Automatic Data Visualization for Novice Pascal Programmers

Brad A. Myers **Ravinder Chandhok** **Atul Sareen**

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213-3890
(412) 268-2565
macgnome@cs.cmu.edu

ABSTRACT

Previous work has demonstrated that presenting the data structures from programs in a graphical manner can significantly help programmers understand and debug their programs. In most previous systems, however, the graphical displays, called *data visualizations*, had to be laboriously hand created. The Amethyst system, which runs on Apple Macintosh computers, provides attractive and appropriate default displays for data structures. The default displays include the appropriate forms for literals of the simple types inside type-specific shapes, and stacked boxes for records and arrays. In the near future, we plan to develop rules for layout of simple dynamic data structures (like linked lists and binary trees), and simple mechanisms for creating customized displays. The visualizations are integrated into an advanced programming environment which is used to teach programming methodology at the introductory level.

INTRODUCTION

Pascal and most other computer languages allow the programmer to define and use a variety of data types. No existing programming environment, however, provides the programmer with similar flexibility when displaying the data structures for debugging or program documentation. Existing debuggers typically use some linear, textual view for the display of any user defined data structure. These linear displays are sufficient for simple types, but grow unwieldy as the data structures become more complicated. This paper describes a new system, called Amethyst, that automatically creates graphical displays for data structures for Pascal programs. Amethyst is part of a MacGNOME [1] programming environment which is used in introductory computer science courses at several universities and high schools. Amethyst stands for A MacGNOME Environment That Helps You See Types.

The pictures that Amethyst creates are similar to the pictures used to explain the data structures in popular textbooks (including the text used by the students[2]). The use of graphical pictures to show program data is called *data visualization* [3]. The particular displays chosen in Amethyst were designed to explain important concepts, such as which dimension of a multi-dimensioned array comes first, and which types are assignment compatible. In addition, a graphic artist participated in the design so the pictures will be visually appealing.

Graphical presentations for data structures are important for a number of reasons. Human information processing is clearly optimized for pictorial information, and pictures make the data easier to understand for the programmer. This will make program debugging and program comprehension easier, because the pictures provide a higher level of abstraction that removes a number of irrelevant details. In particular, some of the programming concepts that students have particular difficulty with can be much better explained with pictures. This includes how Pascal file variables work, the difference between value and **VAR** parameters, and how recursion affects variables. Although it is not one of the goals of the current project, data visualizations would also probably also be helpful for professional programmers since they can be used to abstract out unnecessary details when dealing with large and complex data structures.

ENVIRONMENT

Amethyst is implemented as part of the MacGNOME project. MacGNOME is a family of programming environments designed for novice programmers. It is perhaps the first family of environments designed specifically with novice computer science education in mind. Based on the structure editor generator part of the Gandalf project [4], MacGNOME environments, which are called *GENIEs*, can be generated for any structured language.

GENIEs are unique for three reasons. First, they provide a **structure editor** interface which automatically inserts the appropriate syntax when the user specifies the type of program structure desired*. Second, they provide **multiple views** of the program being edited with the number of views limited only by the imagination of the implementor. Examples currently implemented include an outline view, a

* In addition, GENIEs also allow the insertion of text into the program, via an incremental parser. Pieces of the program can also be edited textually, if desired.

run-time call stack view, two different tree-structured decomposition views [5], and the standard linear program view. Third, a GENIE is an **integrated** environment, which means that all tools use the same structure database as a common interface, and thus appear to the user to be the same tool. For example, the user can set breakpoints in the program by selecting (an editor function) in the program text, and indicating the desire for a breakpoint (a run time system function). This same tight integration is also reflected in the user interface for data visualizations.

OVERVIEW

Previously, although MacGNOME supported some debugging features such as tracing and breakpoints, there was no way for students to view their data structures. The Amethyst project is designed to fill this gap by extending the Pascal GENIE to provide an easy-to-use method for viewing data. Rather than just present a simple textual form for the data, however, it was decided to present a graphical picture of the data structures. This decision was based on many factors, including the high expectations that users have for programs on a graphical computer like the Apple Macintosh. In addition, the teaching staff involved with the project have used various visual and graphical techniques in their teaching for many years (for example [5]). Finally, our own experience as professional computer scientists shows that drawing pictures is often the only way to understand a new data structure.

In the same way that the GENIE removes the burden of syntax from program construction, Amethyst can remove the burden of an awkward debugger interface and allow the teacher and student to concentrate on the material at hand. Indeed, one of the design goals is to choose default displays that are pedagogically sound. The visualizations will correspond to the pictures a teacher might draw on a blackboard in class. In fact, after Amethyst is complete, we hope that most teachers will use a projection of the computer screen, and not draw the pictures by hand.

Since the goal is to provide data visualization for students, it is important that the system be easy to use. Therefore, the student can display a graphical view of the data simply by selecting a variable in the program text and issuing the **Show Value** command from a standard Macintosh menu. The system chooses an appropriate display based on the type of the data object, allocates space in the data view window (or creates a window if one does not yet exist), and displays the current value of the variable (see Figure 1).

Although the default pictures are very helpful and appropriate for many situations, it is also important to provide the ability to create custom displays. Therefore, the internal design of the system uses object-oriented techniques with inheritance which allow users to easily create subclasses of the standard display objects. This can be used, for example, to provide intuitive displays for linked lists, trees, networks, and other regular structures. A future project is to create a graphical editor to help the user design and edit these custom displays.

The internal design of Amethyst also supports multiple, simultaneous views of the same data. This matches well with the GENIE's functionality (there can be more than one view of the program), and can be exploited to illustrate data abstraction. For example, a database program might keep records primarily in a sorted tree, with secondary hash tables for other keys. The two abstractions (or views) of the data could be displayed at the same time, possibly in different windows, which would greatly enhance the student's understanding of the technique.

The entire GENIE system is a working, but still evolving, software base. The Pascal environment has been in use for several years now, and has undergone several revisions per semester. Amethyst is expected to be the final major revision before the system is released as an educational product to the general public. The GENIE itself is written in Object Pascal, a variant of Pascal with object extensions similar to C++. The GENIE also uses a wonderful application building toolkit from Apple Computer called MacApp™ [6].

This paper describes the design and implementation of the Amethyst data visualization part of MacGNOME. Its design and implementation are partially complete, and the first general release of Amethyst is expected in the Fall of 1988.

RELATED WORK

The utility of graphical representations for programs and their data has long been known, and there have been many systems using Visual Programming (creating programs using graphics) and Program Visualization (using graphics to illustrate the code or data of programs). Myers [3], updated in [7], surveys these areas.

The earliest graphical representation for program code was probably the flowchart, and automatic creation of flowcharts dates back at least to 1959 [8]. Since then, there have been many other programs that show program code as some form of flowchart (e.g. FPL [9], PIGS [10], SchemaCode [11], and Pict [12]). Other graphical forms for program code include Petri nets (MOPS-2 [13]), data flow graphs (PROGRAPH [14]), and spreadsheets (Action Graphics [15]).

More relevant to this project are systems that use graphics to illustrate the data of programs. The Incense system [16; 17] was the first system to automatically create displays for data structures. It differs from the current project in that it was designed for professional programmers rather than students, it was implemented in a standalone environment rather than integrated with a real system, and it did not include automatic display management or dynamic animations of changing data.

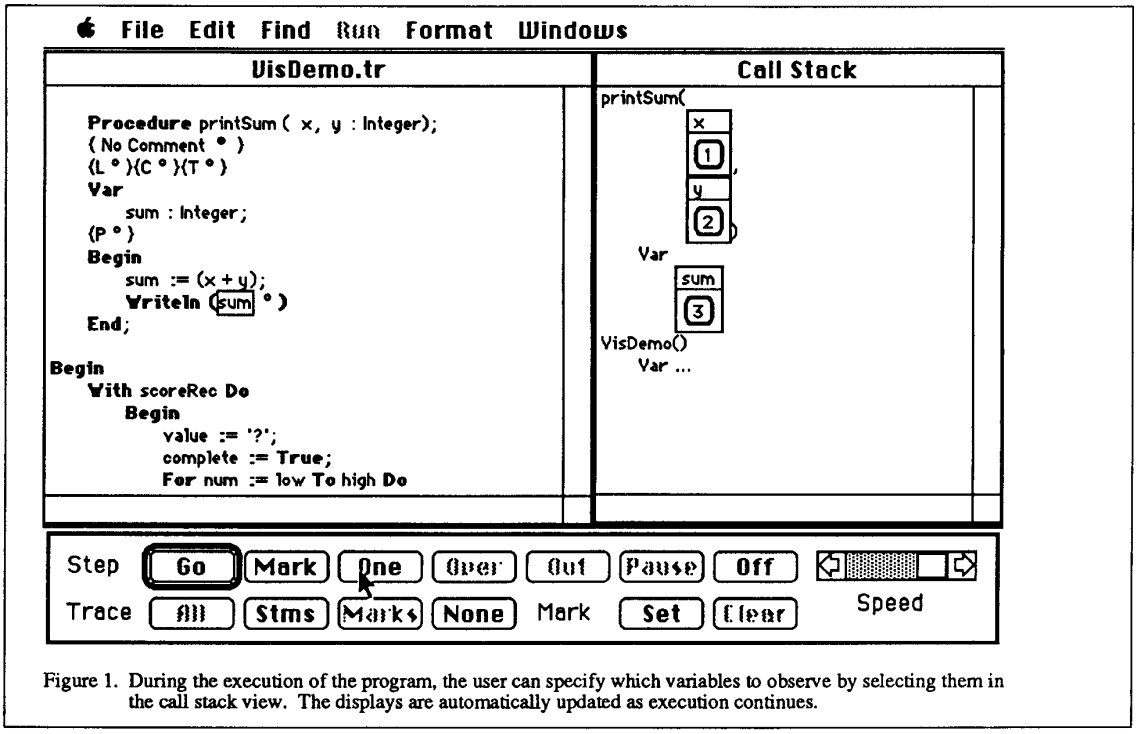

Figure 1. During the execution of the program, the user can specify which variables to observe by selecting them in the call stack view. The displays are automatically updated as execution continues.

One of the first systems to provide dynamic animations of changing data was the Sorting out Sorting movie [18]. The Balsa system [19] provided similar visualizations, and many more, on a computer in real time. The main disadvantage of Balsa was that the visualizations had to be coded by hand. This included designing and implementing the displays by making calls to the underlying graphics package, and inserting probes into the code at appropriate places to update the displays. Balsa has recently been re-implemented on the Macintosh, and is called Balsa-II [20]. The Animation Kit [21], which was written in Smalltalk, provided for smooth animations between states of a program (so that, for example, "lozenges" carrying data values moved smoothly from one variable to another). In order to try to make the specification of the data visualizations easier, the Animation by Demonstration system [22] allowed the programmer to draw examples of the desired format, and then tie these to data objects. The PECAN system integrated some program and data visualizations, although the views of data were mostly textual [23].

DEFAULT DISPLAYS

The primary focus of Amethyst is to provide appropriate static displays of data structures automatically. These displays are updated continuously, so the user never sees an inconsistent view of the data.

In Amethyst, each of the basic types is displayed in a characteristic shaped box (see Figure 2). The shapes are based on the textbook that the students use [2]. Even when inside compound objects, such as arrays and records, elements use these shapes when possible.

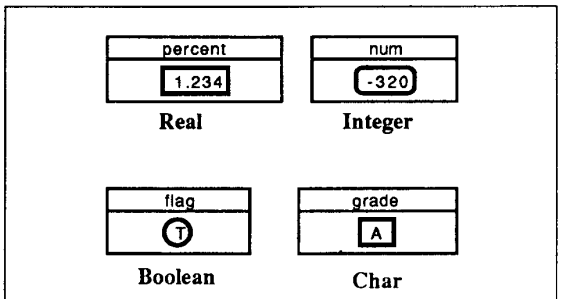

Figure 2. Default displays for each of the simple types

One dimensional arrays are displayed as a conventional row or column (see Figure 3) and two dimensional arrays are shown as a nested boxes (see Figure 4 on page 4).

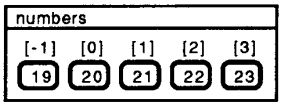

Figure 3. A one dimensional array of integers.

The elements of the array can, of course, be of any type, since the system uses a recursive algorithm to display the

elements. If the array is too big to fit into the window, the conventional Macintosh scrolling mechanisms are used. In the future, other mechanisms for dealing with large data and multi-dimensioned arrays will be developed (these are described in more detail later).

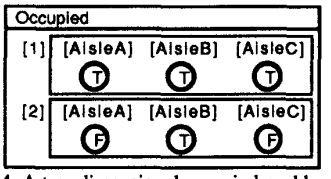

Figure 4. A two dimensional array indexed by an integer and an enumerated type.

The automatic display for records is shown in Figure 5 , which again is based on the standard textbook. Each of the fields can, of course, be of any type.

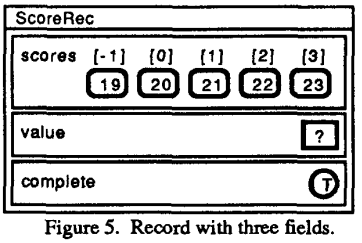

Figure 5. Record with three fields.

The other composite types in Pascal, such as files and sub-program parameters, will be handled in a similar way to arrays and records. The file display will show a portion of the entire file with special marks for the input pointer, the end of lines, and end of file, as shown in Figure 6.

Probably the most interesting (and most difficult) type to display is pointers. This is because the data referred to (the referent) is not graphically enclosed in the pointer object. The present plan is to use a simple two-dimensional space allocation technique which will generate acceptable displays in general, and to use special displays for common pointer structures such as trees and lists.

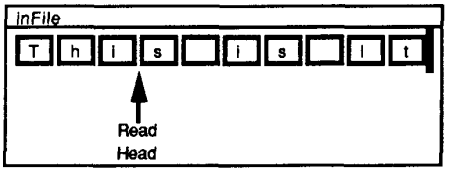

Figure 6. Proposed display for a file of CHAR (TEXT file).

The interaction between the visualization of data on the stack and dynamic data is an area for research, as the execution of the program can cause the graphical relationships to change over time. Figure 7 shows a proposed display for pointers. Note that the "pointed at" value does not have a name*.

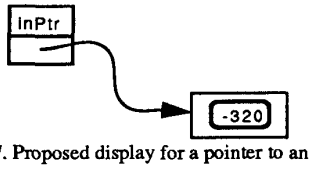

Figure 7. Proposed display for a pointer to an integer.

ARCHITECTURE

The data visualizations are well integrated with the rest of the MacGNOME environment. On the other hand, the implementation is designed to be extendible and separable. This is achieved by using object-oriented techniques and by using a package that insulates the visualization code from details of the particular implementation of the symbol tables and run-time stack.

Type System Package

One goal of the Amethyst implementation is to allow the data visualizations to eventually be used with other, non-MacGNOME programming environments, including ones that may be used by expert programmers. Therefore, a package is under development which separates the data visualization code from all details of the MacGNOME symbol tables, and also from the run time stack and variable allocation mechanisms. Since there is a large amount of commonality among structured languages, it is expected that this interface could be easily extended to also work for C, Ada, Modula-2, and other similar languages. Amethyst is tied to the Macintosh, however, since it uses the standard QuickDraw and MacApp routines to draw the pictures, and uses Apple's Object Pascal Language.

For the symbol table access, there is a different set of routines for each data type. For example, to handle arrays, there are routines to get the index type, the element type, and the low and high bounds of the index. A different set of routines is used to fetch and set the current values of variables. These routines are type-specific, and insure that values come from the correct place and are converted correctly. They also hide any details of bit packing, for example for **Packed Records** or **Packed Arrays**.

* and never can, in Pascal.

Use of Object-Oriented Techniques

When the user requests a visualization of a data structure, a hierarchical display tree is first created. This structure is then processed, formatted, and finally used to draw the visualization itself. Each entity in the display tree is an object, which is an instance of a class. The classes are arranged in a hierarchy, where more specific (lower level) classes inherit behavior from their superclasses. Using these standard object-oriented techniques makes the implementation clean internally, as well as flexible in the sense that an implementor can easily add new subclasses to generate custom displays.

Amethyst consists of two basic types of objects: generators and visObjects (visualization-objects). A generator object provides the mapping function that creates the appropriate type of visObject based on the type of the data to be visualized. Making subclasses of the generator class allows this mapping function to be overridden to provide custom displays. The visObjects are in charge of the actual display of the data values.

Recursive Display Routines for Objects

If a composite type such as an array or record is displayed, then the generator is recursively applied to each subfield until the entire display tree is constructed*. At that point, Amethyst no longer needs any information from the symbol tables; it merely needs access to the actual values of the types.

Drawing the visualization is processed in three phases. These phases, called measure, fit, and draw, are used to allow negotiation for screen space between displays for various types. They function as follows:

Measure - each object is asked to return a preferred and maximum size. For simple types, this merely means measuring the size needed to display the current value, and calculating, if possible, the maximum size that could possibly be needed if the value changed. For aggregate types such as arrays, executing "Measure" requires a summation of all the sizes of the elements.

Fit - after all objects have been "measured", the parent object begins another pass that actually assigns screen space to each object. Again, for simple types there is not much action here, but aggregates can now make layout decisions based on their own allocated space, such as horizontal versus vertical layout for an array. The size specified in the "Fit" call may be different from the size returned by "Measure".

Draw - this causes all of the objects to be displayed.

* When we later begin to handle very large data structures, we may limit the generation to only the parts that are visible on the screen.

This three-phase approach is also used in other parts of the GENIEs [24]. For the data visualizations we have tried, which range up to about 50 objects, the entire display process operates without any appreciable delay (it takes less than 1/2 second to draw a screen-full of data).

After the formatting has been determined, the Amethyst display tree can be easily traversed to draw on the screen. The existence of the tree also makes it possible to provide an easy "reverse map" to translate user requests into specific parts of a type. For example, to translate a mouse click on an array into the actual element desired is merely a recursive search of the display tree. The fact that all the formatting is done recursively enables the use of arbitrary types and displays as elements in composite types.

Integration

Because it is important for all parts of the MacGNOME system to work together in a seamless manner, the integration of the data visualizations into the other parts of the Pascal GENIE has been given a high priority.

A planned extension to the system would be to allow the display in the program view to influence the visualizations. For example, a standard feature in the GENIE editor is the ability to collapse (or "elide") any arbitrary piece of the program so they are no longer shown. The elision of a field or fields in a record declaration will imply that the user does not want to see them and so they will be eliminated from the data visualization. Another example is that the user might select some types or variables and specify a particular display style for them. For example, a user could point to a certain array type and indicate that it should be displayed vertically instead of horizontally.

Another future step is to integrate the visualizations themselves into other views. In the current Pascal GENIE, the run time call stack can be dynamically displayed in a standard textual view. We now have the ability to view parameters and local variables in the call stack view. It might also be nice to be able to see visualizations of the current values of variables in the code view, next to the declarations. Such a complete view of the program execution state would remove many of the cognitive roadblocks commonly encountered when teaching such commonly misunderstood concepts as pass-by-reference (VAR) parameters and recursion.

SPECIAL AND CUSTOM DISPLAYS

One special display that will be implemented soon is an iconic view of each data type. This will be useful for showing large data structures, as an elided piece of a structure can be displayed as its icon until the user requests to see more detail. In the future, these same icons will also be used in the tools menu of a system to allow users to construct type definitions graphically.

One very important aspect of this system is the ability to create custom displays for specific types. This has significant pedagogical motivations, since introductory computing courses typically stress a small number of abstract data types. These "standard" types include simple linked lists, stacks, queues, and binary trees. With appropriate default visualizations for these types, an instructor could easily illustrate algorithms and techniques by building example programs, instead of drawing pictures by hand.

In addition, experienced users may want to define custom displays for their own data structures. This will be especially important if the data visualizations are moved to a system which is used by professional programmers. As discussed in previously, custom and special displays can currently be provided by creating new subclasses, and a graphical tool to create these displays will be developed in the future. Each display format is associated with a particular type (either built-in or user-defined). There can be multiple displays defined for the same type, to allow a choice of viewing formats. In this case, the various choices will be available in a standard menu.

One interesting use of special displays is to view a variable of one type as a different type. For example, integer variables used as indexes into arrays might be displayed as pointers into the array. Similarly, a linked list used to implement a variable-length array might be displayed as an array.

An important class of special displays will be used to provide multiple ways of viewing large data. This includes special automatic layout for trees, lists, and arbitrary graphs and networks. In addition, some new techniques will be explored, such as allowing scrolling of individual elements of composite structures (e.g. for a large array which is a component of a record), and fisheye views[25], where a few elements are shown in full detail, neighboring elements are shown in less detail, and far away elements are elided. We also plan to provide high-level abstract views of entire data structures which will use different pictures than just shrunken versions of the detailed view. A visualizer for PROLOG has shown that this is an effective technique [26]. Finally, we plan to investigate the use of three-dimensional display techniques (with perspective) to view multi-dimensional data.

CURRENT STATUS AND FUTURE WORK

Currently, the displays for all of the simple types is implemented, along with arrays and records, as shown in Figures 1-5. The user can point to variables of these types in the call stack and ask for them to be displayed. The system is currently in use by students and instructors in three Pascal courses. The designs for the other parts are complete, except the graphical tool for designing visualizations.

In addition to finishing the implementation of the parts described in this paper and providing some custom displays, there are a number of other directions that will be explored. The first is to formally and informally test whether the visualizations help the students, and how they are used. Another research area is how to allow editing and data entry using the visualizations. Clearly it would be appropriate to allow users to select objects and type new values, or point to new referents for pointers.

We also want to investigate how to make data visualizations useful to professional programmers. For them, the display of large data structures in various formats will clearly be important, as well as easy ways to create custom displays. A problem with the earlier Incense system was that the creation of the visualizations was often more difficult than the actual data manipulation algorithms, which meant that programmers never created custom visualizations [16]. Hopefully, the proposed graphical tool will eliminate this problem.

Another direction is to investigate ways to support algorithm animation, such as provided by Balsa [19]. Here, it is very important to provide an easy way for the programmer to control the timing of the animations, or else the result may be uninterpretable. Also, it is important that data values change smoothly so the user can understand what is happening [21].
After the implementation for full data visualization and editing is complete, it might be interesting to explore how to extend this system to provide Visual Programming, so that the programs can be entered using the graphical displays. This might work by allowing the programmer to design data structures using pictures, and then demonstrate how the data structures should be manipulated by example. Some recent systems have demonstrated that such Programming by Example techniques can be successful [27].

CONCLUSIONS

Traditionally, programming environments have only given the user textual representations of data. We believe that providing data visualizations is an important step towards the creation of an ideal novice environment. The Amethyst system described here provides this functionality in an easy-to-use manner, while allowing further customization and refinement as we gain experience through actual use. Based on the experience of the teaching staff, we have begun with a small set of pedagogically sound visualizations. As the system is used (by about 700 students each semester), we expect to gain insight as to which displays are effective and what additional functionality is needed. The architecture of the system will allow us to easily add and update the default formats based on this experience. The result is expected to be a useful, attractive, and even fun interface to data structures that will significantly enhance the students' learning experience.

ACKNOWLEDGEMENTS

This material is based in part upon work supported by the National Science Foundation under grant number MDR-8652015. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the Foundation. The authors would like to thank Bernita Myers, Phil Miller, and Kim Chandhok for their help with this paper.

REFERENCES

[1] R. Chandhok, et al. "Programming environments based on structure editing: The GNOME approach."Proceedings of the National Computer Conference (NCC'85) , AFIPS, 1985.

[2] Philip L. Miller and Lee W. Miller. Programming by Design; A First Course in Structured Programming. Belmont, California: Wadsworth Publishing Company, 1987.

[3] Brad A. Myers. "Visual Programming, Programming by Example, and Program Visualization; A Taxonomy," Proceedings SIGCHI'86: Human Factors in Computing Systems. Boston, MA. April 13-17, 1986. pp. 59-66.

[4] A.N. Habermann, and D.S. Notkin. "Gandalf Software Development Environments." IEEE Transactions on Software Engineering, December, 1986.

[5] Jim Roberts, John Pane, Mark Stehlik, and Jacobo Carrasquel. "The Design View; A Design-Oriented High Level Visual Programming Environment" In "Proceedings of the IEEE 1988 Workshop on Visual Languages".

[6] Kurt J. Schmucker. "MacApp: An Application Framework," Byte, August 1986. pp. 189-193.

[7] Brad A. Myers. The State of the Art in Visual Programming and Program Visualization. Carnegie Mellon University, Computer Science Department Technical Report no. CMU-CS-88-114, February, 1988.

[8] Lois M. Haibt. "A Program to Draw Multi-Level Flow Charts," Proceedings of the Western Joint Computer Conference. San Francisco, CA. Vol.15 Mar. 3-5, 1959. pp. 131-137.

[9] Nancy Cunniff, Robert P. Taylor, and John B. Black. "Does Programming Language Affect the Type of Conceptual Bugs in Beginners' Programs? A Comparison of FPL and Pascal," Proceedings SIGCHI'86: Human Factors in Computing Systems. Boston, MA. April 13-17, 1986. pp. 175-182.

[10] Man-Chi Pong. "A Graphical Language for Concurrent Programming," IEEE Computer Society Workshop on Visual Languages. IEEE CS Order No. 722. Dallas, Texas. June 25-27, 1986. pp. 26-33.

[11] Pierre N. Robillard. "Schematic Pseudocode for Program Constructs and its Computer Automation by SchemaCode," CACM. Vol. 29, no, 11. Nov. 1986. pp. 1072-1089.

[12] Ephraim P. Glinert and Steven L. Tanimoto. "Pict: An Interactive Graphical Programming Environment," IEEE Computer. Vol. 17, no. 11 Nov. 1984. pp. 7-25.

[13] Tadashi Ae, Masafumi Yamashita, Wagner Chiepa Cunha, and Hroshi Matsumoto. "Visual User-Interface of A Programming System: MOPS-2," IEEE Computer Society Workshop on Visual Languages. IEEE CS Order No. 722. Dallas, Texas. June 25-27, 1986. pp. 44-53.

[14] Thomas Pietrzykowski, Stanislaw Matwin, and Tomasz Muldner. "The Programming Language PROGRAPH: Yet Another Application of Graphics," Graphics Interface'83, Edmonton, Alberta. May 9-13, 1983. pp. 143-145.

[15] J. Michael Moshell, Charles E. Hughes, Lee W. Lacy, and Richard L. Lewis. "A Spreadsheet-Based Visual Language for Freehand Sketching of Complex Motions," 1987 Workshop on Visual Languages. August 19-21, 1987. Linkoping, Sweden. IEEE Computer Society. pp 94-104.

[16] Brad A. Myers. Displaying Data Structures for Interactive Debugging. Xerox Palo Alto Research Center Technical Report CSL-80-7. June, 1980. 97 pages.

[17] Brad A. Myers. "Incense: A System for Displaying Data Structures," Computer Graphics: SIGGRAPH '83 Conference Proceedings. Vol. 17, no. 3. July 1983. pp. 115-125.

[18] Ron Baecker. Sorting out Sorting. 16mm color, sound film, 25 minutes. Dynamics Graphics Project, Computer Systems Research Institute, University of Toronto, Toronto, Ontario, Canada. 1981. Presented at ACM SIGGRAPH'81. Dallas, TX. Aug. 1981.

[19] Marc H. Brown and Robert Sedgewick. "A System for Algorithm Animation," Computer Graphics: SIGGRAPH'84 Conference Proceedings. Minneapolis, Minn. Vol. 18, no. 3 July 23-27, 1984. pp. 177-186.

[20] Marc H. Brown. "Perspectives on Algorithm Animation," Proceedings SIGCHI'88, Human Factors in Computing Systems. Washington, D.C. May 15-19, 1988. pp. 33-38.

[21] Ralph L. London and Robert A. Druisberg. "Animating Programs in Smalltalk," IEEE Computer. Vol. 18, no. 8. Aug. 1985. pp. 61-71.

[22] Robert Adamy Duisberg. "Visual Programming of Program Visualizations," 1987 Workshop on Visual Languages. August 19-21, 1987. Linkoping, Sweden. IEEE Computer Society. pp. 55-65.

[23] Steven P. Reiss. "PECAN: Program Development Systems that Support Multiple Views," IEEE Transactions on Software Engineering. Vol. SE-11, No, 3, March 1985. pp. 276-285.

[24] D.B. Garlan, and P.L. Miller. "GNOME: An Introductory Programming Environment Based on a Family of Structure Editors." Proceedings of the Software Engineering Symposium on Practical Software Development Environments. ACM-SIGSOFT/SIGPLAN, April 1984.

[25] George W. Furnas. "Generalized Fisheye Views," Proceedings SIGCHI'86: Human Factors in Computing Systems. Boston, MA. April 13-17, 1986. pp. 16-23.

[26] Marc Eisenstadt and Mike Brayshaw. "The Transparent Prolog Machine: an execution model and graphical debugger for logic programming," to appear in Journal of Logic Programming. 1988. Human Cognition Research Laboratory Technical Report No. 21a. The Open University. Milton Keynes, MK7 6AA, England. October, 1987.

[27] Brad A. Myers. "Creating Interaction Techniques by Demonstration," IEEE Computer Graphics and Applications, Vol. 7, no. 9, Sept. 1987. pp. 51-60.