Program Visualization in a Virtual Environment

Michitaka Hirose and Michel Riesterer

Department of Mechano-Informatics, The University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan
E-mail : {hirose,michel}@ihl.t.u-tokyo.ac.jp
Tel. : +81 3 38 12 21 11 (ext. 6367)
Fax : +81 3 38 18 08 35

Keywords: Virtual Reality, Program Visualization, Information Visualization, Computer Graphics, Calls Graph.

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 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 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 that, 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 of the other components of the system. The complexity of the system can make it very difficult to determine the overall structure of the system and the interactions which are present between the components.

In the case of large object-oriented programs, the user typically has fundamental tasks of constructing, debugging and maintaining the system. These tasks require the user to be able to navigate through the system and to interpret the relationships between the system and its behavior. 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. Providing support for the user's understanding of the structure and behavior of the system will inevitably lead to improved reliability and efficiency of these processes.

1.1 Benefits of a visual representation

For obtaining a high level understanding of code, graphical representations are more useful than purely textual representations. 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, and consequently the cognitive model created is too complicated. The aim of presenting to viewers a symbolic visualization of lower level information 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 ?

Many existing program visualization tools display information on a two-dimensional canvas [1], 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. 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 [2], 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 [3][4].

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.

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 automatically 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 interesting information extracted from those files:

- list of the functions defined in each file;
- for each function, the list of calls to others 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.

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 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 these 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 [5], 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 [6].

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.

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 1. 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 2). 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).

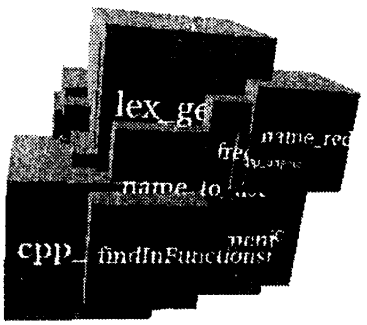

Figure 1: Initial state

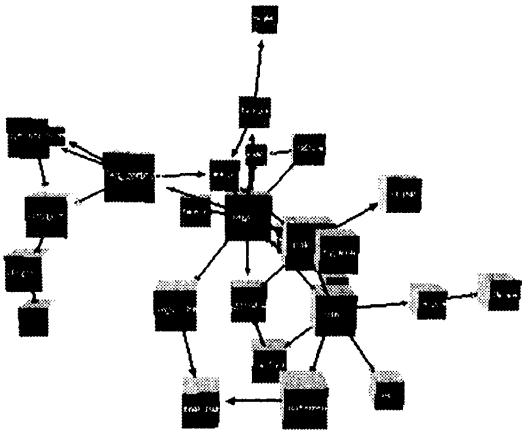

Figure 2: Equilibrium

2.3 Drawback of the self-organizing graphs

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, a part of the graph (or the whole graph) can change greatly, obliging the user to rebuild a new cognitive model of the program. We are still examining a number of ways to improve this by, for example, restricting the area there the changes occur to the inserted object and its neighborhood, or allowing a manual placement for a few new inserted objects.

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; and
- 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.

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. 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.

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 3rd 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.

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 at first overloading him with all the informations 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 of 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.

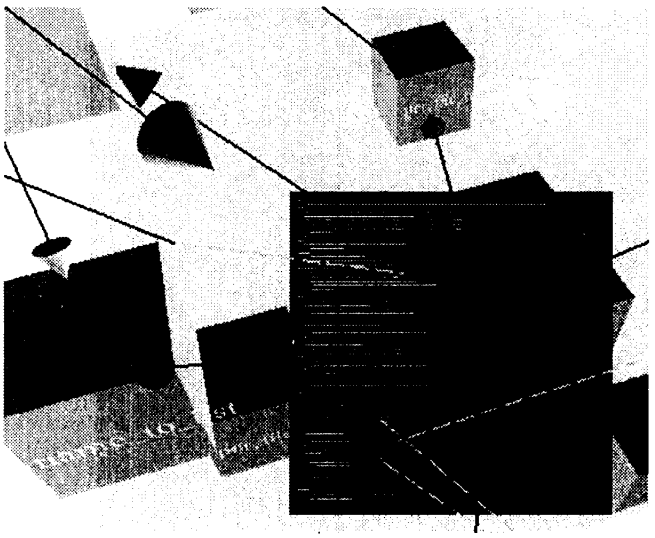

Figure 3: Two levels of abstraction (function level and source code level)

Actually, the way of revealing details is based on transparency: the representation is changing depending on the user's distance from the objects (or groups of

objects), the closer you get to the object, the more transparent the object becomes, revealing the sub-structure inside, if the case arises (figure 3).

4 Conclusion and future work

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 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 FPD algorithm.

This is still a work 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. In spite of the complexity inherent in the abstract nature of programs, a representation based on a graph reflecting the relationships of the components of the system along with several levels of abstraction seems to be an effective approach to understanding the global structure of a program while maintaining the possibility of accessing the details needed to refine the comprehension.

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 the software development environments;
- augmentation of the quantity of information displayed at the same by using different shapes for the objects or making better use of colors;
- a 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.

References

[1] **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
http://hcrl.open.ac.uk/jvlc/JVLC-Body.html

[2] **K.M. Fairchild, S.E. Poltrock, and G.W. Furnas**, *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., 1988, pp. 201–233

[3] *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

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

[5] **Haruo Amari, Toshiki Nagumo, Mikio Okada, Michitaka Hirose, Takemochi Ishii**, *A Virtual Reality Application for Software Visualization*, Proceedings of the VRAIS'93 Conference

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