Another Look at Program Visualization and Algorithm Animation *(Extended Abstract)*

Ricardo A. Baeza-Yates

Depto. de Ciencias de la Computación
Universidad de Chile
Casilla 2777, Santiago, Chile *

Abstract

We present the first results of yet another system that allows the end-user to graphically visualize and animate a program. We emphasize the differences between our approach and previous works on the topic. This system can be seen as a preprocessor that automatically annotates a C program with graphical function calls, and embeds the program in a special environment. The system is implemented in C and uses the X-Windows environment, making it highly portable. It can be used for several purposes, including debugging, teaching, understanding, and exploring algorithms and data structures.

1 Introduction

Visualization of a dynamic process makes facilitates understanding, and has a bearing on research and teaching. Interest in this topic is growing rapidly as can be seen in many special issues or articles in important journals: **IEEE Computer** (Aug, 1985, on *Visual Computing*; May 88, on *Algorithm Animation*; Aug, 1989, on *Scientific Visualization*; and Oct, 1989, on *Visualization in Computing*), **IEEE Trans. on Software Engineering** (Oct, 1990, on *Visual Programming*), and **Communications of the ACM** (Feb, 88, on *Visualizing Scientific Data*). Specific conferences and journals have also appeared: **Journal of Visual Languages and Computing** (published by Addison-Wesley), and **IEEE Workshop on Visual Languages**.

We are interested on graphically simulating a program in execution. Although there are some systems that perform this task, we have taken a somewhat different approach. We have used the best features of known systems and combined them with some new ideas. It is difficult to describe all concepts involved in the resulting system, and in this paper we only stress the main aspects of it.

Program visualization has been used in many different contexts. Among them we can mention teaching [BMD83, NVH86], debugging [LSV+89, ISO87, Mye81], concurrent programming

*This work was supported by grant 91-0868 of CONICYT.

[LSV+89], algorithm design [Bro88b, BK87], software development [BCH+85], and performance evaluation [PN81, LSV+89].

For now, we will call our system NN (a better name will be provided in the future). The main characteristics of NN are:

- Given a C program, NN automatically introduces function calls in the program, embedding it in a graphical simulation environment.
- The user-interface is built on top of X-Windows and provides several windows for different views of the program, including the code itself, the current active function, the tree of function calls, and the data structures.
- For visualizing data, a fixed set of possible views are provided, including a debugging mode and a small number of visual modes, depending on the data types.

The main feature is that an end-user does not need to know any detail about how to animate a program. This allows the system to be used for teaching, where a student can debug, understand, and explore a program without any special knowledge. On the other hand, the visualization will not be as close to the program behavior as in other systems, and the simulation may not reflect some semantic aspects. At this point is necessary to clarify that NN is not AI. In fact, NN does not understand what the program does. It only analyzes every data action, following a set of simple rules to generate the graphical code.

The organization of the paper is as follows : Section 2 describes related work; Section 3 provides an overview of NN and its user interface; Section 4 describes the system architecture; Sections 5 and 6 present the different visual modes of the system; the last section presents its current and future development.

2 Related Work

Several graphical program simulation tools have been developed/ All of them attach *hookpoints* to the program. These hookpoints can be *code intrusive* (CI) or *data intrusive* (DI) [IIWF90], and are generated *manually* (M) or *automatically* (A). Code intrusive hookpoints are special function calls inserted in the original program code. Data intrusive hookpoints are used in object oriented languages by subclassing existing object classes and providing new methods which will drive the animation. We use the above concepts to classify several systems. For example, NN is a CI-A[1] system.

Perhaps the best known system is BALSA (from Brown University) [MR84, BS85], and its successor, BALSA-II [Bro88a, Bro88b]. While BALSA is incredibly powerful, it is not trivial to define a new visualization, and the hookpoints must be included manually. In fact, BALSA-II is a CI-M type system. The learning effort for either system is initially higher than for other systems [Bro88b]. BALSA has been used extensively, with a high level of success, for teaching algorithms in a laboratory where the students can see a visualization developed by the instructor [BS84].

Another interesting system was developed at Tektronix for Smalltalk programs [LD85, Dui86, Dui87]. One of its advantages is that the animation code is embedded in the data by using subclasses

[1] Any association is an unavoidable and unfortunate coincidence.

of the program classes. Thus, this is a DI-M system. The main drawbacks are that updating multiple views is done in a sequential way, which does not allow composite views. Furthermore, some kind of interactions are not possible.

Incense is another system [Mye81] which permits visualization of data structures interactively. Its main goals are debugging, tracing and program documentation. Tango [Sta90] is a recent system similar to BALSA which is based in a special framework. This system requires manual annotation, and a new visualization can be designed in, at most, a few days. Other systems [HHR89, Rei83] are described and compared in [Sta90, Bro88a].

The University of Washington Illustrating Compiler (UWPI) is the closest to the system presented here. In [HWF90] the term "program illustration" is used instead of "program visualization". UWPI is another CI-A system, and supports a subset of Pascal that is similar to RATFOR. It does not support types, records, or pointers. Also, UWPI is not an interactive system.

3 System Overview

The main goals behind the use of NN are:

- **Understanding**: the user comprehends faster what the program does. Thus, it is easier to teach data structures and algorithms, either in a special lab or by personal assignments.
- **Debugging**: "one picture is worth a thousand words". Looking at a bug visually, it is easier to see where and why it happened. For example, one can readily find a null pointer or recognize an array index out of range. In a traditional Unix environment these kind of run time errors give awkward messages such as "segmentation violation", "bus error", etc.
- **Exploring**: visual animation of an algorithm allows to one gain insight into how it performs. This encourages improvements in known algorithms.

The NN users interface tries to fulfill the above goals. The system includes the following views of the program, each of them in a window:

1. **Current activated function**, showing its name, arguments, and local variables. An example is shown in Figure 1. The top bar shows the current line number.
2. **Current tree of program calls** (see Figure 1). The current function is shown as a black node. The arguments given to each called function can be obtained by pointing to the associated node.
3. **Program code**. Figure 2 shows the Quicksort algorithm used in all our examples (taken from [GR91]). The current line is shown in inverse video.
4. **Data views**. Complex data structures, such as arrays and linked structures, are shown, each in a separate window. Their functionality depends on the data type (to be explained later).

All views support scrolling, resizing, and other typical window operations.

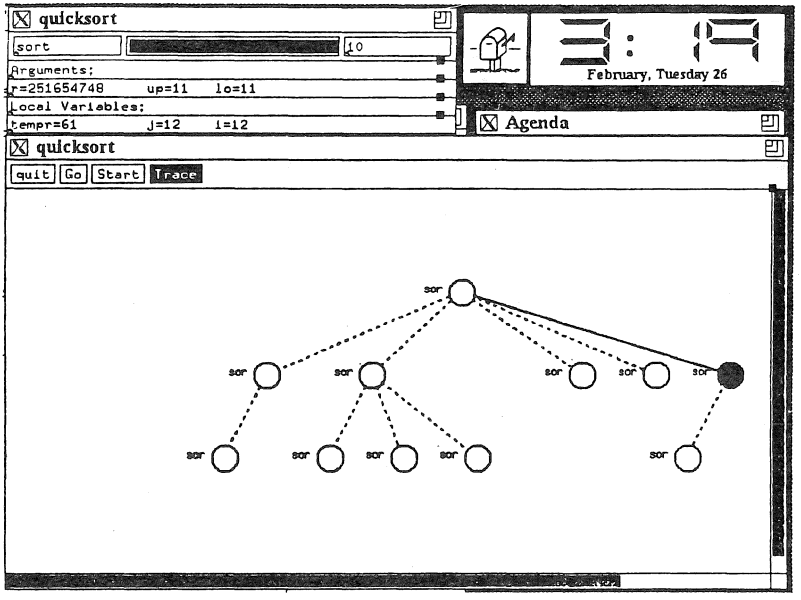

Figure 1: System interface.

```
sort( r, lo, up )
int r[];
int lo, up;

{int i, j, tempr;
while ( up>lo ) {
      i = lo;
      j = up;
      tempr = r[lo];
      /*** Split file in two ***/
      while ( i<j ) {
            for ( ; r[j] > tempr; j-- );
            for ( r[i]=r[j]; i<j && r[i]<=tempr; i++ );
            r[j] = r[i];
            }
      r[i] = tempr;
      /*** Sort recursively, the smallest first ***/
      if ( i-lo < up-i ) { sort(r,lo,i-1);  lo = i+1; }
            else     { sort(r,i+1,up);  up = i-1; }
      }
}
```

Figure 2: Example code: Quicksort

The interface allows the user to:

- run, stop, restart, or quit the animation.
- set a continuous or discrete animation of the program. Every discrete step is defined as a visual event.
- change view parameters, such as the speed of a continuous animation, the size of graphical elements, etc.

4 System Architecture

The system has two different components: the generator of animation code and the graphical library. The generator is run between the standard C preprocessor and the C compiler. The annotated program is compiled and linked with the special animation library. Only the library is dependent of the windowing system, which is X-Windows [SGN88]. The complete process is shown in Figure 3.

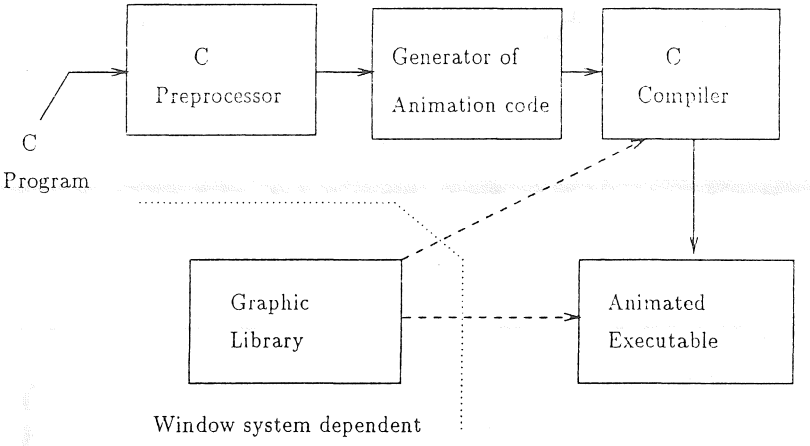

Figure 3: Encapsulation Process.

The current version of the code generator assumes "good programming style". For example, it does not support a goto from outside to inside a loop. Because static binding does not follow all possible variable references in a program, the preprocessor creates a dynamic table where the variable addresses are stored. This table is used at run time to determine to which and where the graphical event should be associated. The generator supports the full C language, although the quality of the generated code is inversely proportional to the program complexity.

The basic events detected by the generator are assignation, comparisons, indexing, and function calls. The generator is built using the standard tools for compilers available in Unix.

The second component is the graphical library, which is the interface between the animation code and the windowing system. The library is implemented using Xlib [SGN88] and the Xt toolkit [You89]. Some of the capabilities of this library are detailed in the following sections.

5 Debugging Mode

One of the main data views is the debugging mode. In this mode the user sees every detail of the data. An example, running the partitioning step of Quicksort, is shown in Figure 4. In this example the position of several indices in the array are shown. An index out of range is shown to the left or to the right of the array, depending on its value. A comparison is shown by a double border and a connecting arrow. If every element of the array is a record, the first element of it is shown. However, the user may change this at run time. The other values can be obtained by pointing at each cell.

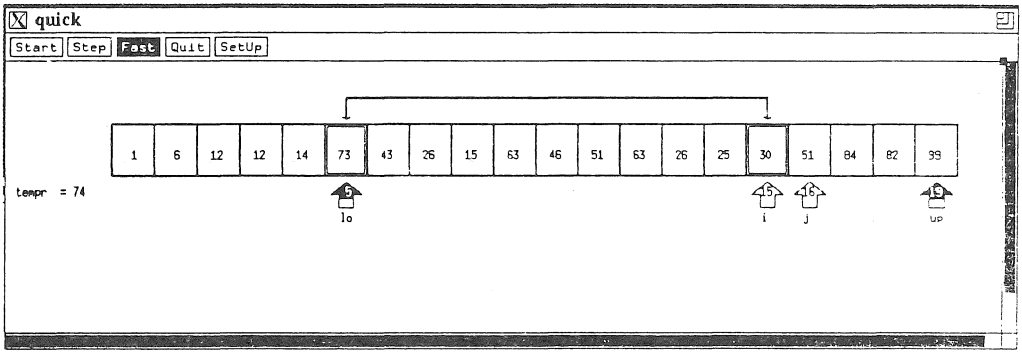

Figure 4: Debugging mode.

It is possible to mix the debugging mode with gray levels. This is shown in Figure 5.

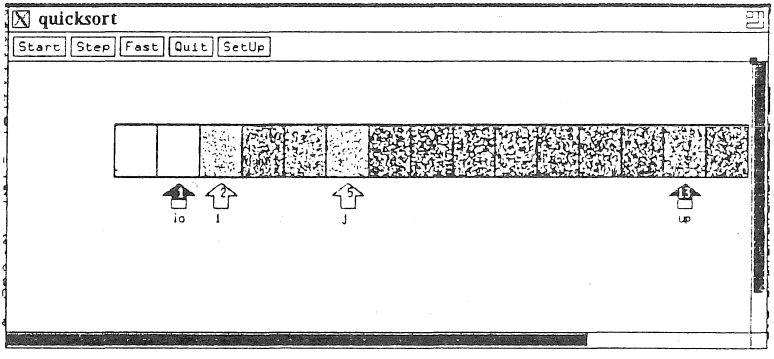

Figure 5: Mixed mode.

6 Visual Mode

Once the program appears to be correct, we need a high level view of its behavior. This allows better understanding of it. An example is shown in Figure 6, which shows the relative sizes of the elements and not their absolute values. We can clearly follow how the sorting is done, and we can have a larger number of elements visible in the window. Currently, two visual modes for arrays are supported. The second one is shown in Figure 7.

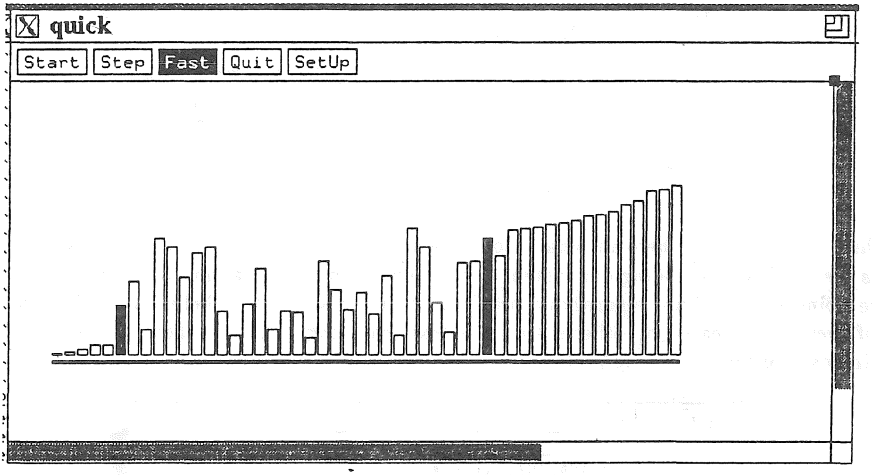

Figure 6: Visual mode.

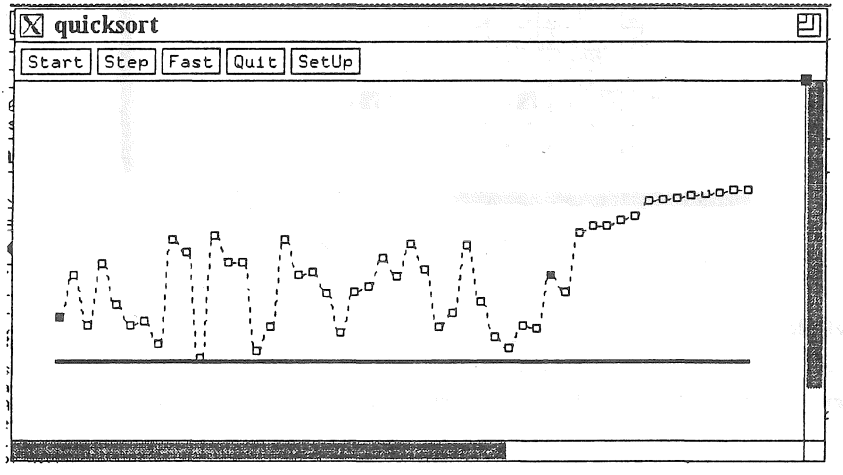

Figure 7: Another visual mode.

7 Future Work

Ongoing work on this system includes drawing of dynamic structures such as linked lists, trees, and graphs. A taste of this is given in Figure 8. On the graphical side, we want to increase the number of available views. This will include color support, better graphical design and other improvements.

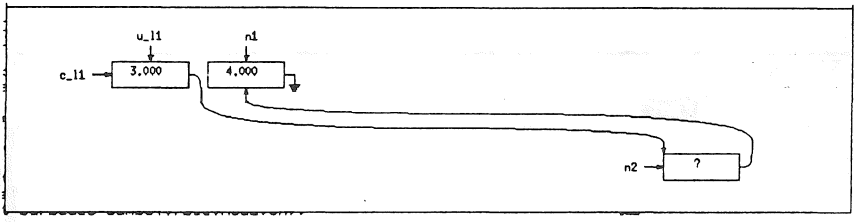

Figure 8: Handling lists.

There are problems where automatic code generation does not suffice. BALSA does a great job in this case. We are extending our system to also support manually annotated code. In particular, we are doing this for hard algorithmic problems, where visualization helps at the research level. One of these, not treated by other systems, is text searching. Figure 9 shows the animation of a two-dimensional text searching problem.

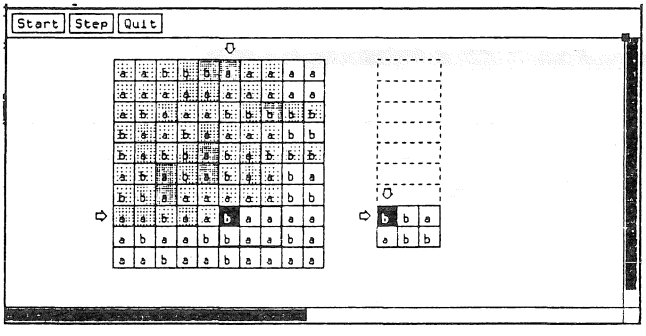

Figure 9: Text searching.

Acknowledgements

I would like to thank the people involved in the implementation of the ideas presented in this paper: Waldo Aguilar, Ivan Diaz, Edson Diaz, Luis Fuentes, Luis Jara and Gaston Quezada.

References

[BCH+85] G.P. Brown, R.T. Carling, C.F. Herot, D.A. Kramlich, and P. Souza. Program visualization: Graphical support for software development. *Computer*, pages 27–35, Aug 1985.

[BK87] J. L. Bentley and B. W. Kernighan. A systems for algorithm animation: Tutorial and user manual. Technical Report 132, AT&T Bell Labs, Murray Hill, N.J., Jan 1987.

[BMD83] Marc H. Brown, Norman Meyrowitz, and Andries Van Dam. Personal computer networks and graphical animation: Rationale and practice for education. *ACM SIGCSE Bulletin*, 15(1):296–307, Feb 1983.

[Bro88a] M. H. Brown. *Algorithm Animation.* MIT Press, 1988.

[Bro88b] M.H. Brown. Exploring algorithms using Balsa-II. *IEEE Computer*, 21(5):14–36, May 1988.

[BS84] Marc H. Brown and Robert Sedgewick. Progress report: Brown university instructional computing laboratory. *ACM SIGCSE Bulletin*, 16(1):91–101, Feb 1984.

[BS85] M. H. Brown and R. Sedgewick. Techniques for algorithm animation. *IEEE Software*, pages 28–39, Jan 1985.

[Dui86] R. A. Duisberg. Animated graphical interfaces using temporal constraints. In *Proc. ACM SIGCHI 86 Conf. Human Factors in Computing Systems*, pages 131 – 136, April 1986.

[Dui87] R. A. Duisberg. Visual programming of program visualizations. In *Proc. Conf. Visual Languages*, Aug. 1987.

[GR91] G.H. Gonnet and Baeza-Yates R. *Handbook of Algorithms and Data Structures - In Pascal and C.* Addison-Wesley, Wokingham, UK, 1991. (second edition).

[HHR89] A. Helttula, A. Hyrskykari, and K. J. Raiha. Graphical specification of algorithm animations with aladdin. In *Proc. 22nd Int'l Conf. System Sciences*, pages 892–901, 1989.

[HWF90] R. R. Henry, K.M. Whaley, and B. Forstall. The university of washington illustrating compiler. In *Proc. ACM SIGPLAN'90*, pages 223–233, White Plains, NY, June 1990.

[ISO87] S. Isoda, T. Shimomura, and Y. Ono. Vips: A visual debugger. *IEEE Software*, 4(2):8–19, May 1987.

[LD85] R.A. London and R.A. Duisberg. Animating programs using Smalltalk. *Computer*, pages 61–71, Aug 1985.

[LSV+89] T. Lehr, Z. Segall, D.F. Vrsalovic, E. Caplan, A.L. Chung, and C.E. Fineman. Visualizing performance debugging. *Computer*, 22(10), 1989.

[MR84] M.H.Brown and R.Sedgewick. A system for algorithm animation. *Computer Graphics*, pages 177–186, July 1984.

[Mye81] B.A. Myers. Incense: A system for displaying data structures. *Computer Graphics*, 17:76–93, Nov 1981.

[NVH86] J. Nievergelt, A. Ventura, and H. Hinterberger. *Interactive Computer Programs for Education*. Addison-Wesley, Reading, Mass., 1986.

[PN81] Berhard Plattner and Jurg Nievergelt. Monitoring program execution: A survey. *Computer*, 14:76–93, Nov. 1981.

[Rei83] S.P. Reiss. Pecan: A program development system that supports multiple views. Technical report, Brown University, Providence, RI, 1983.

[SGN88] Robert W. Scheifler, James Gettys, and Ron Newman. *X WINDOW SYSTEM*. Digital Press, 1988.

[Sta90] John T. Stasko. Tango: A framework and system for algorithm animation. *IEEE Computer*, 23(9):27–39, Sep 1990.

[You89] Douglas A. Young. *X Window System Programming and Applications with Xt*. Prentice Hall, Englewood Cliffs, NJ, 1989.