Program Visualization in a Virtual Environment

Michitaka Hirose, Testuro Ogi, and Michel Riesterer
Department of Mechano-Informatics, The University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan
E-mail : {hirose,ogi,michel}@ihl.t.u-tokyo.ac.jp
Tel. : +81 3 3812 2111 (ext. 6367)
Fax : +81 3 3818 0835

Abstract

The content of this paper describes the work carried out in the authors' laboratory to create a concept demonstrator of program visualization in a virtual environment. Understanding and maintaining programs is a difficult task, even when the programs have been well designed and engineered, and providing the user with a graphical representation may be useful because it uses the instinctive skills of humans in recognizing and comprehending graphical information and patterns by visualization. However, the presentation of the information must be recognizable enough to allow the user to concentrate his or her cognitive skills for the purpose of the comprehension of the program, not of the representation itself.

The approach described in this paper is to maintain a representation of the system within a virtual environment. The positioning of the components within the three-dimensional space is determined by behavior attached to the components which, typically, leads to a spatial organization that reflects the semantic structure of the system. Furthermore, different levels of abstraction in the visualization allow the user to understand the system from the overall structure to the more specific details of the implementation.

1. Introduction

As software programs become larger and larger, understanding, debugging and maintaining them becomes harder and harder, even when the programs have been well designed and engineered. Complex systems (e.g. large object oriented programs) are usually composed of a large number of components, and each component usually has some relationship with several other system components. The complexity of the system can make it very difficult to determine its overall structure and the interactions which occur between the components.

In the case of large object-oriented programs, the user typically has the fundamental tasks of constructing, debugging and maintaining the system. Often, the user has to build his own cognitive model of the system, and to perform this task, he extracts information from the documentation (if it is available) and from the source code in order to gradually put together the pieces of this "software puzzle". In the search for useful information scattered through the code, the user has to continuously shift focus to different sections of code, and these tasks place a significant load upon the user.

Often the visualization process must take place within the programmer's head, and often overwhelms the ability of the programmer to deal with complexity. Advanced, high-quality interactive graphics can serve as visualization tools that can aid better comprehension of the program by reducing the cognitive burden on the programmer's short-term memory. To provide the user with high-quality stereoscopic graphics as well as a sense of immersion, we decided to use an immersive display similar to CAVE [Cruz93].

1.1. Benefits of a visual representation

For obtaining a high-level understanding of code, graphical representations are more useful than purely textual representations. Now, programs are considered simply as strings of alphanumeric text stored in disk files. Even if it is clear that the structure of a complex program is embedded within its code, the simple representation of code as a text file is not suitable for understanding the overall structure of the system which is often very hard to untangle from the details of the implementation. This applies both when building and debugging a system and, more significantly, when attempting to maintain an existing system.

In fact, the human brain seems more suited to processing visual patterns, with processing text being only a particular case (if we consider each word or character as a visual entity). So in textual representations the visual information presented to the user is too numerous (each character count for one entity), and consequently the cognitive model created is too complicated. The aim of presenting to viewers a symbolic visualization of lower level information (e.g. the abstraction of all the code of a function in one cubic object) is to allow them to generate a simple initial mental model they can use for further investigation according to their particular needs.

1.2. Why a 3-dimensional representation?

The use of three-dimensional, high-resolution color graphics in scientific simulation makes a radical improvement in the sense of presence achieved. A similarly enhanced sense of presence for the understanding and debugging process seen as simulation should be a considerable source of power.

The layout of objects can also gain from the use of a three-dimensional representation. Many existing program visualization tools display information on a two-dimensional canvas [Price93], calculating the position of objects by using one of the many layout algorithms available. Unfortunately, the result is often a bird's nest graph which is very difficult to understand because of the great number of crossing lines. Even if the use of a three-dimensional representation does not solve all the problems at once, it offers some advantages. One of them is that the additional dimension allows a bigger working space to position and organize the information (or in other words "more room to move"). Another advantage is the possibility for the user to choose the focus of this attention and the level of detail required by his position and orientation in space.

Some pioneering work has been done in the field of information and database visualization [Fairchild88], and gradually representing the structure or behavior of programs in a three-dimensional space has become a more popular method, one example being the visualization of parallel programs [Koike95][PVM95].

After explaining how the graph is generated and how the spatial layout is performed, we will take a look at some strategies for managing the complexity of the information presented to the user by reducing the amount of information shown and by organizing it in different levels of abstraction. Finally, we will describe the system used to visualize the model.

2. Graph generation and objects organization

Until now, many program visualization tools have been developed (mainly research prototypes) and most of them rely on user interaction for generating the actual visual representation or for selecting what should be represented in each file (for example by instrumenting the source

code). The prototype we are discussing in this paper tries to provide an improved level of automation by parsing the source and generating a visual representation.

2.1. Graph generation

Given a set of C/C++ source files, the program automatically generates a file containing all the relevant information extracted from those files:

- a list of the functions defined in each file;
- for each function, the list of calls to other functions; and
- data structures.

This file is then read by another program which builds a graph from all the data (with several levels of sub-graphs) and runs a simulator in order to try to reach a state in which the position of the different entities reflects the global structure of the parsed source files (figure 1).

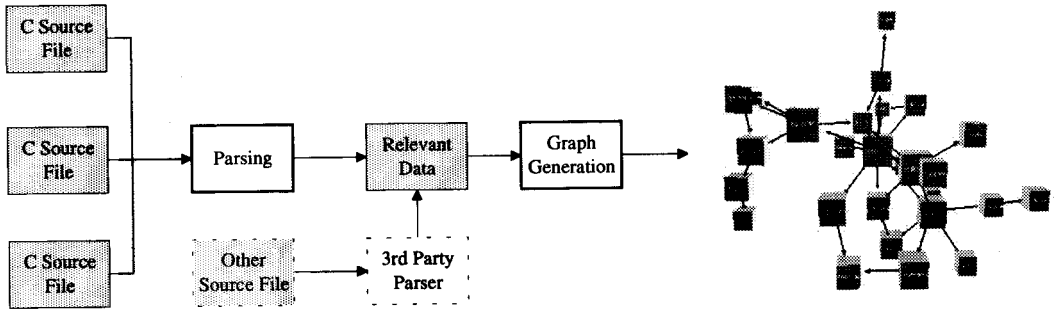

Figure 1: Processing of source files

There are mainly two advantages of generating an intermediate file containing the relevant data in the first step:

- it obviates the need to process all the files each time one wants to consult the graphical model of the source files, and possibly saves time if there is a large volume of files to parse. Furthermore, it is possible to save the current state of the analyzed set of files by saving only this intermediate file.
- the user can examine the results of the parsing and eventually modify this file to tune it for his needs, or write his own parser generating the same kind of output so as to be able to visualize other languages not supported by the provided parser.

2.2. Positioning of objects in space

After extracting the pertinent data from the code, the next step is to achieve a spatial organization of the different entities that tries to reach the following three objectives:

- to show the structure of the data as clearly as possible;
- not to put too big a load on the user by asking him to place all the objects;
- to be simple enough to allow the user to gain a quick comprehension of the displayed graph.

This leads to a fundamental question about representing objects in a three-dimensional space (or even in a two-dimensional space), i.e., the question of how to organize the spatial layout of objects within the space in an efficient way. One way of organizing objects in space is to assign a special meaning to each axis. For example in [Hirose93], one axis is used to display the timing in which the different tasks are executed, while the other two are used to display inter-tasks structural information.

Actually we use the technique of self-organizing graphs or in other words a FDP (force-directed placement) algorithm. Although conventional layout techniques require a routine which attempts to perform a suitable layout on a given graph while trying to satisfy a number of criteria (aesthetic or other), self-organizing graphs allow the graph itself to perform the layout by modeling it as an initially unstable physical system and allowing the system to reach a stable equilibrium [Hendley95].

Actually, the behavior of the objects within the space is loosely modeled on physical systems. Each object behaves according a set of very simple rules:

- by default, each object exerts a repulsive force on every other object. This force is inversely proportional to the distance between the objects; and
- a relationship between two objects causes an attractive force to be exerted between these two objects.

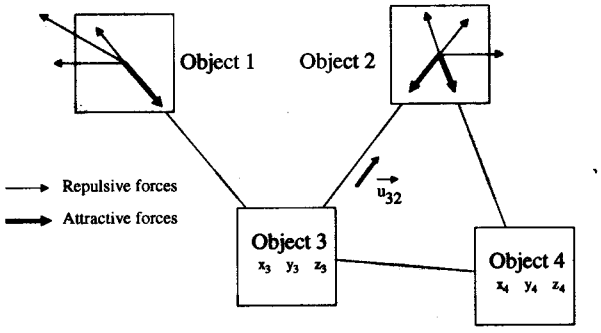

Figure 2: Forces exerted between objects

If we call $\vec{F}_i$ the force exerted on object i, then we have:

$$\vec{F}_i = \sum_{\substack{\text{all objects} \\ j \neq i}} -\frac{K}{Dist(i,j)}\vec{u}_{ij} + \sum_{\substack{j \\ j \text{ in relation with } i}} K' Dist(i,j)\vec{u}_{ij}$$

where K and K' are two constants, $Dist(i,j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}$ and $\vec{u}_{ij}$ is the unitary vector from object i to object j (see figure 2).

Initially, objects are given a random position within a small space creating a state of high energy (repulsive forces are strong because the objects are confined in a small space) as in figure 3. By running a simulator, objects are moving so that they are gradually organized in a way which reflects the relationships between them. By doing this, the underlying structure can become apparent from the spatial layout of the objects (figure 4). This set of rules, although very simple, is quite efficient, and the system's spatial layout usually approaches a state of equilibrium relatively quickly (within a few hundred iterations).

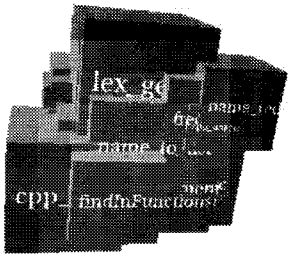

Figure 3: Initial state

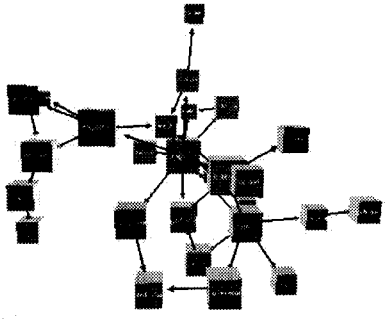

Figure 4: Equilibrium

When this state of equilibrium is reached, the global appearance of the graph in the three-dimensional space tends to represent the global structure of the system, but neither the length of a particular link nor a given object's position convey a special meaning.

3. Management of complexity

Programs are pure abstract entities and are often composed of a large number of components with complex relationships between them. Because programs have neither intuitive appearance nor physical form, the complexity which arises often places them beyond the cognitive ability of a single person. Inherent to this complexity and the amount of information to deal with, the program visualization tools have to find a compromise between displaying too much information (such a representation reflects the complex nature of software but is quite difficult to apprehend and use efficiently) and showing too simple information (that is easy to understand but not very helpful to work with). Therefore it is necessary to reduce the complexity and only show the user the information he needs. There are several ways of achieving this:

- reduce the number of components to show to the user by selecting the relevant information while parsing the source code;
- create several levels of abstraction allowing the user to choose between a global view and a more detailed view depending on the task he has to perform; and
- act on the appearance of objects by changing their color or geometric aspect.

3.1. Reduction of the number of components

The first strategy to reduce the amount of information shown to the user is to reduce the number of objects in the graph. By objects in the graph, we mean nodes as well as links. A first and simple thing to do is to display only one link between two nodes even in the case of multiple calls between functions (figure 5). This reduces the global number of links, but also reduces locally the number of links between two nodes, producing a graph that is much easier to read.

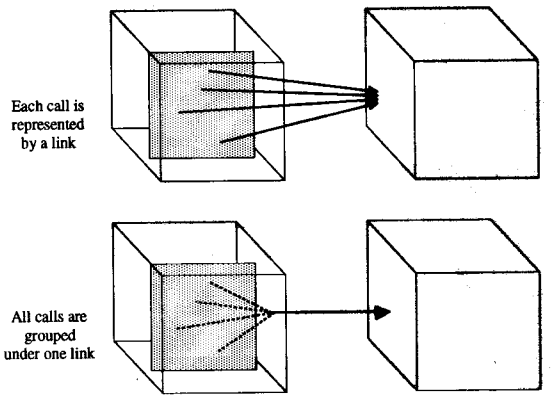

Figure 5: Reducing the number of links

The second strategy applied was to display only the functions defined in the set of sources parsed. That means that all the calls to standard libraries or third party libraries (like for example X11, Xm, or other GUI libraries), which are usually numerous but not really essential for the understanding of the program structure (e.g. the `printf` function in the standard library) are not represented. That does not mean that the information is not available to user, but

to keep the graph smaller, this information is displayed by using other methods based on the appearance of objects (see 3.3).

This choice to hide all "external" functions seemed appropriate because we assumed that usually the user tries initially to gain a global comprehension of the set of source files he is working on, and thus to initially overload him with all the information about these calls is not necessary.

3.2. Levels of abstraction

It is useful to be able to apprehend a program at different levels of detail, from the relations between the logical modules composing the system to the specificity of a particular implementation, because users do not always need the same level of understanding a program, and because it is easier to build a mental model of the system starting with a global view and then to refine it gradually.

To allow the user to gain a global understanding of the system as well as enabling him to get more details if necessary, objects are grouped into one distinctive compound object when they are not a focus of attention. By doing this, the global structure becomes more apparent because there are fewer objects displayed at the same time, and the user can see and manipulate relevant details while maintaining a view of the overall context. This approach of nested graphs allows the abstraction of information from complex graphs by clarifying their global structure while enabling the details to be viewed when necessary. However, at present, the only method implemented to allow the user to select more or less detail is based on the user's distance from the objects. This can be seen as similar to the approach used in a number of programming environments and visual languages, where ellipsis is used to hide unnecessary detail, with the difference that in the case of our tool details show up automatically when the user gets closer to the objects.

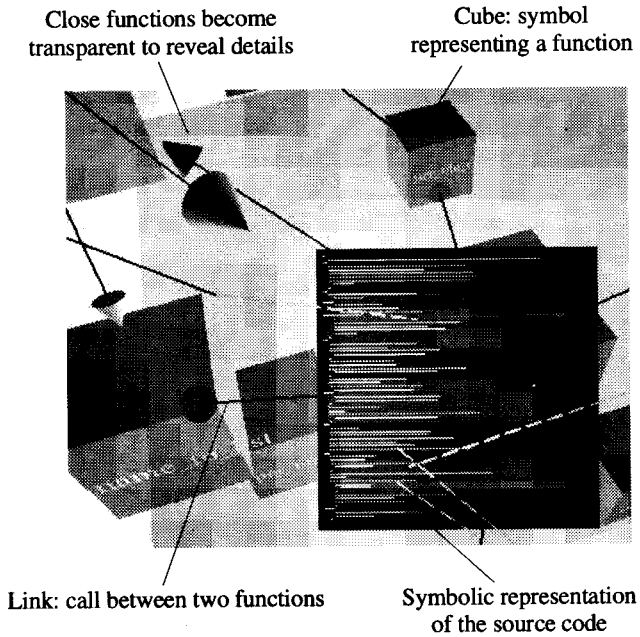

Figure 6: Two levels of abstraction (function level and source code level)

Actually, the way of revealing details is based on transparency: the representation changes depending on the user's distance from the objects (or groups of objects), in that the closer you get to the object, the more transparent the object becomes, revealing the sub-structure inside, if the case arises (figure 6).

Showing too many details can be confusing for the user (and can also slow-down the visualization), so the sub-structure is displayed only when the user (or more precisely when the user's viewpoint in the virtual environment) is really close to an object. The transparency is very progressive, and when one object occupies about one fourth of the screen, its sub-structure can be seen clearly. So usually at the same time there are only one or two objects really transparent and a few others more or less transparent which sub-structure can be guessed on one screen.

3.3. Appearance of objects

The appearance of the graph is very important to help the understanding process. Apart from the layout algorithm, a number of other ways can be used to enhance the readability of the graph. Two possibilities to add information to the graph without adding more objects are to use different colors or to change the geometric aspect of objects

3.3.1. Use of color

Usually, programs contain a kind of hierarchy (at least a starting point) and sometimes a layout based on it can reveal the logical train of nodes' processing more clearly. Due to the use of a kind of simulation to place the objects in our prototype, it is difficult to predict the position of the objects when the equilibrium is reached or to constrain the objects to adopt a top-down layout reflecting their hierarchical position. Consequently, when the user explores the model for the first time it could be difficult to find the object(s) to begin with. Therefore to assist the user in understanding the hierarchical structure of the program, the use of color can be an effective approach. The idea is to modulate the intensity of the color according to the hierarchical position of the objects. For example in C/C++, the starting point is the `main` function. By giving a brighter color to it and to the functions called directly by it, the user can find where to begin more easily.

Another possible use of color is to display information related to the use of third parties libraries. Color-coding can provide easily perceivable cues for classifying objects in a small number of categories (like the use of such or such kind of library). Objects making calls to a given library can share a common color so it is easier to locate them. For example, if a special color is assigned to all the objects making calls to the X11 library or any other GUI library, the user can see more quickly where the objects dealing with the user-interface part of the program are in the graph.

Finally, the use of color can also be an effective way of highlighting or clustering objects. For example when the user wants to know where a given variable or structure is used in the program, highlighting the objects using it by changing their color allows him to locate them quite easily and to possibly see them as a group sharing some property (the use of the variable), but without moving them, i.e., without obliging the user to build a new cognitive model because of changes in the objects' positions.

3.3.2. Geometric appearance

With regard to the use of the objects' geometric appearance to code some information, there are mainly two possibilities: to change the size of objects or to change the shape of objects. Firstly, it is possible to change the size of the object according to the amount of information it contains. In our prototype, the size of an object representing a function depends on its number of lines : the object is scaled by a factor equal to the logarithm of this number of lines, so that the size of objects is not to disproportional while different enough to be distinguished, and moreover, there

is lower-limit to prevent objects representing small functions to become quite invisible among bigger ones. For an object representing a whole file, the size is computed to allow enough space inside it to contain the functions defined in it.

Another way to reflect a particular feature of the code is to use a particular shape for an object. As program code is an abstract concept, there is no inherent shape associated with objects like functions or variables. Currently we are representing the function as cubic shapes and the classes (for C++) as more rounded shapes such as spheres. In fact, symbolic shapes representing the different entities can be chosen freely. However, these shapes should be different enough to avoid confusion in identifying the type of object, and be simple enough so that they can be rendered in real-time.

We are still examining the use of different combinations of shapes and colors to determine what kind of properties or attributes are worth representing and what are the best graphical ways to do this.

4. Visualization of the model

Once the graph is generated and the objects positioned we need to be able to visualize the model with a display that allows the user to get as good comprehension as possible. To do this, we use an immersive display similar to CAVE.

4.1. Visualization system

An explanation of the hardware structure of the prototype we are using now is shown in figure 7. The prototype has been built using C++ on Silicon Graphics Infinite Reality stations, and the graphical display is performed using Performer 2.0 that provides real-time visualization ability.

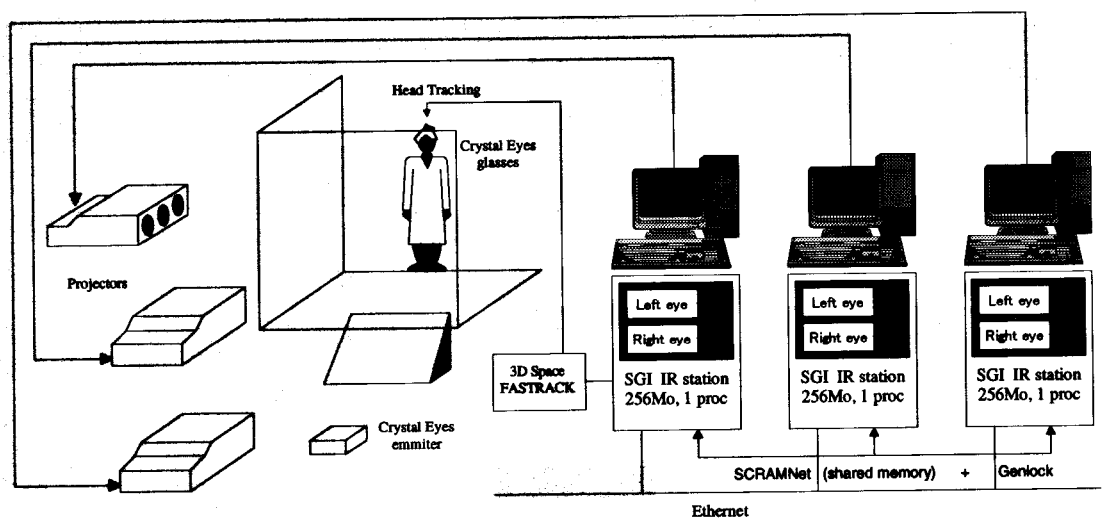

Figure 7: Hardware architecture

The final version of this immersive display will be composed of five big screens (left, right, front, top, bottom), but as shown in figure 7, we are now working with a prototype composed of only three screens that are a little smaller than those of the final version. A computer generated picture of the five-screens immersive display is shown in figure 8.

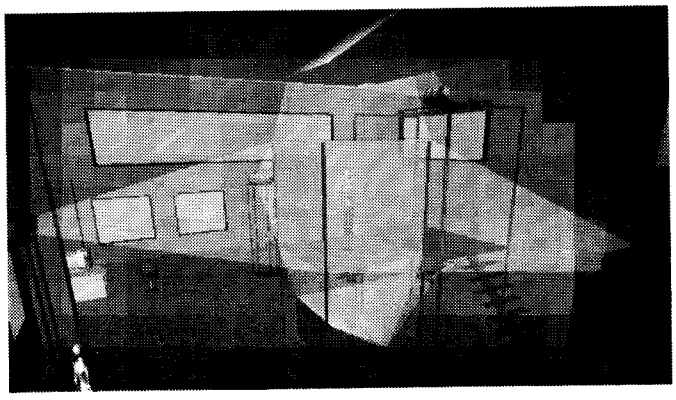

Figure 8: A computer model of the final immersive display

Having recourse to five screens displaying perspective stereoscopic views gives the user the impression that he or she is *in* the simulation and thus allows a quicker understanding of the model on which they are working.

4.2. Navigation and interaction

The use of a perspective stereoscopic view coupled with the possibility of a real-time motion of the model improves the level of comprehension of three-dimensional graphs [Ware94]. Actually, the ability to examine the model from different points of view seems to be a key factor to aid understanding. In fact, the kind of graph generated to represent data extracted from a program source code has no intrinsic dimensionality but it seems that our brain is more efficient when dealing with three-dimensional representations [Purcell91].

If the ability to navigate freely to examine the model is important, the kind of motion used to achieve this goal does not seem so important. So we give the user the choice of using the model of motion he thinks to be the more appropriate for him. Currently only two models are available:

- one is a flying model: the user changes his point of view as if he was flying through the data space in an aircraft;
- the other is an "examiner" model: the user can rotate the view around his actual point of focus as if he was holding the model in his hand and rotating it by moving his hand.

Apart from the possibility of real-time motion to examine the graph, the interaction is still rudimentary, this prototype can be considered more as a viewer than as a development tool since there is no way to modify interactively the code (yet). The user can choose a function in the list of all the functions defined in the program and have his position changed so that he can see the selected function. There is also a selection feature to find where given variables or structures are used, the highlighting being done by changing the color of the relevant objects.

5. Discussion and conclusion

The prototype has a good level of automation, from the parsing of the code to the automatic organization of objects in space according to their relationships using a FDP algorithm. This automation in the placement of objects is valuable because it does not require the user to create the actual representation himself. However, it has a drawback: when a new component or link is added, part of the graph (or the whole graph) can change greatly, obliging the user to rebuild a new cognitive model of the program. This problem can be partially solved: because we are using a random function which can give the same set of numbers each time the program is

executed, for a given set of data, the produced graph is always the same; changes occur only when an object or link is added or deleted. We are still examining a number of ways to improve this by, for example, restricting the area where the changes occur to the inserted object and its neighborhood, or allowing a manual placement for a few new inserted objects.

In this paper, the authors have described a prototype of program visualization tool that uses a virtual environment to maintain a representation of the program's structure model. The use of a good level of automation and the use of an immersive display allows quicker comprehension of the structure of programs. This is a work still in progress, but from the beginning the main problem of this kind of representation was clear: how to generate a meaningful abstraction which at the same time can hide the underlying complexity and be simple to understand and effective to use.

There is still a lot of work to be done, and in the future the authors will work on the following points:

- better integration with software development environments to modifications, because this tool is still a kind of viewer only, aimed to aid better comprehension of program structure;
- augmentation of the quantity of information displayed at the same by using other metaphors or by improving the use of colors;
- better use of the potential of the virtual reality such as improving the interaction with the visual representation or adding sound technology to give a broader range of interface channels.

6. Literature

[Cruz93] **Cruz-Neira, C., Sandin, D.J., and DeFanti, T.A.**, *Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE*, Computer Graphics Proceedings, Annual Conference Series, p.135—142, 1993

[Price93] **Price, B.A., Baecker, R.M., and Small, I.S.**, *A Principled Taxonomy of Software Visualization*, Journal of Visual Languages and Computing **4** (3), p.211—266, 1993, http://hcrl.open.ac.uk/jvlc/JVLC-Body.html

[Fairchild88] **Fairchild, K.M., Poltrock, S.E., and Furnas, G.W.**, *SEMNET: Three-Dimensional Graphic Representations of Large Knowledge Bases*, in Cognitive Science and Its Applications for Human Computer Interaction, R. Guindon, ed., Lawrence Erlbaum, Hillsdale, N.J., p. 201—233, 1988

[Koike95] *VisuaLinda : 3D Visualisation of Parallel Linda Programs*, **Koike Labs**, University of Electro-Communications in Tokyo, 1995, http://www.vogue.is.uec.ac.jp/vlinda.html

[PVM95] *The PVM Trace Project*, University of New Brunswisck, 1995, http://www.omg.unb.ca/hci/projects/hci-pvmtrace.html

[Hirose93] **Amari, H., Nagumo, T., Okada, M., Hirose, M., Ishii, T.**, A Virtual Reality Application for Software Visualization, Proceedings of VRAIS'93 Conference

[Hendley95] **Hendley, B., and Drew, N.**, Visualisation of complex systems, Proceedings of the HCI'95 Conference, http://www.cs.bham.ac.uk/~nsd/Research/Papers/HCI95/hci95.html

[Ware94] **Ware, C., and Franck, G.**, *Viewing a Graph in a Virtual Reality Display is Three Times as Good as a 2D Diagram*, IEEE Conference on Visual Languages, Conference Proceedings, p. 182—183, October 1994

[Purcell91] **Purcell, D.G. and Stewart, A.L.**, *The Object Detection Effect: Configuration Enhances Perception*, Perception and Psychophysics, 50(3), p. 215—224, 1991