# Graphics for Software Visualization


## Henry Lieberman

Visible Language Workshop
Media Laboratory
Massachusetts Institute of Technology
Cambridge, Mass. 02139 USA



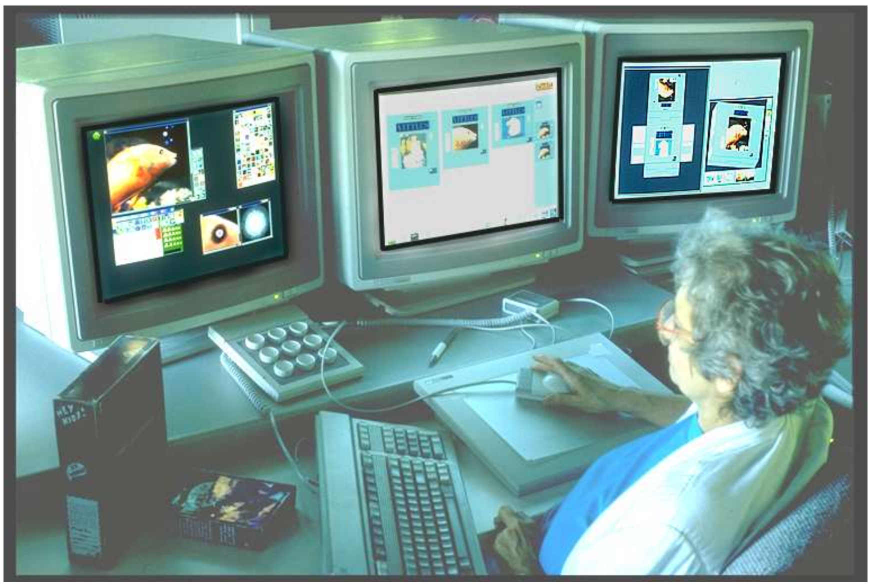
**Abstract**

Paradoxically, very few contemporary interactive environments for writing and debugging computer programs make good use of the high-resolution color graphics screens now becoming commonplace. At the same time, growing complexity of software is reaching a crisis point, outstripping the ability of programmers to envision the relationships between components in a complex program. Just as in the visualization of complex scientific simulations, graphics can be a vital tool in helping to visualize the dynamic behavior of programs. Nowhere is the need for such visualization tools more acute than in debugging, a costly and frustrating phase of programming. Principles of information presentation from graphic design, and theories of the problem-solving processes involved in debugging from artificial intelligence can contribute to innovative solutions.

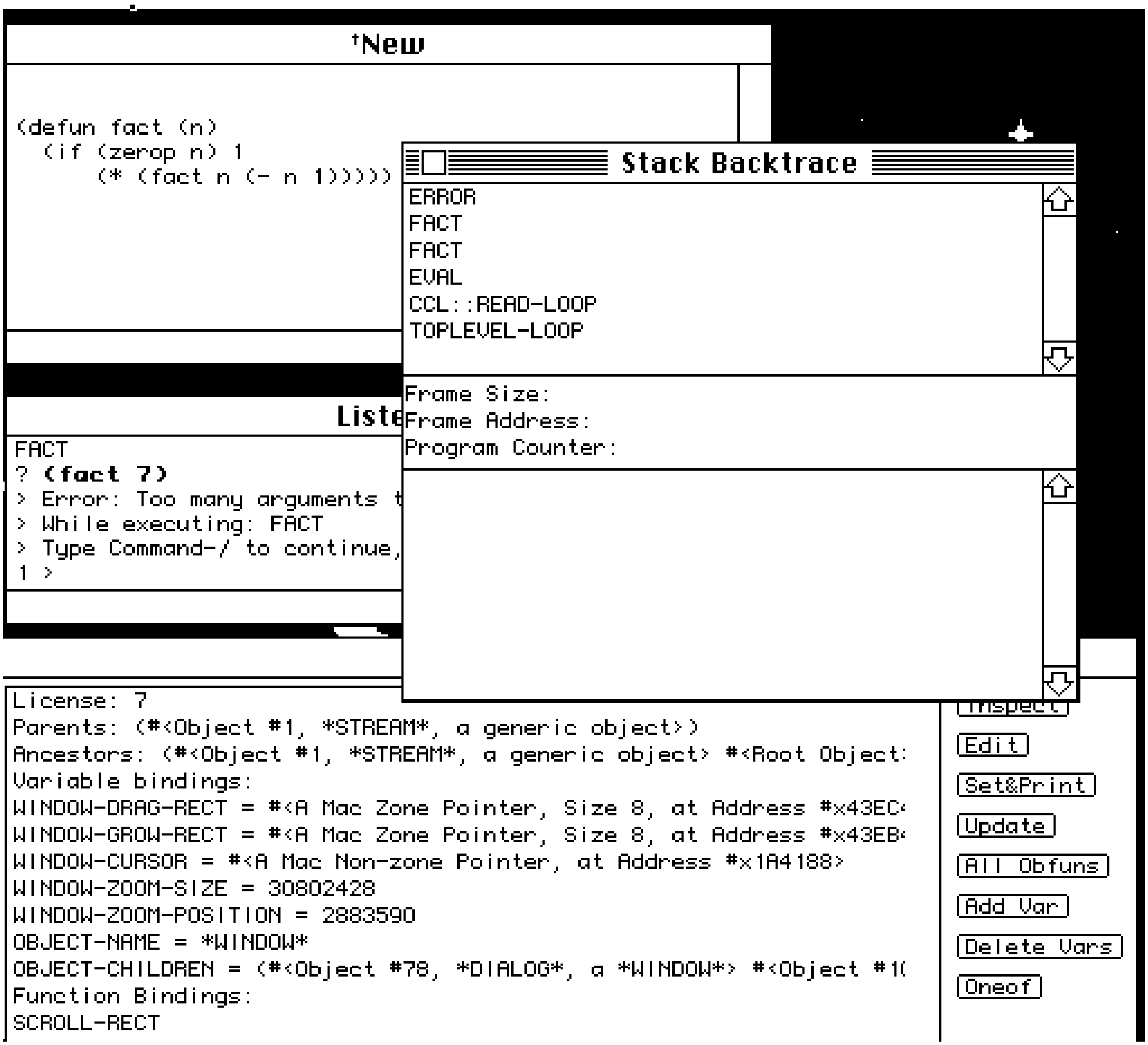
This paper presents some prototypes of alternative visual representations for programs, implemented in the context of an interactive debugger. These range from a relatively conventional textual representation enhanced with animated color typography, to a highly unconventional three-dimensional graphical representation.

**Enhancing the visual richness of programming environments holds promise for dealing with increasing complexity**

Figure "VLW Graphics Environment" shows a graphics environment on the HP 350 workstation at the Visible Language Workshop at MIT's Media Laboratory. This environment includes high-resolution scanned color photography, near real-time three-dimensional polyhedra, anti-aliased typography, and translucent imagery.



The next figure shows a screen from one of the more graphically sophisticated among contemporary programming environments. The contrast in graphic quality between the two environments is striking. The programming environment uses simple linear black-and-white text, in a single low-quality, fixed-width font, and no use of pictorial information at all beyond simple scrolling rectangular windows and menus.




In the past few years, technology in computer graphics has improved drastically. High quality, pixel-mapped color displays are now within the reach of users of low-cost personal computers. New algorithms for creating sophisticated pictures have created a revolution in electronic publishing and in displaying realistic synthesized imagery.

Yet it is surprising how few of these advances have made an impact on the programming environment itself. Even in the most advanced systems, the representation of computer programs themselves remains more or less as it has been for more than twenty years. Programs are simply strings of alphanumeric text, stored in disk files. They are essentially one-dimensional, with vertical relationships between text elements playing no more important role than in "pretty print" formatters. The text is usually displayed in a single fixed-width typeface, in a uniform type size, and in monochrome only. Important features of a program and relatively unimportant ones are not distinguished visually on the screen. The visual representation of the program text remains fairly static, despite the fact that the context in which the program is used is subject to constant change. To date, these remain unchallenged assumptions in most contemporary programming environments.

As computer software is applied to more and more ambitious problems, its complexity grows ever faster. For applications that are close to the edge of the state of the art, dealing with complexity becomes the major obstacle to further progress, even eclipsing such issues as run-time efficiency. This dilemma has become known as the "complexity barrier". The perceived complexity of a computer system is not just a simple function of the amount of information in it, however. How the information is organized, presented, and accessed counts. A complex system can be made manageable if it is organized in a logically layered fashion, if important relationships are clearly presented, if relevant information is easily accessed.

**Debugging tools will serve as the initial testbed for graphical enhancement of the programming environment**

We have chosen to target *debugging* as the most important aspect of the programming environment to serve as a testbed for our ideas in improving the graphic quality of development tools. First, debugging is the aspect of contemporary programming environments that we feel is in most dire need of improvement. Few current systems have debugging tools of any level of sophistication beyond simple tracers and breakpoints. Second, programmers typically spend large fractions of their time in debugging activities, accounting for an often underestimated part of the total cost of software projects. Advances in the technology of debugging will have the greatest leverage in improving the productivity of programmers. Third, the effectiveness of debugging in detecting bugs is a major determinant in the ultimate reliability of the resulting programs.

Yet, despite the importance of the debugging problem, it has received relatively little attention from the developers of programming environments. The past decade or two has seen much work in programming languages themselves, much of which promises only minimal improvements in overall programmer productivity. The debugging tools available in most programming environments have improved little, if at all during the same period.

Work in artificial intelligence applied to the software development process has recently reached the point where cognitive theories of the debugging process can serve as guides to the development of interactive tools. Our previous work [Lieberman 84b] develops a theory of the control structure of the debugging process, so that the design of debugging systems can be made to match the cognitive tasks involved in debugging.

## Debugging is a visualization process

A graphical attack on the debugging problem promises to be especially effective, since one view of debugging is that it is essentially a process of *visualization* [Lieberman 84a]. The use of three-dimensional, animated color graphics in scientific simulation, as advocated in the SigGraph Visualization in Scientific Computing Report [SigGraph 87] makes a radical improvement in the sense of presence achieved. A similarly enhanced sense of presence for the debugging process seen as simulation should be a considerable source of power.

Debugging depends critically on the ability to visualize the internal operation of a program, to compare the program's actual behavior against the programmer's expectations and intentions. With poor debugging technology, this visualization must take place largely 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 lead to more agile and effective debugging by reducing the cognitive burden on the programmer's short-term memory.

In addition, a crying need exists for debugging tools specialized for the needs of programs which themselves have graphical interfaces [Lieberman 86]. Many current debugging tools rely simply on the linear printing of text for output, making them appropriate for debugging programs using textual read-and-print interfaces, but next to useless for debugging programs whose output is in the form of rapid updates to graphical objects on the screen. Frequent dynamic changes to the pixel memory can overwrite or obscure the debugging information. Input from devices like the mouse can cause confusion between the debugger and the program being debugged. A good set of debugging tools for graphical programs needs to make apparent the correspondence between graphical primitives and their visual effects on the screen.

## Applying graphical ideas to debugging tools: visual representations for programs in a single-stepping debugger

We have been experimenting with a variety of graphical interfaces to a single-stepping debugger. The debugger operates on code written in Common Lisp, both because of Lisp's suitability for knowledge-based graphics, simple syntax, and because of the increasing popularity of Common Lisp implementations on a variety of hardware platforms.

We choose a stepper as a particularly appropriate vehicle for our ideas because it presents the programmer with a *dynamic* view of the operation of the program. The key to establishing effective communication between the programmer and the system about the dynamic behavior of programs lies in establishing a coherent *visual language* for two-way communication of properties of program and data elements. This involves making good choices of particular graphic properties and establishing easily-understood mappings between those properties and semantic properties of the program as it runs.

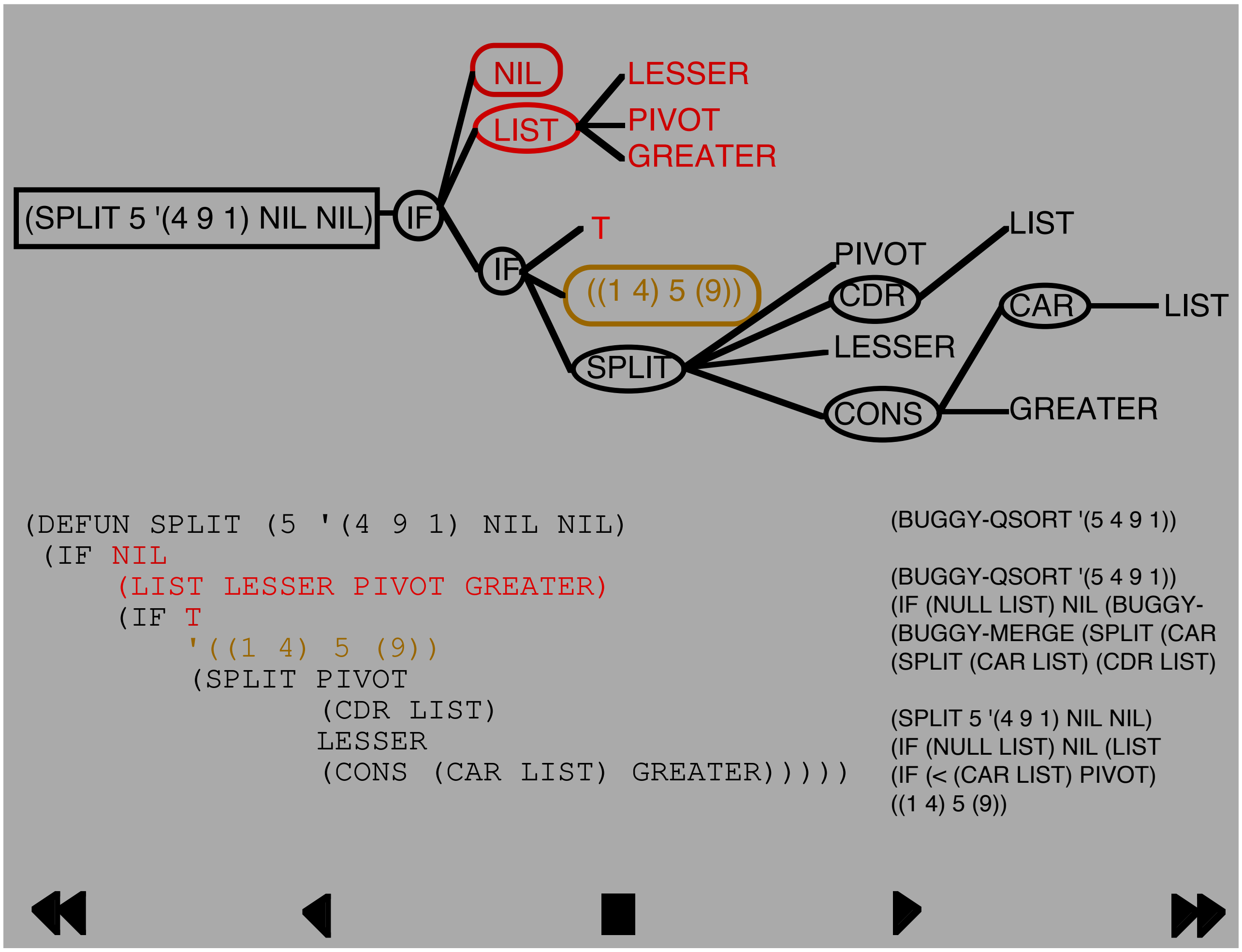
There are three basic kinds of cognitive tasks involved in the debugging process, all of which stand to benefit enormously from enhanced graphical presentation. These are *localization*, which is the task of determining exactly which components of a program take responsibility for an observed symptom of an error, *instrumentation*, which is monitoring the behavior of a particular specified component, and *repair*, replacing incorrect components with correct ones. Instrumentation tools are analogous to oscilloscope probes for electrical circuits; they report the performance of a specified component. The interface to instrumentation tools requires visual emphasis of instrumented code. Localization tools require visual emphasis of "hot spots" to direct instrumentation probes, and responsive control over the level of detail viewed. Repair tools need visual indication of repair points, history of modification and versioning, and improved visual interfaces for editing. Most current debugging tools, such as tracers and breakpoints, fall squarely in the instrumentation camp; the most pressing need is now for improved tools for localization. Among common tools, a stepper is best oriented toward localization, but traditionally, steppers have been hampered by lack of control over the level of detail displayed and poor visual presentation.

Our visual representations for programs and their execution range from minor enhancements to relatively conventional representations, to highly unconventional abstract pictorial representations. Our goal is not to determine a single best representation, but to uncover creative new possibilities in the realm of program visualization. Of course, it is difficult to convey the interactive nature of a dynamic interface such as this in a static paper. For a much better idea of the "feel" of this prototype, we refer the reader to a companion videotape [Cite Software Visualization videotape].

**Animation conveys temporal relationships in a running program**

Programs describe dynamic processes, so animation of program code is essential, especially during debugging. Figure "Animated Stepper" shows a prototype debugging environment, using animated text and typographical highlighting to indicate the temporal relationships of pieces of program code and their values. It shows three different points of view on an executing program simultaneously, updated at every step.

Just as real-time "What you see is what you get" text editors made a revolutionary step by displaying a continuously updated image of a file being edited, this debugger displays continuously updated views of the dynamic state of program execution.




Work at Brown [VanDam 83] and Toronto [Baecker 83] has concentrated on animation of the data manipulated by algorithms, but we feel that it is important to animate the *correspondence* between programs and the data they manipulate to be most useful during design and debugging [Lieberman 84a]. Animation of programs becomes an especially important issue as we move into the realm of *parallel programming,* as we will in the near future. Animation, an inherently parallel process, is best suited toward the presentation of parallel programs.

Many theoretical models of programming languages link the process of evaluation of programming language constructs to processes of *substitution*, transforming a program by substituting replaceable components of the program with their values according to the rules of interpretation. This stepper makes that model visually appararent. As each expression is evaluated, it is replaced by its value. A two-window mode is also available which shows both expression and value simultaneously. Analogues of techniques for editing animated sequences used in the film and video media can be applied for dynamic editing of program state during debugging sessions.

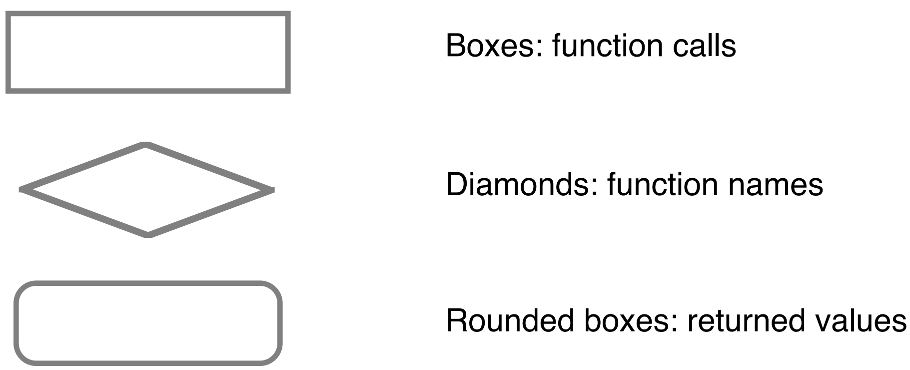
The control structure of this debugger is radically different from conventional debuggers -- it is *reversible*. A user can run it forward until an error occurs, then backward using a remembered history until the precise source of the error is located. Bi-directional control of the animation of program states [Lieberman 87] adds a new dimension of interaction to the debugging process. The user can control level of detail in the debugger by analogy to a *zoom lens* -- a computation can be viewed starting at a very coarse level of detail, then re-examined at successively finer levels of detail until the precise location of the malfunctioning code is discovered.

**Programs can be pictures as well as text**

Graphical represention of programs, such as the venerable flowchart, have been used in the past, but are now enjoying a renaissance due to the increased graphical fluency of contemporary workstations. [Shu 88] is a good survey of experimental programming languages that use pictures instead of text to represent programs. In most of the projects described in this book, the emphasis is on the construction of programming language expressions from interactively drawing and manipulating pictures. We are not only interested in how the pictorial representations look, but how the pictorial representations behave as animations as the program runs. This is essential for debugging, and insufficiently addressed in most of the projects Shu mentions.

For our prototype stepper, we have chosen a graphical representation of the function-calling tree representing the program. Evaluation is displayed as visual replacement of a subtree representing an expression with its value. The graphical form of the program is animated in sync with the textual representation. Boxes repesent function calls, diamonds are the functions, and lines connect arguments to functions. Rounded-corner boxes represent returned values, and the same color-coding scheme used for text is also applied to the graphical representation.

Objects in graph stepper




Certain kinds of relationships are more readily perceivable in the graphical representation than the textual one. For example, a "wrong number of arguments" error is more visually apparent from observing the number of lines emanating from a function than by counting parentheses or commas in a programming language.

**Color provides visual expression of semantic relationships**

Color can be exploited as another tool for increasing the dimensionality of the interface to programming systems. There are basically two kinds of uses of color to convey cognitive information at the programming interface: one relies on sharp distinctions, another on more subtle relationships.

The sharp distinctions are used to provide easily perceivable cues for classifying objects in a small number of categories. Color-coding of subway lines on a map for easy identification is a good example. Programming language constructs or data types can be color-coded. Feedback from the system can be color-coded to indicate its purpose -- error messages could appear in red to emphasize their warning function, less severe warnings in orange, reassuring confirmations in blue. The techniques commonly used for such semantic encoding on black-and-white displays, such as underlining, reverse video, or blinking are far less effective visually and much more disruptive of the visual continuity of the display.

In our prototype stepper, color coding conveys the temporal status of program elements. Code starts out black, brightens to white as it is about to be executed, is replaced by its value in yellow, and finally fades to gray as it recedes into the past. Error messages appear in an attention-getting red.

**Translucency permits overlap without obscuring**

Subtle transparency and translucency color effects [Bolt 77] can be used to increase the effective information density of the display. Intensity and saturation variables can serve as gentle indicators of focus and direction. Layers of translucent text allow the presence of background information to be partially perceived through a foreground display, permitting physical sharing of screen space between the layers. Even if the text is not fully readable, there is some value to allowing its presence to be perceived, just as it is often valuable to see some partially-obscured window on a conventional desktop display. Translucency permits continuous control over visibility relations, whereas mousing of windows gives only discrete control.

This debugger uses translucency as a presentation technique in two ways. First, transitions between succesive states of the program are accomplished using smooth fading. This is important because, as in the use of fading between scenes in cinematic editing, it helps to maintain visual continuity.

Translucency -- partial visibility for visual continuity and unobtrusive presentation

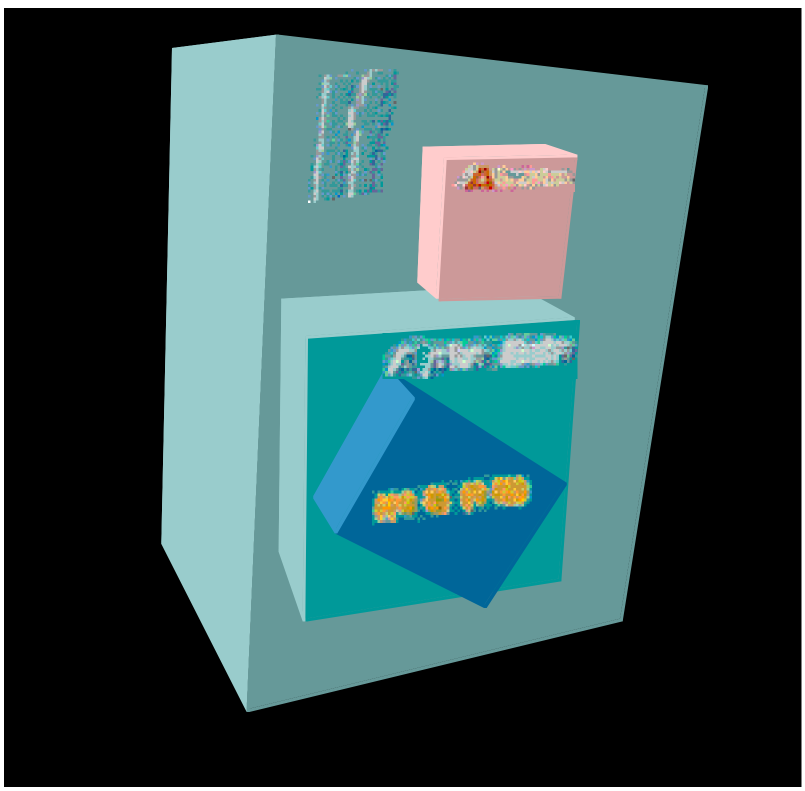

Second, the debugger can be run in a mode where it leaves a translucent image of the previous frame on the screen as the next frame is being displayed. This helps to maintain a sense of history, without being visually disruptive and intruding on the information displayed in the current scene.

**A more radical departure: visualizing programs as three-dimensional objects**

Flat representations of programs and data should give way to multi-dimensional representations. Advances in three-dimensional computer graphics, including real-time

3D polygon-based displays such as the HP Turbo SRX displays, now make practical the use of the third dimension for programming interfaces. Graduation to three-space affords more "room to move" for programming environments, which seem to be constantly fighting against screen space limitations. Recent advances in 3D input [Zimmerman and Lanier 87] even make possible direct manipulation interfaces in three dimensions. These developments will go far beyond the simple "two-and-a-half" dimensional environment afforded by current generation "desktops", which merely display overlapping, flat, rectangular windows.

The next illustration [Figure "3D Stepper"] represents a much more radical departure from conventional presentation of program text. It uses a three-dimensional, pictorial, highly abstract representation of program execution. The picture is from a prototype, built using a 3D visualization and storyboarding system for the HP 350 devised by Bob Sabiston of the Visible Language Workshop. The visual language used here is so unconventional that it takes a bit more explanation to see the correspondence with program code than does the previous 2D graph representation. Again, we face difficulty fully conveying it in the static print medium, and refer the reader to the videotape.

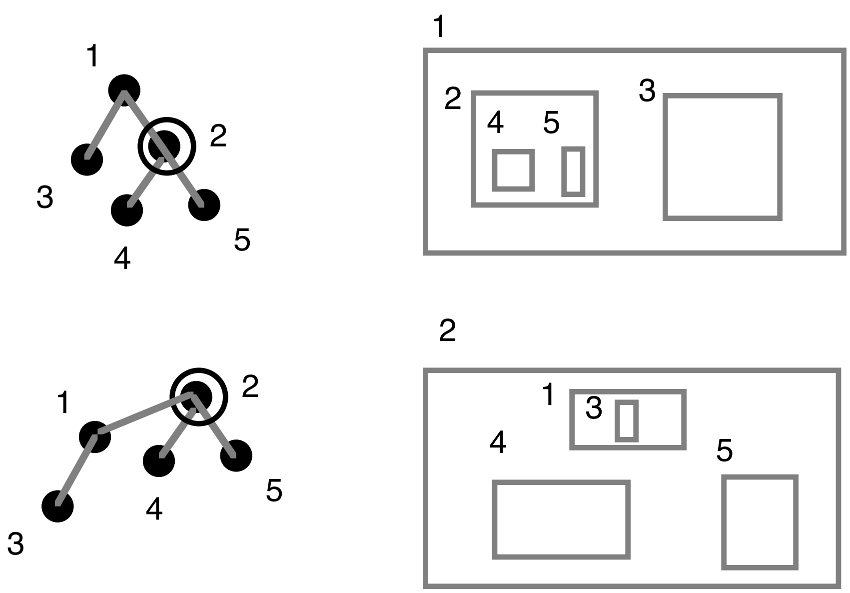


The "cereal box" polyhedra represent program elements. Their size, shape, color and position are signficant. Containment represents function-argument relations. Hue indicates the *direction* of the relation. Saturation indicates the depth of containment. The warm colors indicate travelling inside an expression, and the cool colors indicate travel toward the outside of the expression.

Put another way, the reddish elements are a "windshield" view of the program: things get smaller [and darker] the farther ahead of you they are. The bluish elements are the "rear-view mirror" view: things get smaller as they are farther behind. The outermost box always represents the current focus.

In previous program visualizations, the visual display of the program remains constant, and the locus of execution moves as the program runs. In this representation the point of attention always remains fixed, and the display of the program is continually changed to reflect the current point of view. This *re-rooting* process was inspired in part by a 2D data browser constructed by the Media Lab's Mike Travers for the Cyc knowledge base at MCC.

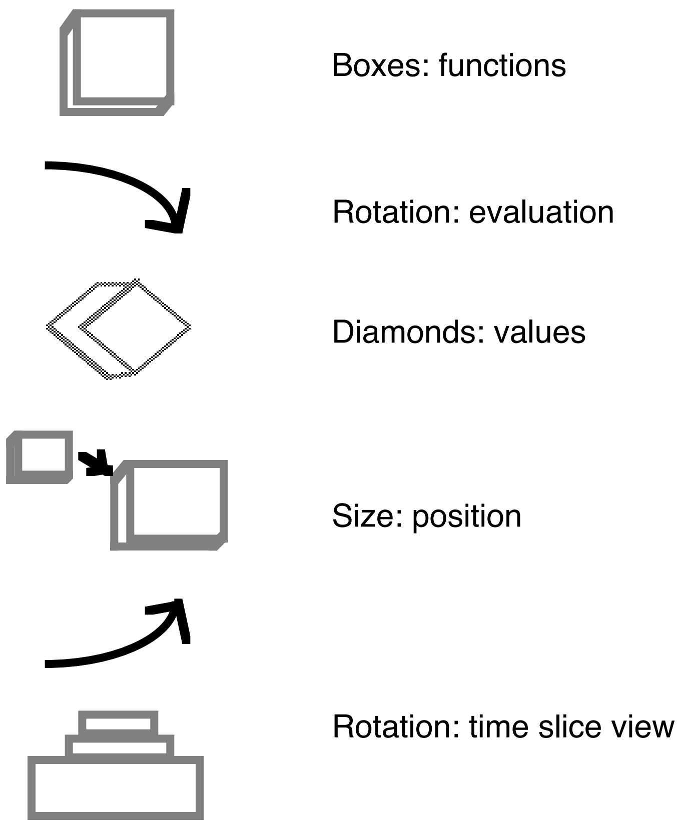
Re-rooting keeps viewpoint fixed at root node




The animation shows boxes grow and shrink smoothly as the program runs. Values returned are represented by diamonds. Thus, in this system, evaluation is modeled physically by rotation.

Note that the boxes are also labelled on their sides. We take the metaphorical use of the word "stack" in programming and give it direct visual reality. A face of a rectangular solid displays a momentary view of a program element. Sucessive views can be "stacked up" in three-space, thus mapping the third dimension onto time. This provides a direct spatial representation of the correspondence between a stack frame and the code running in it, unlike flat representations. With 3D input, a direct manipulation interface could allow simply rotating the solid to view it from another perspective. Positioning the solid so that the edges holding the labels of the stack frames are facing the viewer generates the time-slice perspective of a traditional stack browser.

3D polyhedra represent program elements




Is this a good visual representation for programs? We don't yet know. Users seem to find it strange at first, but there is a rapid process of adaptation that comes as one learns to associate the visual cues with the properties of program execution. Exploring a variety of such representations should yield insight into how visual presentation affects conceptualization.

Both the 2D graph and the 3D "cereal box" representation are small examples of how *spatial memory* can be exploited as a tool for dealing with navigational problems in the programming environment. As systems grow larger and larger, finding things within the system becomes a more and more critical task. When confronted with a large and unfamiliar system, considerable time needs to be spent learning what facilities the system has to offer, a process known as *browsing*. The information learned is mentally indexed and recalled by *where* it was located. In graphically rich computer environments, the conceptual space of a software system can be mapped onto the visual space of a computer display screen. Our innate talents at locating real-world objects in space can serve as a cognitive aid in search problems.

**Summary: A comparison of visual languages for program execution**

## Visual languages for program execution

| | Conventional | WSYWIG text | Graph | Cereal boxes |
|---|---|---|---|---|
| **Program representation** | Linear text | Prettyprinting | Tree | Polyhedra |
| **Function argument relation** | Nested parens | Nested parens | Node-arc connections | Nested polyhedra |
| **Temporal relations** | Linear printout | Color and typeface changes | Color and typeface changes | Color and size changes |
| **Evaluation** | Linear printout | Subexpression replacement, translucency | Subgraph replacement, translucency | Animated rotations, growth, shrinkage |
| **Viewpoint motion** | Linear printout | Animated highlighting | Animated highlighting | Re-rooting |
| **Time slice view** | Incremental linear printout upon request | Dynamically updated stack | Dynamically updated calling graph | Polyhedra rotation |

The chart summarizes a comparison of the visual languages for representing program execution presented above.

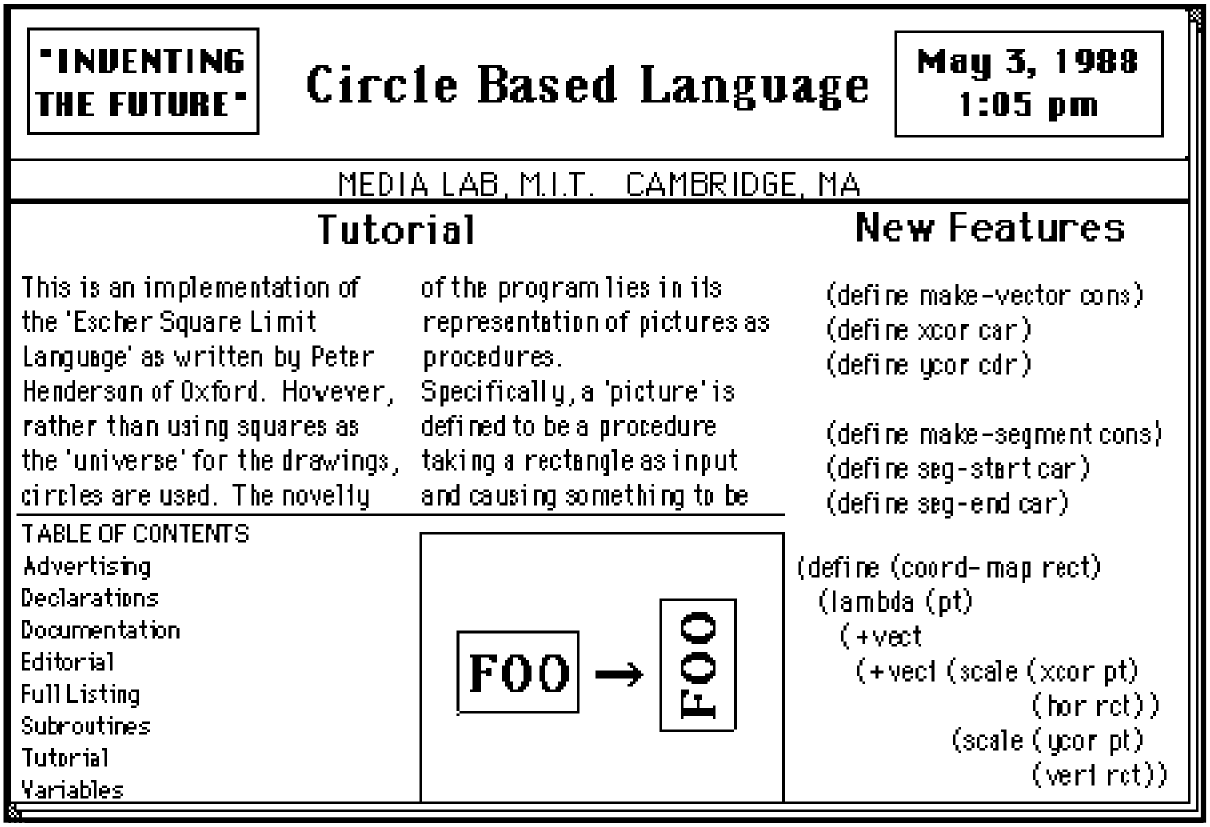
**Expertise in traditional graphic design can be applied to graphical problems in the programming environment**

Graphic design is a discipline which strives to increase the effectiveness of visual communication. Graphic designers make functional choices about the kinds of visual elements used in written communication and their spatial arrangement in order to facilitate the transfer of information. A key issue in graphic design is to find a good match between the choices of visual components and their intended purpose in the material. Good graphic designers know that well-considered choices of type styles and layouts make a tangible difference in the readability of books or the visual impact of advertisements. Environments for computer programming would be much improved if we could increase the effectiveness with which computer programs perform their role as vehicles of communication. We sense that an opportunity exists to appropriate knowledge from the traditional graphic design community to apply to the problem of how to improve the communicative function of programming languages.

It is now commonplace for designers of interactive computer interfaces to consult professional graphic designers for suggestions to improve the appearance of their screens. Rather than just make design judgments on an ad-hoc basis for each interface, we would like to try to represent in the machine some of the underlying design knowledge, so that intelligent presentation choices can be made dynamically. Part of the process involves using the knowledge engineering techniques of artificial intelligence, eliciting protocols from expert designers and embedding the knowledge in the machine in the form of rules.

Some work has been done in applying traditional graphic design criteria to presentation of program text [Baecker and Marcus 83], but this work has been limited to two-dimensional black-and-white text on static paper hardcopy, whereas we are more interested in the potential of dynamic screen display with an extended graphical vocabulary.

Baecker and Marcus took as their metaphor the traditional design of hardcopy *books*. We have been experimenting with a alternative metaphor for screen presentation based on a *newspaper* metaphor. We have been analyzing how layout decisions are made for newspapers and trying to apply, where appropriate, some of this knowledge in designing screens for programming environments. Below, a screen shot from a Hypercard prototype of this idea implemented by MIT student Keith Mackay and the author.




Several considerations make newpapers a more appropriate metaphor than books for screen organization for interactive programming environments. First, newspapers are a more dynamic form of media than books. Information in a newspaper is expected to change more rapidly than information found in books. Many decisions made in the presentation of information in newspapers reflect this fact.

The format of a newspaper is designed to present many kinds of information simultaneously, at varying levels of detail, some of which are related and some of which may be unrelated. In a programming environment, the programmer may want to look at information from a variety of sources at once, perhaps regarding unrelated parts of one or several programs. In contrast, a book is generally about a single, coherent topic. Books are generally designed to be read sequentially, cover to cover. Newspapers are designed to be read in small bursts, jumping around from one part to another to suit the reader's interest. This non-linear pattern of cosuming information is more appropriate to the situation of programming environments.

The graphic design of newspapers is oriented toward management of a fundamental resource: the limited attention of the daily reader. Newspapers know that no reader can read all of the day's news. They must concentrate on distinguishing the truly important from the less important, and directing readers' attention to those items of personal interest. In programming environments, programmers are overwhelmed by the amount of information generated by programs, so arranging a layout to optimize the attention of the programmer is very important. Newspapers must, as must programming environments, balance information presentation functions [looking at a newspaper just to find out "what happened today"] with information retrieval functions [searching directly for sports, financial, fashion or other news of particular personal interest].

And finally, newspapers never waste space. We have been searching for ways to break out of the now-ubiquitous "desktop metaphor" of overlapping rectangular windows, often wasteful of scarce screen real estate. The tightly-packed format of a newspaper, where articles "flow" to fill up spaces is attractive in the highly-limited screen space of most personal computers.

**Cognitive issues in the programming interface: bandwidth and attention**

Though we do consider the aesthetic aspects of programming environments to be important, concern for the graphical aspects of programming environments is not just cosmetic. Our goal is to make programming environments more effective by matching the graphical characteristics of the programming environment to the cognitive tasks involved in the programming process.

A key issue is increasing the *bandwidth* of the interface to a programming environment. People are capable of absorbing and transmitting information at much higher rates than present-day interfaces are geared toward handling. The effectiveness of many operations, such as information retrieval and program debugging is critically dependent upon the bandwidth of the interface. Some ways of increasing the bandwidth of interfaces are to use faster processor speeds or higher-resolution screens. However, if the interface is still limited to presenting linear, sequential text at ever-increasing rates, these may only result in increased user confusion due to screen clutter, screen disruption, and visual fatigue. Color, typographics, 3D and other visual properties are all means of information presentation that can increase the real bandwidth of the interface, without increasing the rates at which text must be presented and read by the user. They can do this because they effectively harness the tremendous innate processing power of the human visual system.

To manage complex systems effectively, *attention* is a critical issue. If there's a great deal of detail in a system, one must learn to pay attention only to what is important, ignoring everything else. What is considered important, of course, changes over time with changing purposes, goals and intentions. The use of different type size, color, position, and other visual properties can help the user and system communicate about what parts of the program are important, and for what purposes. Animation causes these properties to change over time, tracking shifts of the focus of attention.

Issues of attention become most critical when considering the needs of programming interfaces oriented toward *non-expert* or *occasional users*. As computer applications become more widespread, more and more people with little prior computer experience will be called upon to, at least occasionally, play the role of programmer. Beginning or infrequent users have very little tolerance for the cluttered and confusing displays to which expert programmers have become numbed by long experience. More effective use of a richer graphical vocabulary is a route toward being able to drastically increase the bandwidth of an interface without leading to the cluttered displays that overwhelm non-expert users.

**Acknowledgments**

Major support for the work described in this paper was provided by Hewlett-Packard. I would like to thank Nick Copping, TW Cook, JB Eriksen, and Pat Cole at HP's Software Development Environments group, and Muriel Cooper and Nicholas Negroponte at the MIT Media Lab for their support of this project. The Visible Language Workshop is also supported by grants and equipment from HP, IBM, Apple, DARPA, and NYNEX and Xerox.

## • Bibliography


[Apple 84] Apple Documentation Group, Inside Macintosh, Apple Computer, Cupertino, CA, 1984.

[Brand 87] Stewart Brand, The Media Lab: Inventing the Future at MIT, 1987.

[Baecker 83] Ronald Baecker, Sorting out Sorting, Color videotape, Human Computing Resources, Toronto, Canada.

[Baecker and Marcus 83] Ronald Baecker and Aaron Marcus, On Enhancing the Interface to the Source Code of Computer Programs, Computers and Human Interaction Conference, San Francisco, 1983.

[Bender, et. al. 85] Walter Bender et. al., Color Word Processing, IEEE Computer Graphics and Applications, June 1985

[Bolt 77] Richard Bolt, Color Transparency Effects From Mosaics of Opaque Color, MIT Architecture Machine Group, June 1977.

[Bolt 84] Richard Bolt, The Human Interface, Lifelong Learning Associates, 1984.

[Levitt 86] David Levitt, Hookup, Conference on Small Computers and the Arts, Philadelphia 1986.

[Lieberman 84a] Henry Lieberman, Seeing What Your Programs are Doing, International Journal of Man-Machine Studies, July 1984.

[Lieberman 84b] Henry Lieberman, Steps Toward Better Debugging Tools for Lisp, Proceedings of the Third Lisp Conference, Austin, Texas, August 1984.

[Lieberman 85] Henry Lieberman, There's More to Menu Systems Than Meets the Screen, SigGraph Computer Graphics Conference, San Francisco, July 1985.

[Lieberman 86] Henry Lieberman, Debugging Graphical User Interfaces, SigGraph Workshop on Software Tools for User Interfaces, Seattle, October 1986.

[Lieberman 87] Henry Lieberman, Reversible Object-Oriented Interpreters, First European Conference on Object-Oriented Programming, Paris, June 1987.

[London 86] Ralph London, et. al. Animating Program Behavior in Smalltalk, First Conference on Object-Oriented Systems, Languages, and Applications, Portland, Oregon, 1986.

[Ludolph 88] Frank Ludolph, Dan Ingalls, et al. The Fabrik Visual Programming Environment, OOPSLA 88, San Diego

[Mackinlay 87] Jock Mackinlay, Automatic Design of Graphical Presentations, Ph.D. Thesis, Stanford University, 1987.

[Rajan and Eisenstadt] Tim Rajan and Marc Eisenstadt, The Transparent Prolog Machine, Open University, Milton Keynes, England, 1988

[Shapiro 86] Ehud Shapiro, Algorithmic Program Debugging, MIT Press, 1986.

[Shu 88] Nan Shu, Visual Programming, Van Nostrand Reinhold,1988

[SigGraph 87] SigGraph report and videotape, Visualization in Scientific Computing, ACM Press, 1987

[Van Dam, et. al., 1985] Andy Van Dam, et. al., BALSA - The Brown Algorithm Simulator, Brown University, 1985.

[Zimmerman, Lanier, et. al. 1987] Tom Zimmerman, Jaron Lanier, et. al., A Hand Gesture Input Device, Computers and Human Interaction Conference [CHI+GI], Tortonto, Ontario, April 1987.