphical objects to specify these properties and supports declarative mappings.
- **Abstraction**. The graphical objects should be display independent. They should not require placement information nor should they make assumptions about their physical location or size on the display.
- **Concreteness**. At the same time, the objects should allow the setting of display properties. This permits the user to move objects around by saving the resultant placement and allows location and size to represent characteristics of the abstraction objects.

Abstract graphical objects are represented in PLUM as entities with properties, constraints, and components. The properties represent the parameters that control how the object is to be drawn. The constraints relate objects to each other. The components identify other objects that are contained within this object.

A common set of properties is shared by all objects. These include stylistic properties, sizing information, and a priority setting. The priority value is a general means for specifying that an object is important and should be emphasized in the display. Different graphical objects treat this differently. The style properties include color (surface, text, line, and marker), font (family and size), line and fill styles. The sizing properties allow the object's size to be scaled by a multiplier and a addend independently in the three dimensions. This is useful for explicitly controlling the size of the object, for example making the size proportional to a value associated with the abstraction data.

As a first step toward a viable framework for visualization, we determined the set of abstract graphics objects that covered a representative sample of the desired visualizations. We drew the visualization ideas from our own experience, from a review of the literature, and through exchanges with other researchers.

The initial set of abstract graphics objects were drawn from our previous experience with GELO. These are 3D analogues of the generic objects provided by GELO:

• **Data Objects**. These represent a box in 3-space that contains a shape and an optional text string. The shape can be either 2D or 3D. Additional properties of data objects include the basic size when no text is included (the size with text is dependent on the size of the text string and the font size), the ratio of width to height and width to depth (to make squares, cubes, etc.), and the opacity of the object. Data objects have no constraints or component objects.

Figure 1 shows an example of a 3D call graph display generated using PLUM. Each container box represents a file and each smaller box represents a routine. Arcs represent call connections between functions in a file or calls from routines of one file to those of another. In this diagram data objects are used to for each of the function boxes as well as for the labels on each of the file boxes.

• **Arc Objects**. These represent connections between two objects. An arc object always appears as a component of some other object, typically the common parent of what it connects. The properties associated with an arc object specify the location where the arc connects to the source and target objects, the type and style of arrows, and whether the arc should be splined or not. Arc objects support components in the form of labels. Each label can be placed anywhere along the arc. Figure 1 contains several examples of arc objects, green arcs for connections within a file box and red group arcs for connections between files. The line thickness of each arc denotes the number of connections it represents.

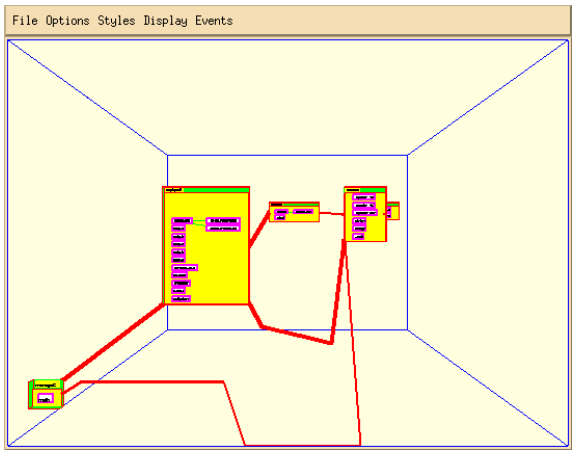

FIGURE 1. Call graph display showing data, arc, tiled and layout objects

• **Tiled Objects**. These represent connected groupings of objects. Each tiling consists of a 3D block that is subdivided into rectilinear regions or tiles. The tiling assumes an integer coordinate space and each tile is specified by providing the object contained in the tile and two diagonally opposite corners of the tile in this coordinate space. Simple tiled objects are used in figure 1 for the file boxes. Each file box is a tiling containing two components, a data object containing the name of the file and a layout object containing the routines defined in that file.

The tiled object determines the layout of the tiled components by solving a system of constraints. These constraints are defined by the desired sizes of the components, the requirement that the coordinate space represented by the components be consistent (i.e. that all tiles that have an X coordinate of 1 must line up), and by additional constraints that can be associated with the tiling. The additional constraints linearly relate two dimensions, assign a particular dimension a constant size, or specify the degree of flexibility for each component. The various constraints are mapped into a set of linear equations that solve for the positions of each tile coordinate. If the system is under constrained (which it normally is), then additional implicit constraints are added to make the tile coordinate space be proportional to the resultant layout. If the system is over constrained, then the specified constraints are prioritized and are eliminated in groups until the system is solvable.

• **Layout Objects**. These represent a rectilinear region that contains two types of components, nodes and arcs. The object is responsible for doing a layout of the nodes within the region using the connection information specified by the arcs. The properties supported by layout objects control this layout. The layout is done by applying some heuristic, settable as a property, to assign relative positions for each component. These relative positions correspond to a 3D array of blocks, each of which can contain one object. The layout object uses this relative positioning to compute the position of each row, column, and

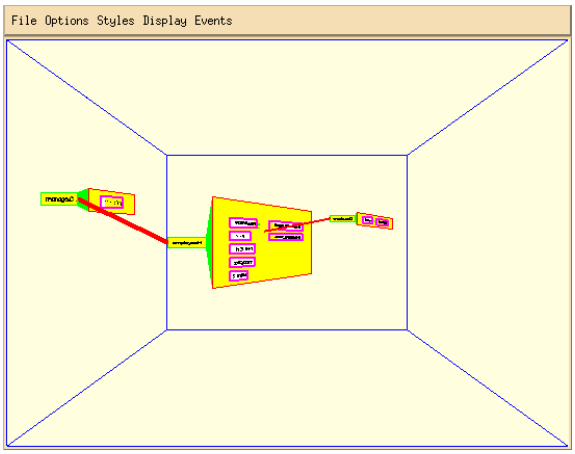

FIGURE 2. Call graph display using tagged objects

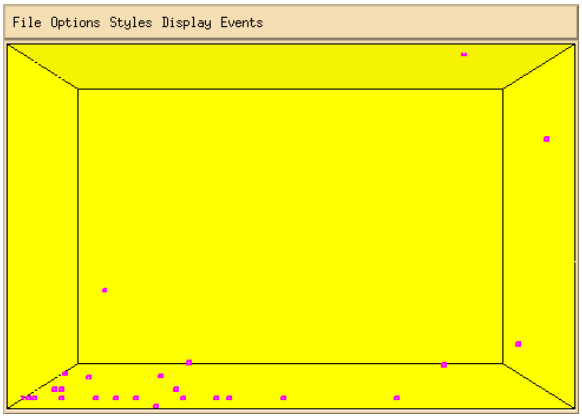

FIGURE 3. Scatter plot display of run time vs. length vs. static entries for a call graph

rank in the 3D matrix and uses these computed positions to assign actual positions to the various components. The layout heuristic is also responsible for finding pivot points for routing the arcs. Additional properties of the layout allow the setting of the amount of white space between rows or columns, whether all elements of a given row, column, or rank should be the same size or not, whether objects should be centered within their relative block or not, and the amount of space to leave around the outside of the layout. A variety of different heuristics are provided. These are discussed in section 5. Figure 1 contains several examples of layout objects. The main component of each of the file boxes is a 2-D layout object whose components are the local arcs and the data objects for the routines in that file. The whole call graph is drawn using a 3D layout object. The components here are the group arcs and the tilings that represent the files.

A second set of abstract data objects have been built to handle visualizations that seemed useful but that were not easily implemented using the above primitives. Some of these reflect 3D visualizations that have no 2-D equivalent, others reflect PLUM's attempt to be more inclusive in the coverage of potential visualizations. These include:

• **Tagged Objects**. These consist of a tag object, generally a label, connected to another object, the contents, using a hinge-like mechanism. They reflect a generalization of a visualization proposed by Wen [23]. The hinge can have a size, can be attached either to the right or below the tag object, and can be set at any angle, settings derived from properties of the tagged object. An additional property shrinks the contents object to the size of the tag. Tagged objects are used in 3D displays to make the tag visible to the viewer from the front while hiding temporarily the information associated with the tag. This information can be seen by causing the contents to be rotated forwards or by flying around in 3D space. Figure 2 shows a call graph where tagged objects are used to represent the files. In this case there is a small gap between the tag and the layout object and the layout object is rotated at a 45 degree angle.

• **Scatter Plots**. These consist of a rectilinear region with components placed based on three values. Scatter plot components can be arbitrary objects. The scatter plot object computes the range of values in each dimension and finds the associated location for each component. It then places the components accordingly. The properties associated with the scatter plot control the basic size of the region and the size of the components. Figure 3 shows a scatter plot display of a call graph that provides information about the routines. The X dimension reflects the amount of run time spent in the routine (as determined by profiling), the Y dimension represents the length in lines of the routine, and the Z dimension represents the number of other routines that call the routine.

• **Markers**. These correspond to simple markers. They are a simplified form of data object designed specifically for use in scatter plots. The tags representing routines in figure 3 are marker objects.

• **Time Sequences**. One of the uses of 3D is to show the history of execution (or any other time-based property of programs) in the Z dimension. Time sequence objects were one way that we have developed for doing this. These consist of a base component which identifies an X-Y position for each object and a set of elements. Each element consists of an object, a from and to time, and a reference to an object in the base component. The element is drawn using the X and Y position and size of the referenced base component and using the from and to times to set the position and size in the Z dimension. Properties of time sequence objects specify the relative size for a given unit of time and the direction of time. Figure 4 shows a top-down view of a dynamic call graph display using a time sequence object. The layout object at the front (bottom) is a 2-D representation of the call graph. The boxes behind (above) this object represent dynamic instances of the elements of the call graph. Here time flows backwards (from bottom to top) in Z.

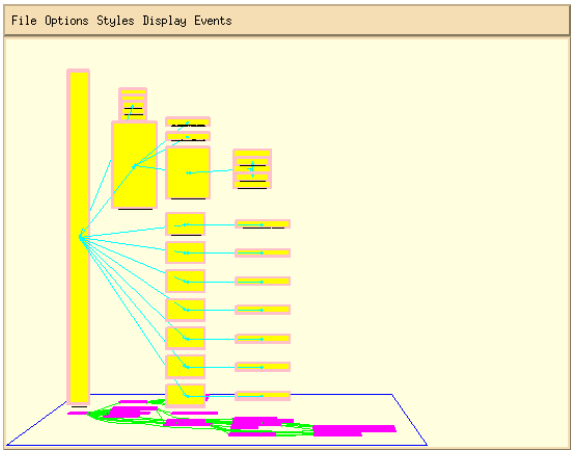

FIGURE 4. Dynamic call graph display using a time sequence object

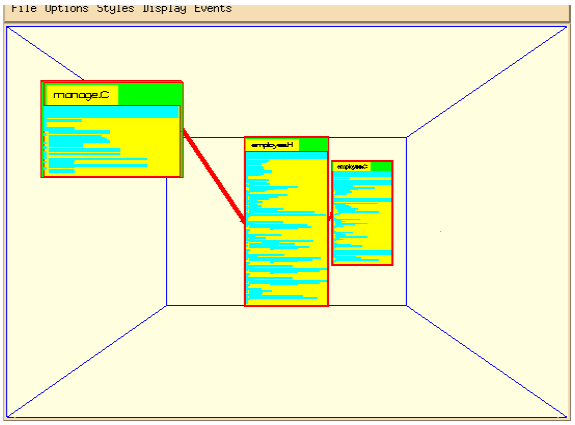

FIGURE 5. Call graph display showing file objects

• **File Objects**. The central focus for programming is the source code and many visualizations relate directly to files. Based on ideas developed at Bell Labs, a file object presents an abstract view of a file that can be augmented with additional information to reflect properties associated with regions in the file. A file is represented as a planar region where each line of the file corresponds to one line in the region. If region is too long, it is divided into columns and the columns placed side by side. The lines are used to represent information about the corresponding portion of the file. Each line can be drawn full width or can be drawn to reflect the indentation and line length of the corresponding text. The latter mode allows identification of program structures, block comments, etc. and allows a programmer who is familiar with the code to associate the representation with the text. Each line can also be associated with a height and a color value, allowing both color and depth to be used to convey information about that line in the graphical representation. Properties associated with file objects allow the setting the file name, the width of each line, how the line should be drawn, the width of the border around the lines, the number of lines in a column, and the mapping from data values to depth and color. Figure 5 shows file objects being used in a call graph display.

The current implementation of PLUM is still expanding. In addition to these generic objects, we are planning to implement objects that reflect other three-dimensional representations that have been proposed. These include:

• **Cone Trees**. This 3D representation for a tree was developed at Xerox [20]. Here the children of a node are arrange in a planar circle underneath their parent. Selecting a child node causes it to rotate to the front. Our cone tree objects will allow the children to extend either to the right (cam trees) or below, and will support arbitrary objects as the root and children components. Additional properties will be used to control the size of the children.

• **Perspective Objects**. These objects are reflections of the perspective wall that was also developed at Xerox [11]. We have experimented with generalized perspective-based transforms (for example using a flattened hemisphere) as well as fish-eye transformations and plan to make some implementation of these available as an abstract graphical object that is effectively a specialized type of layout.

• **Generalized Arrays**. These objects reflect the need to display array-like structures in three dimensions. While this can be done using tilings, a generalized array object will provide a simpler and more consistent interface. In a standard array each component occupies the space for one unit in each relevant dimension. A generalized array allows each component to specify its coordinates in each dimension so that a single components can span multiple units. This will be useful, for example, in putting up a display of blocks in memory where the Y dimension reflects the page of memory the block occurs in, the X dimension reflects the location within that page, and the Z dimension reflects the lifetime of the block.

4.0 Support for abstract graphics objects

PLUM is an object-oriented package organized around the abstract graphics objects. There are three base classes that represent these objects, *PlumObject* to represent the graphics objects themselves, *PlumComponent* to represent the components, and *PlumConstraint* to represent the constraints. Subtypes of PlumObject represent the different type of abstract graphics objects, for example *PlumDataObject* represents a data object. Subtypes of *PlumComponent* represent the different types of components. Each component object contains the *PlumObject* for that component as well as the data that is component specific. For example, a tiled component object contains the extent of the corresponding tile.

The interface between PLUM and an application is based on these three classes. The application instantiates new *PlumObject*s to represent the relevant abstract graphics objects. Methods on these objects are then used to relate these by creating *PlumComponent* and *PlumConstraint* objects. Other methods support the setting and

retrieval of property values. There are also methods to support the layout and drawing of each object type and to support animation.

Converting the abstract graphics objects into a display is done in three stages. The first stage computes the desired size of each object. This is done by making a bottom-up pass over the tree of graphics objects. Primitive objects, such as data objects, compute the size they need directly from their property values. Composite objects determine their size using the previously computed sizes of their components. For tiled objects this means setting up and solving the system of constraints that reflects the tiling. For layout objects this means computing the relative position of each component.

The second step involves layout. This is done using a top-down pass over the tree of graphics objects. Here each composite object sets the position and size of its components relative to itself. Sizing is done by allowing the component to draw itself at its desired size and then scaling the result to fit into the desired space.

The final step in the conversion process is drawing the objects. Each graphics object is responsible for creating a portion of a graphics display list. Primitive objects generate the appropriate display commands to draw their shape. Composite objects generate display commands for the display glue that holds their components together and invoke the drawing method for their components where appropriate. The display lists are handled by a separate package, PDLG, that is designed to provide graphics system independence, i.e. it is possible to implement our display lists on top of PEX, XGL on the Suns, or SGI's GL graphics library.

PDLG also handles interaction with the resultant display. 3D graphical displays are useless unless the user is free to move around to get a better sense of depth and to view the result from different perspectives. Moreover, PLUM displays are designed to be interactive. PDLG provides two flavors of interaction. First, it directly provides syntactic interactions such as panning, zooming, flying around the display, and changing from orthogonal to perspective views. Second, it provides the facilities needed by a higher-level tool, either a graphical browser, a graphical query language, or a graphical editor, to allow the user to directly manipulate the display. This includes event feedback as well as correlating the mouse position with objects in the display.

The illusion of 3D on a two-dimensional display is enhanced by movement. This is done in part by the syntactic interactors that are part of PDLG. Animation provides additional motion. Animation also serves as another channel of information in the display, highlighting changes by drawing the user's attention to movement. We felt that PLUM should provide the option of animating the structured displays of program data. At the same time, we felt that it would be difficult and would discourage the use of animation if the animation had to be programmed by the application. Thus we have attempted to integrate automatic animation into PLUM.

Animation in PLUM is done unless it is explicitly turned off by the application. The application first generates the initial display. An update to this display can be defined by changing properties of the objects, by adding or removing components or constraints, or by providing a complete new set of graphics objects. PLUM's support for animation is independent of the method used.

After the application changes the display structure, it tells PLUM that the display should be updated. At this point PLUM compares the new display structure with the old, matching as much of it as possible and identifying the differences. For each object that appeared in the old display, it saves the old display information. Then the new display is drawn using a sequence of frames where each frame represents some fraction of the way between the original and the new display. It is the responsibility of the various graphics objects to draw themselves appropriately for the frame. However, PLUM provides generic support for these objects. It will automatically compute the proper size and position for each object for the frame. Moreover, it will generate style properties, such as colors, that are appropriate for the frame. The result, when combined with double buffered graphics support, is a smooth, automatically generated transition from the original display to the updated display.

5.0 Layout methods

Graph layout has been extensively studied in two dimensions [6]. The problem is one of placing nodes and arcs to produce an aesthetically pleasing graph. This is generally translated into more specific problems such as reducing arc length and the number of crossings or of emphasizing symmetry. While we provide a variety of approaches in our 2-D layout packages, the algorithm of choice for program data has been one based on level graphs [21] since it tends to emphasize hierarchy and since it generally produces a reasonable looking result.

Moving graph layout algorithms from two to three dimensions is not trivial. The first problem is determining what "looks good" in three dimensions. Because 3D graphics imply that the user is going to move around and look at the graph from different perspectives, assumptions based on the user's view may not be valid. For example, the heuristic of minimizing crossings is meaningless. Given any two arcs in three space that do not intersect, we can find a perspective where they do not cross and a second perspective where they do cross. Since most arcs will not physically intersect in three space, the number of crossings will vary with the perspective.

A second problem is that three space offers many more degrees of freedom. In two dimensions there are two alternatives to laying out a level graph, representing the levels as either rows or columns, and the resulting graphs are identical except for orientation. In three dimensions one has three alternatives for how to represent levels. Moreover, once the leveling is done, each level can be potentially represented by a plane and hence by an arbitrary 2-D

layout. One could, for example, apply a 2-D level graph algorithm to the remaining nodes, i.e. do leveling twice. Alternatively, the algorithm could place the nodes in a circle as in cone trees.

A third problem that arises is that we want to use the third dimension to convey information and not just to provide more space for layout. This means that we have to find layout methods that reflect properties of the underlying objects. For example, layout methods must be able to assign a Z coordinate to a node based on its accumulated run time or what file its in or how distant it is from a set of selected nodes that the user is focusing on.

In PLUM we have implemented a flexible approach to 3D layout to experiment with different algorithms and to gain experience with what works and what does not. Our approach allows layout methods to work in various ways. Some methods, such as leveling, work for one dimension and depend on another layout method to handle the remaining dimensions. Other layout methods are comprehensive, working in all three dimensions at once. Still others, such as local optimization, don't compute the layout in any dimension, but instead modify a layout that is already present. All the layout methods allow values to be defined by the application or by the user. Each coordinate can be given a default relative or a default absolute value. Relative values identify the location in the array that is used by layout objects. These are typically used to represent program assigned values. Absolute values can be used to exactly reflect user manipulations of the underlying objects. All the layout methods are also parameterized with properties. The initial set of 3D layout methods includes:

- **Depth first layout**. This comprehensive (all dimensions) approach is quite simple. It does a depth first search through the graph, visiting each node once. As each node is visited it is placed as close to its parent as possible. Properties here determine whether arcs are considered directional for the depth first search, whether the graph should be laid out in two or three dimensions, and what layout directions are preferred, i.e. down and then to the right.
- **Breadth first layout**. This is similar to depth first layout except that a breadth first search is used in place of a depth first search.
- **Averaged layout**. This is a slightly more sophisticated version of the above. The nodes are looked at in order, either the order they were defined in or a depth or breadth first search order. When a node is considered, the locations of all nodes connected to it that have been previously placed are averaged together to get a target location. Then the new node is placed as close as possible to this target location. The properties of this layout strategy determine the search order, specify whether the layout should be two or three dimensional, and determine the preferred direction for the layout.
- **Level graph layout**. This layout method handles a single dimension. It computes a leveling of all the nodes that do not have that dimension previously defined and assigns a value in that dimension based on the leveling. Once the leveling is done, a secondary layout method, specified as a property, is applied to handle the remaining dimensions. Properties specify the dimension to be used and whether arcs should be considered directional or not. Other properties control the type of leveling. Normally leveling starts at the root node and assigns each subsequent node a level that is one greater than any of the nodes it is connected to. Bottom-up leveling starts with the leaf nodes. For both of these the level of a node is the maximum of the levels of its predecessors. In breadth-first leveling, the level is assigned on the first visit to the node and not changed. A final property determines whether level heuristics should be applied to the arcs through the insertion of pivot points for the arcs for each level that the arc traverses.
- **Level ranking layout**. This method handles one dimension and assumes that some prior dimension was handled by a level graph layout. It attempts to order the nodes within a level by considering their position relative to the nodes above and below it in the leveling. This is done by making multiple passes over the leveling, alternating top-down with bottom-up, and assigning the positions within each level to minimize arc length. The properties of level ranking layout specify the dimension to work on, the dimension of the previous leveling, the layout method to apply next for further dimensions, and the number of passes that should be made over the graph.
- **Unique value layout**. This method handles one dimension, assigning a unique value to each node in that dimension using a modified depth-first search algorithm. This is useful, for example, to assign a unique Z position for each file grouping in a call graph layout. The properties here set the dimension to be used, the subsequent layout method, and the first value to be used in the dimension.
- **Local optimization**. This is a post-processing approach that takes a complete layout and applies an optimization algorithm to it. The optimization approach is to assume a linear attractive force between connected nodes and an inverse square repulsive force between nodes that are not connected. Then a relaxation algorithm is employed to find the resultant positions of the nodes. Properties here specify the method that provides the initial layout, the value of the attractive and repulsive forces, and the number of passes to be made in the relaxation algorithm.

6.0 Experience with PLUM

We have been using PLUM as a basis for 3D call graph visualization using a prototype browser and program information provided by FIELD. The system gathers information about the static call graph from the FIELD cross reference database. It augments this with information about global variables from the database, dynamic call information from the debugger, and profiling information from the profiling interface tool.

The prototype browser provides two basic functions. First, it provides some semantic interactions with the gen-

erated picture. It allows the user to collapse and expand nodes within the hierarchy, to exclude nodes from the display, and to select nodes of primary interest. This provides a simple test of PLUM's support for higher level interaction. Second, the browser is a prototype facility for experimenting with different visualizations. It provides a hard coded mapping from objects representing the gathered information into PLUM graphics objects. This mapping is controlled by a set of twenty-eight user settable parameters to allow full experimentation with the capabilities of PLUM. Figures 1 through 5 were all generated using this browser with various option settings.

PLUM has been designed to handle relatively large displays with thousands of objects. Most of the internal algorithms are linear or at worst *n log n*. The layout methods have all been chosen for speed. GELO, its 2D predecessor, has been successfully used to display graphs of this magnitude. While our current experience with PLUM has been limited to visualizations with only several hundred nodes, we don't foresee any problems in scaling up. Note that our abstraction display methodology emphasizes the use of hierarchy and user selection in creating the display and, based on our experiences with GELO, we expect that most useful displays will contain fewer than a thousand objects.

Our experiences have demonstrated that PLUM is a flexible package that is capable of providing 3D program visualizations. It has provided a solid foundation for experimenting with different visualizations. It has encouraged us to develop new visualizations. For example, tagged layouts were implemented using the PLUM facilities in under six hundred lines of code while scatter plots were implemented in under four hundred lines. It has given us a good sense of what is required for 3D layout and provided a base for experimenting with different layout algorithms. It has demonstrated that automatic animation will be effective in program visualization.

There is a lot left to be done. We have to improve the aesthetics of the generated diagrams. This includes determining how to reduce the amount of blank space between nodes without causing nodes to intersect in perspective, working on new layout algorithms, determining appropriate routing heuristics, making text more readable, and making more effective use of color, shape, and texture. A second area for future work involves the specification and generation of additional 3D presentation strategies. We have outlined some in section 3. Additional methods should be developed that use spatial cues more effectively and that can provide two presentations simultaneously, one in the X-Y plane and one in the (X,Y)-Z plane. Finally, once PLUM is relatively robust, we plan to use it as a basis for determining the empirical value of 3D visualizations of program data.

7.0 References

1. David B. Baskerville, "Graphic presentation of data structures in the DBX debugger," UC Berkeley UCB/CSD 86/260 (1985).

2. Marc H. Brown and Robert Sedgewick, "Techniques for algorithm animation," *IEEE Software* Vol. 2(1) pp. 28-39 (1985).

3. Marc H. Brown and John Hershberger, "Color and sound in algorithm animation," *Computer* Vol. 25(12) pp. 52-63 (December 1992).

4. Marc H. Brown and Marc A. Nojork, "Algorithm animation using 3D interactive graphics," DEC Systems Research Center (1992).

5. Jacques Davy, "GoPATH programmer's guide," Bull Imaging and Office Solutions (December 1992).

6. P. Eades and R. Tamassia, "Algorithms for automatic graph drawing: an annotated bibliography," *Networks*, (1993).

7. Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner, Jr., "Seesoft -- a tool of visualizing software," AT&T Bell Laboratories (1991).

8. Belinda B. Flynn and David Maier, "Specification and generation of displays for complex database objects," Oregon Graduate Institute of Science and Technology (1992).

9. Sadahiro Isoda, Takao Shimonmura, and Yuji Ono, "VIPS: A visual debugger," *IEEE Software* Vol. 4(3) pp. 8-19 (May 1987).

10. Mark A. Linton and John M. Vlissides, "Unidraw: A framework for building domain-specific graphical editors," *Proc. UIST '89*, pp. 158-167 (November 1989).

11. Jock D. Mackinlay, George G. Robertson, and Stuart K. Card, "The perspective wall: Detail and context smoothly integrated," *Proc. CHI'91*, pp. 173-179 (April 1991).

12. Brad A Myers, "Incense: a system for displaying data structures," *Computer Graphics* Vol. 17(3) pp. 115-125 (July 1983).

13. Brad A. Myers, Dario A. Guise, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal, "Garnet: Comprehensive support for graphical, highly interactive user interfaces," *IEEE Computer*, pp. 71-85 (November 1990).

14. Michael L. Powell and Mark A. Linton, "Visual abstraction in an interactive programming environment," *SIGPLAN Notices* Vol. 18(6) pp. 14-21 (June 1983).

15. Steven P. Reiss and Joseph N. Pato, "Displaying program and data structures," *Proc 20th Hawaii Intl Conf System Sciences*, (January 1987).

16. Steven P. Reiss, "Working in the Garden environment for conceptual programming," *IEEE Software* Vol. 4(6) pp. 16-27 (November 1987).

17. Steven P. Reiss, Scott Meyers, and Carolyn Duby, "Using GELO to visualize software systems," *Proc. UIST '89*, pp. 149-157 (November 1989).

18. Steven P. Reiss, "Connecting tools using message passing in the FIELD environment," *IEEE Software* Vol. 7(4) pp. 57-67 (July 1990).

19. Steven P. Reiss and Manojit Sarkar, "Generating program abstractions using an object-oriented database," Brown University Department of Computer Science (1992).

20. George G. Robertson, Jock D. Mackinlay, and Stuart K. Card, "Cone trees: Animated 3D visualizations of hierarchical information," *Proc. CHI'91*, pp. 189-194 (April 1991).

21. L. A. Rowe, M. Davis, E. Messinger, C. Meyer, C. Spirakis, and A. Tuan, "A browser for directed graphs," *Software Practice and Experience* Vol. 17(1) pp. 61-76 (1987).

22. John T. Stasko, "TANGO: A framework and system for algorithm animation," *IEEE Computer* Vol. 23(9) pp. 27- 39 (September 1990).

23. James Wen, "A three dimensional browser for visualizing orthogonal hierarchies," Brown University (1992).