A Three-Dimensional Representation For Program Execution

Henry Lieberman

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

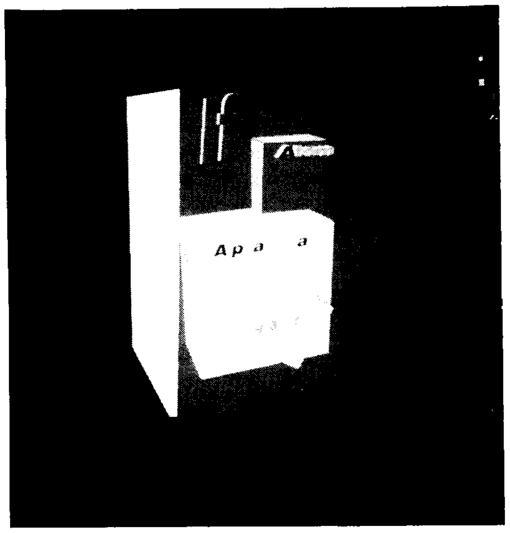
Abstract

Modern graphical workstations make possible interactive real-time manipulation of three-dimensional objects. While 3D graphics is usually used to model real-world objects, in this paper we explore an abstract three-dimensional pictorial representation of computer programs. Since programs are descriptions of dynamic processes, the focus is on the dynamic behavior of the graphical representation, resulting in an animation of the program's execution. The goal is to aid the debugging process by helping the programmer visualize the dynamic aspects of a program's behavior. 3D representations help make use of the enormous innate power of the human visual system.

Shu's book on visual programming [Shu 88] discusses many projects which use pictorial representations for computer programs and their data. Most of the representations that are described in Shu's book and other sources are two-dimensional, and concentrate mainly on the static appearance of program code. The experiments described in this paper are aimed at extending the visual vocabulary for graphical programming, using an abstract three-dimensional representation for computer programs in the context of an interactive dynamic debugger. Three-dimensional representations are now made practical by hardware which automatically performs rendering of solid objects just as easily as displaying a line of text.

Other graphical techniques employed in this debugger to provide an integrated view of program execution are:

- a *re-rooting* process which keeps a fixed viewpoint while displaying a changing context,
- the display of evaluation as substitution of graphical objects,
- a reversible control structure,
- use of 3D rotation to simultaneously present sequential and time-slice views.



• Programming in three dimensions

Program elements are represented by colored polyhedra. Each program element is labeled by three-dimensional text attached to a face of the program element. The size, shape, color and position of program elements are significant. The properties of the polyhedra change to reflect the current state of the program. Boxes grow, shrink, turn, move, and change color as the program runs. We present this illustration now to show the reader what the display looks like. Later, we explain the reasons for each design choice in detail.

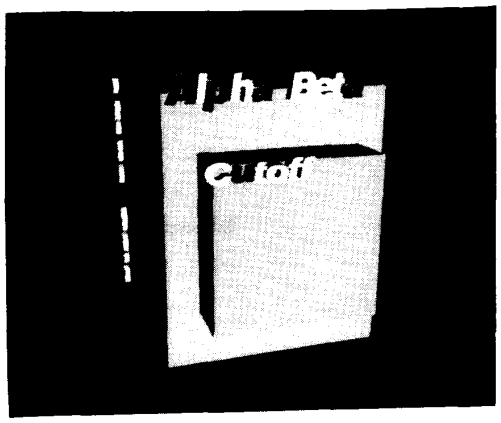
The examples in this paper were generated on an HP 350 Bobcat workstation with a Renaissance display, which has hardware for rendering shaded polyhedra. We are converting to a next generation machine which we hope will be fast enough to manipulate these displays interactively in real time. The examples were produced using a 3D storyboarding system designed by Bob Sabiston, and are currently being hooked to the debugging interpreter described in [Lieberman 87] and [Lieberman 89].

• Display of programs as tree structures

There are two related aspects to the visual representation: static display of program structure, and its dynamic behavior during evaluation. Rather than simply display the static structure of the program as source code text, a more useful graphic representation makes apparent the tree structure of function calls and arguments which comprise the program. In functional languages such as Lisp, the textual syntax uses nested parentheses and "pretty-printing" to make this tree structure apparent.

We have previously experimented with graphical representations that draw the calling tree with functions and arguments represented as nodes and arcs [Lieberman 89]. This is a good representation for displaying the tree structure in two dimensions. Note that drawing the function-calling tree is *not* the same kind of graphical representation for programs as the venerable and justly-maligned flowchart.

Another possibility for depicting the function-argument relationship spatially is to use *containment* of graphic forms. This has been done in several graphical programming projects, with [Smith 79], [Lakin 80], and [Abelson, et. al. 87] as good examples. A large icon representing a function graphically encloses smaller icons which represent arguments. This is the approach



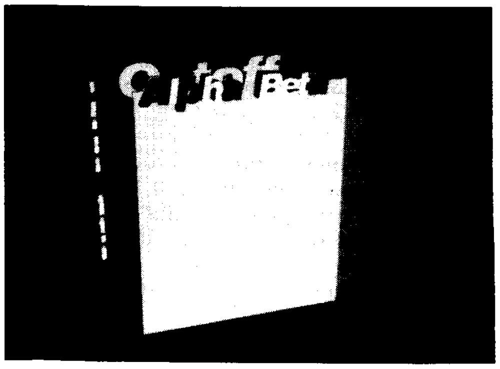
adopted here, except in three dimensions rather than two, as illustrated above. This represents a function call to a function named `Alpha-Beta`, with an argument named `Cutoff`. We could have more complex three-dimensional abstract pictorial representations of program elements, but here we stick to boxes as the simplest 3D shapes, distinguished only by text labels.

• Move the focus or move the context?

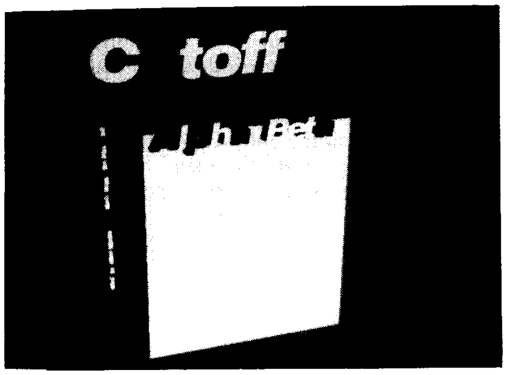
All visual representations for dynamic display of programs are faced with a common design decision: How should we display the relationship between the particular part of the program that is the "focus" of current interest [such as the source expression currently being evaluated] and the rest of the program?

There are two fundamental possibilities for display of the viewpoint or current focus of interest. First, the program display itself can remain static, and the viewpoint can move by means of highlighting, blinking, cursor motion or some other attention-getting device. This is the alternative used in most traditional static, textual program representations. Display of source text, highlighting the current expression, and "follow the bouncing ball" cursors are examples of this approach. The alternative, which is much less common in programming environments, is to leave the viewpoint fixed, and alter the display of the surrounding context to reflect the current point of view.

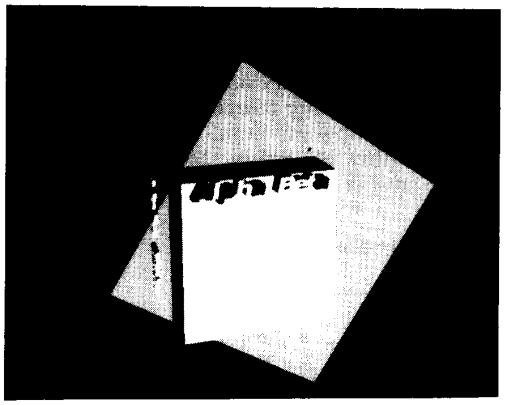
Many text editors use both "move the focus" and



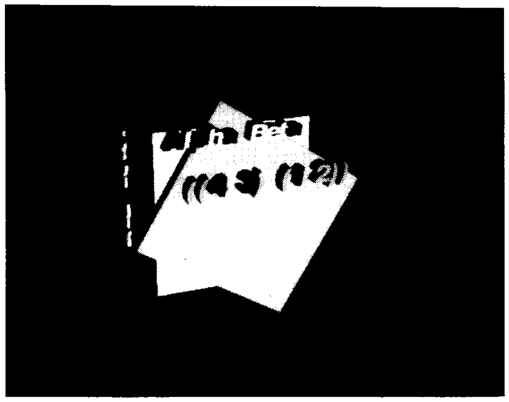
The pictures on this page illustrate a sequence in the animation of the execution of a call to the ALPHA-BETA function. The initial state was shown in the second illustration of this paper. Evaluating the argument, it grows to intersect the shrinking function.



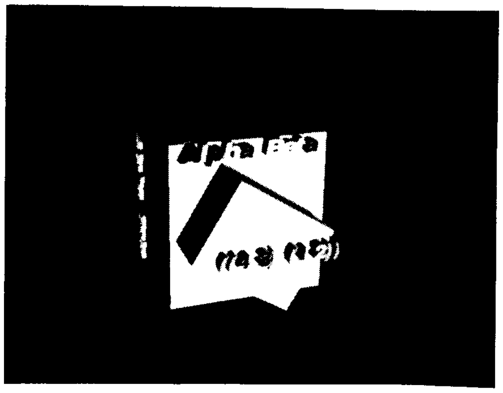
The argument CUTOFF now becomes the outermost box, signifying that it is the current focus of evaluation. The function changes from gray to blue, to show that it is receding from interest, while the argument changes from red to gray.

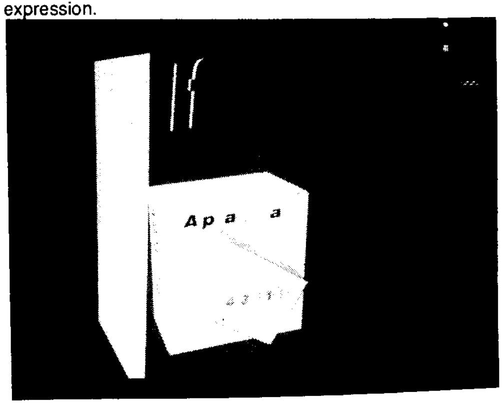


The argument turns to indicate that it is being evaluated. As the value returns, the label is replaced by that of its value. Labels for program expressions are white, and labels for values are yellow.



As the focus of attention moves away from the list returned as a value, the diamond shrinks and changes color to blue. As we invoke the definition of ALPHA-BETA, the expression beginning with IF pops up to surround the previous expression.





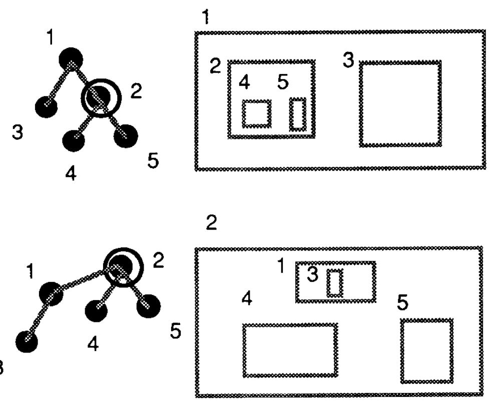
"move the context" approaches. The motion of a text cursor within a single window is a "move the focus" interface -- the text does not move, but the cursor indicating the current position does move. Scroll bars on a window, however, are an example of a "move the context" interface. The focus, namely the outline of the window which presents text to the user, remains fixed, while the context, the text displayed in the contents of the window, shifts to reflect the new position of the scroll bar.,

In the past, preference for static display of context and dynamic display of focus was motivated in part by the need to minimize the amount of dynamic display update, since the focus is usually much smaller than the display of the surrounding context. The increasing speed of modern computational platforms now gives us the freedom to experiment with the alternative, since globally re-writing the screen at each change is no longer prohibitive. Though this approach sacrifices something in visual continuity between successive frames, it has some advantages in making it easier to control the level of detail in large programs. By experimenting with dynamic display of context and static display of focus, we hope to find out what the real tradeoffs are between these alternatives once the unfamiliarity and novelty wear off.

• Re-rooting: A "move the context" interface for tree structures

Re-rooting is a method of "shifting focus" in tree structures. The idea is that when you move from the root node of a tree along one of its branches to a target node, you can view the result by a new tree that has the target node as its root and maintains all the same connectivity as the original. In the illustration, we move from the node marked 1 in the tree to the circled node 2. To the right of each tree, we display a tree of rectangles ordered by containment, and show the effect of the re-rooting operation. Re-rooting in the containment tree causes the target rectangle 2 to grow to become the outermost rectangle, and the former root shrinks to fit within it.

Re-rooting keeps viewpoint fixed at root node



Re-rooting a containment tree is a good way to emphasize the target node, because it is always the largest displayed object. This is why re-rooting is a good strategy for displaying program trees, because the expression currently being evaluated is always the largest, outermost object.

One problem with simple re-rooting, as illustrated above, is that some information, namely that 1 was the original root node, is lost. In general, when you re-root a tree, some of the links which originally all pointed "downward", may then point "upward". For display of program trees, the direction is significant and must be preserved. Our solution is to use *color* to indicate direction. The warm colors all indicate downward links, and saturation is an indicator of tree depth. The cool colors indicate upward links, also becoming darker the farther upward in the tree you get. Things get smaller and darker the farther away you get, either in the upward or downward direction.

Put another way, the reddish elements are a "windshield" view of the program: things get smaller [and darker] the farther ahead 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, and is gray, the "midpoint" between the reds and the blues.

I first came across the idea of re-rooting in [Baker 1975]. My use of re-rooting to display program trees was inspired by a two-dimensional browser for static data structures in the CYC knowledge base by Michael Travers [Travers 1989].

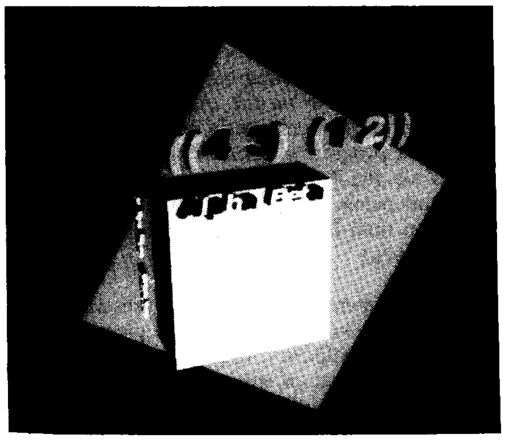
• Displaying evaluation: rotation and substitution

Re-rooting is used for display of the static program tree, but the main type of event that needs to be displayed dynamically is the evaluation of program expressions and display of returned values. [We do not consider here explicitly any special treatment for code with side effects.] Our basic model of evaluation is as a *substitution* process. To display an evaluation, a display of the value of the expression is substituted for the display of the expression itself. Substitution is a simple, easily explained graphic model for evaluation. The substitution model and its role in graphical debuggers is described in more detail in [Lieberman 84] and [Lieberman 89].

We need some graphic device to distinguish expressions that are to be evaluated from values returned. In the three dimensional representation, evaluation is represented graphically as *rotation* about the axis which points toward the viewer. This keeps the same labeled face toward the viewer, while maintaining a clear distinction between the expressions, which are boxes, and values, which are diamonds. The text labeling the value replaces the text that labels the expression.

In this version, the replacement operation is performed suddenly, but we have also experimented with interfaces in which creating or removing display objects is performed gradually, with fade-ins and fade-outs. This helps enhance the visual continuity of the display. We have also experimented with *translucent* display of objects, so that less important information [such as a history of previously seen states] remains unobtrusively displayed on the screen but does not obscure more important information.

The effect of re-rooting and value substitution is shown in the illustration. The `Alpha-Beta` expression has been re-rooted to the variable `Cutoff` and then evaluated, so that the outermost object is the diamond representing the returned value, the list `((4 3) (1 2))`.



• The third dimension: a "side view mirror"

The use of the third dimension helps solve a tricky problem in presenting a dynamic view of a program. There are two different points of view that the user is typically interested in seeing. The first is the "head on" view of the running program as a sequence of events, ordered sequentially by time, as would be presented by a conventional single-stepping debugger. The second is what is traditionally called the "stack" in many programming environments, a time-slice view of the active goals the program is trying to satisfy at the moment. This is a path from the root of the computation to the currently active node.

In conventional debuggers, you normally see the sequential view of events, but it possible to stop the program and request a view of the stack at a given moment. In many systems, each "stack frame" must be requested individually. In [Lieberman 89], this situation is improved by showing a continuous display of the stack in a separate window, automatically updated dynamically as the program runs. This provides an overview of the stack without having to specifically request it, and the animated correspondence between changes in the stack and changes in the program code help give the user a better feel for what the program is doing.

But this is still unsatisfactory, since the stack and program displays are in separate windows, the correspondence between a program expression and its stack frame is not made visually apparent.

In the three-dimensional program representation, each program expression is represented by a box, and in addition to labeling each box on its face, we also *label the box on its side.* As you move away from the expression currently being evaluated, each smaller program element is *moved toward the viewer.* Thus, the Z axis is being used as the "stack depth" dimension. [Actually, this displays more information than is typically available with most stack browsers.]

In the three-dimensional program representation, the physical metaphor of a "stack" is given direct visual reality. While the "head on" view normally shows the progress of events sequentially, *the stack view can be seen simply by rotating the three dimensional point of view.* Each successive stack frame "sticks out" of its containing expression so that by rotating the point of view, the "side view" shows all stack frames at once. Rotation, pan and zoom operations performed by the hardware can easily emphasize some portion of the stack of particular interest to the user. Since seeing the stack is simply another viewpoint on the program display, the correspondence between each event and its associated stack frame is visually evident.

• Acknowledgments

Major support for the work described in this paper was provided by a grant from Hewlett-Packard. I would like to thank Nick Copping, TW Cook, JB Eriksen, and Pat Cole at HP's Software Engineering Systems 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

[Abelson and diSessa 87] Hal Abelson and Andy di Sessa, Boxer: A Principled Design for an Integrated Computational Environment, CACM, 1987

[Baker 75] Henry Baker, Shallow Binding in Lisp 1.5, MIT Artificial Intelligence Lab, 1975.

[Baecker 83] Ronald Baecker, Sorting out Sorting, Color videotape, University of Toronto Media Centre, Toronto, Canada.

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

[Lakin 80] Fred Lakin, Computing With Text-Graphic Forms, Proceedings of the First Lisp Conference, Stanford, California, 1980

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

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

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

[Lieberman 89] Henry Lieberman, Graphics for Software Visualization, MIT Media Lab, 1989

[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

[Smith 79] David Canfield Smith, Pygmalion: A Creative Programming Environment, Birkhauser-Verlag, 1979

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

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

[Travers 89] Michael Travers, A Visual Representation for Knowledge Structures, Hypertext Conference, Pittsburgh, PA, USA, November 1989.

[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.