

Software Visualization for Debugging

Representing programs through algorithm animation, typographic source-code presentation, and interactive auralization transforms the hunt for bugs into a cognitively accessible multimedia experience.



SOFTWARE VISUALIZATION IS THE SYSTEMATIC AND IMAGINATIVE use of the technology of interactive computer graphics and the disciplines of graphic design, typography, color, cinematography, animation, and sound design to enhance the comprehension of algorithms and computer programs [12, 13]. Here, we demonstrate that graphical and auditory representations of programs¹ are useful in debugging and can enliven and enrich programming as a cognitively accessible multimedia experience.

To illustrate these ideas, we present three visualization approaches we have explored—algorithm animation, typographic source code presentation, and interactive auralization for debugging—demonstrating the richness of software visualization media and portraying design trade-offs inherent in their

use. We use a 30-minute film (designed to teach nine sorting algorithms) to demonstrate the power of algorithm animation. We show how the design and typesetting of computer program source text can enhance the program's readability. And we show how a programming environment we created—LogoMedia—is useful for the interactive construction of visualizations during program creation and debugging.

¹We use "visualization" in the sense of forming a mental image of something that can be aided by graphical, auditory, and other sensory modalities.

Visualization

Algorithm Animation: How Does the Program Work?

Animation is a compelling medium for displaying program behavior. Programs execute over time, so they can be vividly represented through animation, portraying how they carry out their processing and how their essential state changes over that time. A program's state is determined by its data, so one way to portray a program is to show the data transforming over time. By viewing these transformations, or several sequences resulting from different sets of initial data, we can perceive structure and causality, ultimately inferring why and how the program is working or not working.

Software visualization is a powerful tool for presenting algorithms and programs and assisting programmers and students struggling to debug them. But animating algorithms is not a trivial endeavor; to be effective, algorithm animation must abstract or highlight only those aspects of a program essential at the moment. Visualizations must enhance relevant features and suppress extraneous detail; employ clear, uncluttered, and attractive graphic design; and use appropriate timing and pacing.

We demonstrated the power of this idea in our 1981 30-minute color and sound teaching film *Sorting Out Sorting* [1, 2], which uses animation of program data and an explanatory narrative to teach nine internal sorting methods. The movie has been used successfully with computer science students in universities and high schools. Students who have watched the animation carefully can program the methods themselves and understand the concept of efficiency differences between n^2 algorithms and those algorithms whose execution time is proportional to $n \log n$.

Assume, for example, we wish to sort an array of numbers. We can portray each datum as a vertical bar (see Figure 1), whose height is proportional to the datum's value. Initially, the heights

of adjacent items vary upward and downward. Successive steps produce rearrangements of the data until ultimately the elements are arrayed left to right in order of increasing height.

The film deals with three *insertion sorts*, three *exchange sorts*, and three *selection sorts*, beginning with insertion sorts, in

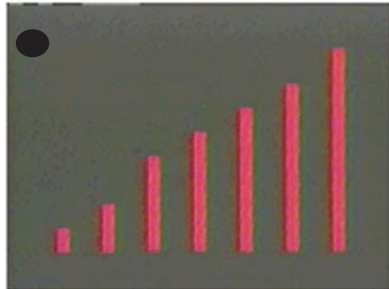
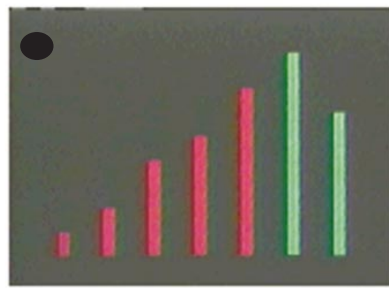
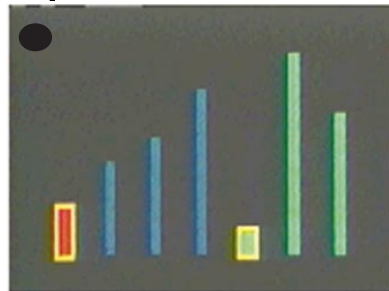
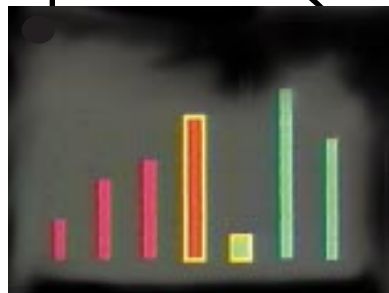


Figure 1.
Linear Insertion

FOR EACH NEW ITEM, VIEWERS SCAN THE ARRAY SORTED SO FAR, LOOKING FOR THE CORRECT POSITION; WHEN WE FIND IT, WE MOVE ALL LARGER ITEMS ONE SLOT TO THE RIGHT AND INSERT THE NEW ITEM. (A) FIRST COMPARISON OF THE FOURTH PASS, WITH THE FIRST FOUR ITEMS ALREADY ORDERED; (B) FINAL COMPARISON OF THE FOURTH PASS; (C) END OF THE FOURTH PASS, AFTER THE FIFTH ITEM HAS BEEN MOVED TO THE FRONT; (D) DATA HAS BEEN SORTED. GREEN DENOTES "UNSORTED" RED "SORTED," AND YELLOW BORDERS INDICATE THAT TWO ITEMS ARE BEING COMPARED.

which successive items of data are inserted into their correct positions relative to items previously considered. The movie introduces Linear Insertion (shown in Figure 1), the simplest of the insertion sorts; Binary Insertion; and Shellsort (see Figure 2). Exchange sorts interchange pairs of items until the data is sorted. The film demonstrates two n^2 exchange methods, Bubblesort and Shakersort, and one $n \log n$ method, Quicksort (see Figure 3). In selection sorts, the algorithm selects one item in turn and positions it in the correct final position. The movie presents three such methods—Straight Selection, Tree Selection, and Heapsort.

Animation Design Challenges. A problem for viewers in early versions of the film was a lack of consistent visual conventions. Viewers should be able to forget about the technique of presentation and concentrate instead on the content being taught. Without an appropriate set of visual conventions, such as one color to denote “sorted” items and another for items “still to be considered,” they may spend more energy trying to figure out what the picture means than trying to follow the algorithm.

Another central problem was timing. The steps of the algorithms must first be presented slowly, giving time for the narrator to explain what is happening and for the viewer to absorb it. However, once the algorithm is understood, later steps may be boring.

We needed a visually interesting and convincing way to convey the concept of efficiency. We could have used animated performance statistics, but if we wanted to show the algorithms operating on large

Programs execute over time, so they can be vividly represented through animation, portraying how they carry out their processing and how their essential state changes over that time.

amounts of data, we would have had new representation problems. Fitting the desired information legibly onto the screen and compressing the animation into a reasonable time span required designing new methods of portraying the data and illustrating the algorithm’s progress.

More generally, we were faced throughout the film with the problem that totally literal and consistent presentations can be uninteresting. Yet consistency is required so changes made for visual purposes are not interpreted falsely as clues to understanding the algorithm. Being literal and explaining things step by step are required to aid initial understanding, but we also had to add dramatic interest as we presented more advanced material.

Design Solutions. Presenting nine algorithms, grouped into three groups of three, lent itself to a pleasing symmetry. Within each group, we adopted a different set of visual cues while retaining the same underlying conventions. Thus, in each group, one color indicates items “yet to be considered;” a second color, items “already sorted;” and a third, borders around items currently being compared. Whenever items are dimmed and faded into the background, they are “not currently being considered” within the context of the algorithm.

Only the data appears on the screen. There are no pointers, no labels, no gimmicks. Attributes of the data and the algorithm are

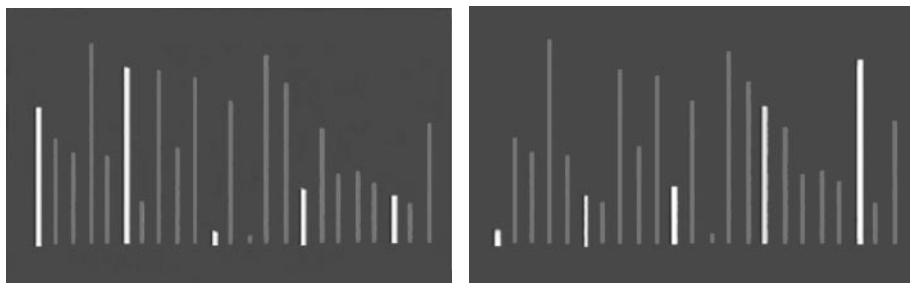


Figure 2. Shellsort

SHELLSORT PERFORMS INSERTION SORTS ON SUBSEQUENCES OF THE DATA SPACED WIDELY APART, QUICKLY MOVING ITEMS CLOSER TO THEIR ULTIMATE DESTINATION. IT THEN PERFORMS INSERTION SORTS ON SUBSEQUENCES OF THE DATA SPACED MORE CLOSELY TOGETHER. IT CONTINUES THIS WAY UNTIL IT FINALLY PERFORMS A LINEAR INSERTION SORT AS THE LAST PASS. BECAUSE ITEMS ARE ALREADY CLOSE TO WHERE THEY BELONG, THIS SORT IS VERY EFFICIENT. THE TWO FRAMES SHOW THE BEGINNING AND END STATES OF THE FIRST PASS, WHICH PERFORMS AN INSERTION SORT ON A SUBSEQUENCE OF THE DATA CONSISTING OF EVERY FIFTH ITEM.

conveyed entirely by these clues, and by the motion of the data, the accompanying narrative, and to a lesser extent the music track, which is not directly driven by the data but conveys the feeling of what is going on. Where necessary, the pace of the sort was slowed to allow time for complex narration and for viewers to digest what is going on. The pace is sometimes speeded up to avoid boredom after the initial explanatory passes.

After presenting all three algorithms of each class, we illustrate their efficiency with line graphs showing numbers of data comparisons and movements and total execution time for sorts of n items, where n ranges from 10 to 500 items. The three graphs distinguish clearly the $n \log n$ from the n^2 algorithms.

Figure 3.

The race of the three exchange sorts



(A) THE QUICKSORT COMPLETES

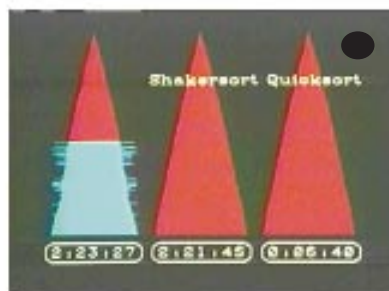
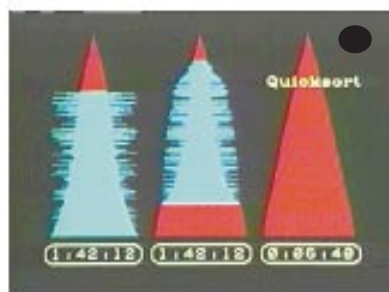
AFTER ROUGHLY 7 SECONDS;

(B) THE SHAKERSORT TAKES MORE THAN 1 MINUTE AND 40 SECONDS TO APPROACH COMPLETION;

(C) SHAKERSORT COMPLETES AT 2 MINUTES AND 21 SECONDS;

(D) THE BUBBLESORT COMPLETES AT JUST OVER 2 MINUTES AND 45 SECONDS.

(BUBBLESORT WORKS FROM THE TOP DOWN; SHAKERSORT WORKS FROM BOTH THE TOP AND THE BOTTOM.)

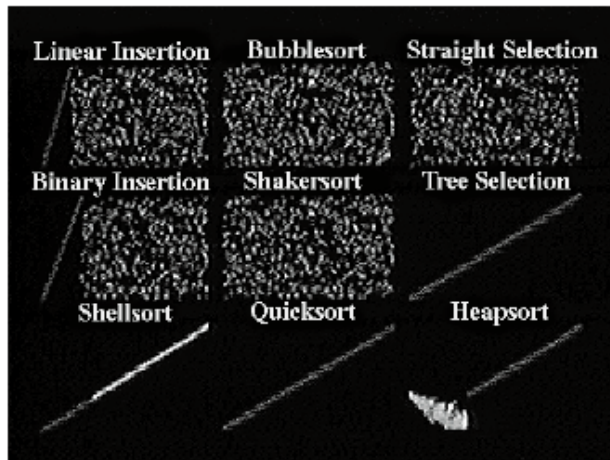
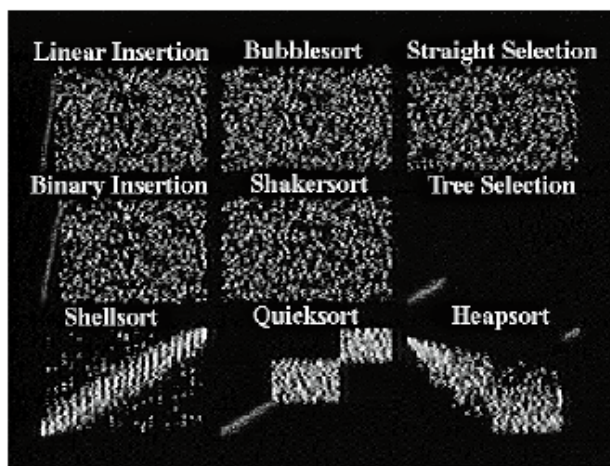


Illustrating algorithm efficiency more dramatically, we then run a “race” of all three techniques on the screen, sorting 250 items of data. Each algorithm is accompanied by a digital clock measuring film time, stopping as soon as the data is sorted, as in Figure 3. Each algorithm’s title appears as soon as its data is sorted. The slowest algorithms take more than 2 minutes, while the $n \log n$ sorts finish in 5 to 15 seconds.

We close with a “grand race” of all nine algorithms, sorting 2,500 items of data each (see Figure 4). Each item of data is represented by a colored dot. The value of the item is represented by its vertical location and its position in the array by its horizontal location. Unsorted data appears as a cloud and sorted

Figure 4. Sorting Out Sorting’s grand race

UNSORTED DATA APPEARS AS A CLOUD; SORTED DATA ARE THE DIAGONAL LINE. THE DIFFERENCES BETWEEN THE $n \log n$ SORTS—SHELLSORT, QUICKSORT, TREESORT, AND HEAPSORT—AND THE n^2 SORTS ARE VISIBLE. NOTICE SHELLSORT’S PUSHING OF THE DATA TOWARD THE LINE, QUICKSORT’S RECURSIVE SUBDIVISIONS, AND HEAPSORT’S STRANGE DATA FUNNEL.



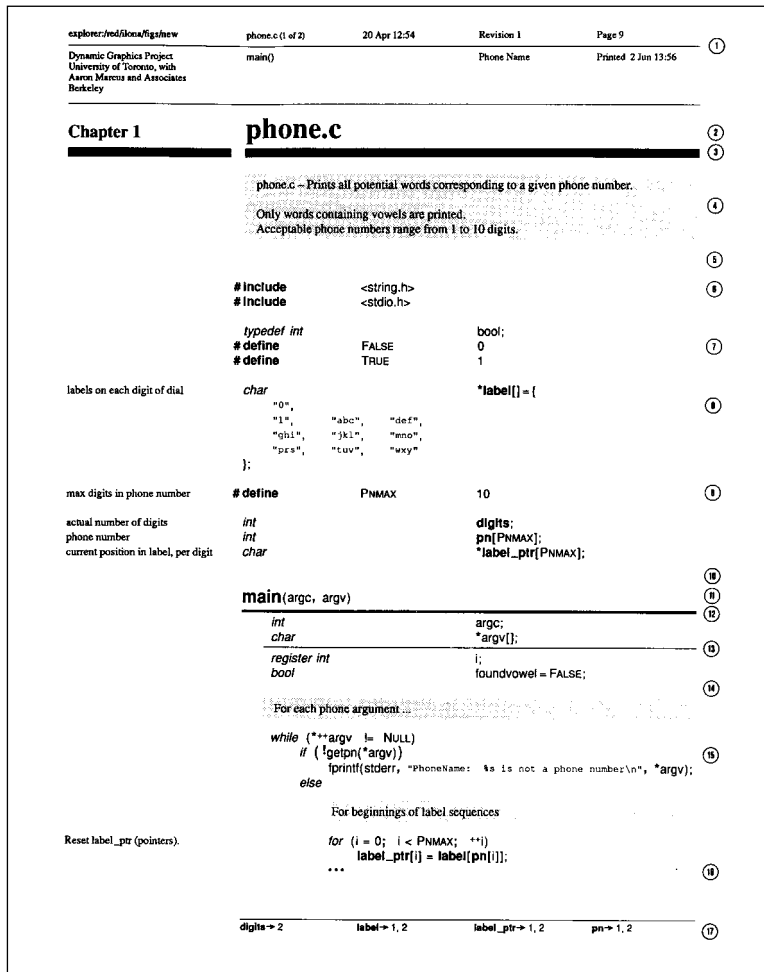


Figure 5. Page 1 of a designed and typeset program [3]

OUTPUT IS ON LOOSELEAF 8.5 x 11 INCH PAGES, EACH SEPARATED INTO FOUR REGIONS—HEADER (1), FOOTNOTE AREA (17), MAIN TEXT COLUMN FOR CODE AND MOST COMMENTS (3, RIGHT), AND MARGINALIA COMMENTS (3, LEFT). EACH FILE IS A SEPARATE “CHAPTER” WITH THE FILENAME AS A LARGE, BOLD TITLE (2). EXTRA WHITE SPACE SEPARATES PROLOGUE COMMENTS AND CODE (5), FUNCTION DEFINITIONS AND DECLARATIONS (10), INDIVIDUAL FUNCTION DEFINITIONS, AND HEADER AND BODY OF FUNCTION DEFINITION (14). CROSS-REFERENCES RELATING USES OF GLOBAL VARIABLES TO THEIR DEFINITIONS ARE FOOTNOTES TO THE SOURCE TEXT (17).

EACH FILE MAY INCLUDE A PROLOGUE COMMENT DESCRIBING THE MODULE’S PURPOSE (4) IN A SERIF FONT ON A LIGHT GRAY BACKGROUND. MARGINALIA COMMENTS ON THE SAME LINES AS SOURCE CODE ARE IN A SMALL-SIZE SERIF FONT IN THE LEFT COLUMN (9, LEFT).

INTRODUCTORY TEXT OF A FUNCTION DEFINITION—THE FUNCTION NAME—IS A “HEADLINE” IN LARGE

data as a diagonal line.

Tree Selection and Quicksort finish in 20 seconds each, the other $n \log n$ algorithms within another 20 seconds. Their sorted data then fades out while the n^2 sorts plod along during the final credits and then also fade out. This final fadeout happens long before they are finished, since it would take another 54 minutes for Bubblesort to complete.

The grand race also illuminates how the algorithms operate. We see how Shellsort moves all the data items close to their final positions, then finishes the job on the final pass. We see the recursive behavior of Quicksort as it picks up rectangular regions of the array and squashes them into a line. We see the peculiar way Heapsort organizes and shapes the data into a funnel as it picks off the successive largest remaining elements.

The film’s 2-minute-30-second Epilogue is an opportunity for review by replaying the entire film at 12 times normal speed, thereby generating visual patterns unique to each method and not obvious at normal speed.

The film goes beyond a step-by-step presentation of the algorithms, commu-

SANS SERIF TYPE (11). A HEAVY RULE IS UNDER THE INTRODUCTORY TEXT OF A FUNCTION DEFINITION (12). A LIGHT RULE IS UNDER THE DECLARATION OF FORMAL PARAMETERS (13).

DECLARED IDENTIFIERS ARE ALIGNED TO AN IMPLIED VERTICAL LINE AT AN APPROPRIATE HORIZONTAL TAB POSITION (7). INITIALIZERS ARE AT REASONABLE TAB POSITIONS, AND PROGRAMMER CARRIAGE RETURNS ARE RESPECTED AS REQUESTS FOR “NEW LINES” (8).

SYSTEMATIC SYNTAX-DIRECTED INDENTATION AND PLACEMENT OF KEYWORDS ARE EMPLOYED (15). SINCE CURLY BRACES ARE REDUNDANT WITH SYSTEMATIC INDENTATION, THE USER MAY SUPPRESS THEM (15). IN CONVENTIONAL PROGRAM LISTINGS, IT IS IMPOSSIBLE, WITHOUT TURNING THE PAGE, TO TELL WHERE A PARTICULAR CONTROL CONSTRUCT (IN THIS CASE, THE *for*) CONTINUES ON THE FOLLOWING PAGE. OUR SOLUTION IS AN ELLIPSIS, IN LINE WITH THE *for*, SIGNIFYING THE FIRST STATEMENT ON THE NEXT PAGE IS AT THE SAME NESTING LEVEL AS THE *for* (16).

nicating an understanding of them as dynamic processes. Seeing the programs in process, running, we see the algorithms in new and unexpected ways. We see sorting waves ripple through the data. We see data reorganize itself as if it had a life of its own. These views produce new understandings difficult to express in words.

Sorting Out Sorting [1] (600 copies sold over the past 15 years) encapsulates in 30 minutes of animation the essence of what written treatments require 30 or more detailed pages to convey. Interviews with students and an informal, unpublished experiment make clear that the film communicates both the substance of the algorithms and the concept of their relative efficiency.

The film was also instrumental in stimulating further work in algorithm animation, most notably by Marc Brown [4], who together with *Sorting Out Sorting* has inspired much of the work in the field. The project also taught us many lessons about algorithm animation, including:

- Significant insights into algorithm behavior can be gained only by viewing the data—if the illustrations and the timing are carefully designed and accompanied by appropriate narration.
- Timing is the key to effective algorithm animation; fancy rendering and smooth motion are not required.
- Straightforward techniques of algorithm animation aid program understanding and therefore should be useful for debugging.

Source Code Presentation: How Should Programs Look?

To debug a program, one must also view and think about source code from different perspectives. We therefore developed techniques for enhancing source code readability and comprehensibility [3, 13].

Program appearance has changed little since the first high-level languages were developed in the 1960s. Unlike symbolic systems, such as circuits, maps, mathematics, and music, we lack sophisticated notations and conventions employing the tools of design and typography, such as typefaces, weights, point sizes, line spacing, rules, gray scale, diagrams, and pictures. Knuth [7] expressed the challenge eloquently:

I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be works of literature. Hence, my title: 'Literate Programming' . . . Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a

We see sorting waves ripple through the data. We see data reorganize itself as if it had a life of its own. These views produce new understandings difficult to express in words.

computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.

Our work has sought to demonstrate and explain how communication about and comprehension of programs can be aided by improving the visual schema and appearance of the programs.

To illustrate, Figure 5 is the first of two pages of a typical short C program that prints an alphabetic equivalent of a numerical phone number.² We explain our design's salient features with reference numbers, so, for example, (1) and (17) refer to the small numbers in circles in the right margin of the program page.

At first glance, it may seem we are talking only about "prettyprinting," but our approach goes significantly beyond prettyprinting in several ways:

- We realized that rich typographic representations with many degrees of freedom allows source code presentation qualitatively different from that of conventional prettyprinting.
- We developed graphic design principles for software visualization and applied them to programming languages.
- We developed experimental variations in an iterative graphic design process to help us derive design guidelines and conventions for program appearance.
- We formalized these guidelines and specifications in a graphic design manual for C program appearance.
- We developed a flexible experimental tool, the SEE visual compiler, to automate production of enhanced C source text. SEE was highly parametric, allowing easy experimentation with novel variations to suit the style preferences of programmers.

²Because this research began in 1982, we focused on paper listings for output. The same approach and principles apply to program formatting for interactive use on workstations.

- We enlarged our scope from the narrow issue of formatting source code to a broader concern with programs as technical publications. We considered the entire context in which code is used, including the supporting texts and notations that make a program a living piece of written communication. Therefore, to help programmers master software complexity, we developed proposals for designing, typesetting, printing, and publishing program books. We also integrated bodies of program text and metatext and incorporated displays of essential program structure, called *program views*.

Program books are needed because typical source text does not itself have sufficient communicative depth. A large real program is an information narrative in which the components should be arranged in a logical, easy-to-find, easy-to-read, easy-to-remember sequence. The reader should be able to quickly find a table of contents to the document, determine its parts, identify desired sections, and find their locations. Within the source text, the overall structure and appearance of the page should furnish clues regarding the nature of the contents. For example, headers and footnotes should reinforce the structure and sequencing of the document.

To illustrate these concepts, Figure 6 consists of four miniature pages from a prototype program book (see also [10, 11]) based on Henry Spencer's implementation of Joe Weizenbaum's Eliza program.

In developing the concept of the program book, we applied the visible language skills of graphic design, guided by the metaphors and precedents of literature, printing, and publishing, to demonstrate that program text should and can be made perceptually and cognitively more accessible and usable. Future generations of students and programmers should be able to read the great works of programming literature, such as a Unix kernel, a Logo interpreter, and a program by Knuth, all typeset in beautiful editions and accessible from a student's physical or virtual bookshelf.

Enhanced program presentation produces listings that facilitate the reading, comprehension, and debugging of the programs. See [3, 10, 11] for theory and experiments³ suggesting that making the interface to a program's source code and documentation intelligible, communicative, and attractive ultimately leads to significant productivity gains.

Interactive Visualization for Debugging: How Does a Program Sound?

For program presentations to be useful in debugging, they cannot always be "canned" by visualization.

³For example, in our study of third-year programming students in which we compared SEE program listings with conventional listings on 200-line programs, the enhanced source code presentation increased the programs' readability by 21% as measured by the subjects' performance on a comprehension test ($p < 0.001$).

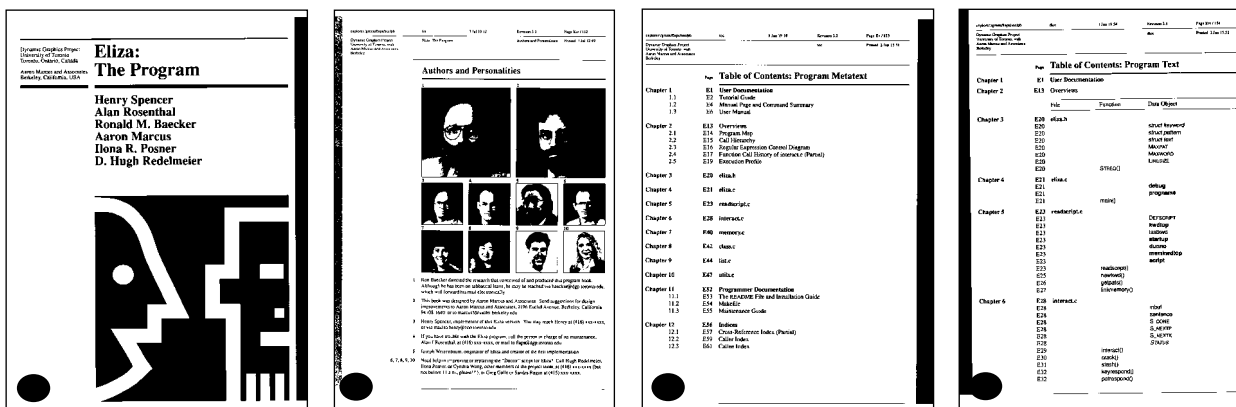


Figure 6. Miniatures of sample pages from a C program book of a new implementation of Eliza [3]

- (A) THE BOOK'S COVER PAGE, INCLUDING TITLE, AUTHORS, AND AN ILLUSTRATION.
- (B) THE AUTHORS AND PERSONALITIES PAGE, INCLUDING KEY PEOPLE IN THE PROGRAM'S DEVELOPMENT AND MAINTENANCE.
- (C) A TABLE OF CONTENTS PAGE SHOWING THE TOP-LEVEL STRUCTURE OF THE ELIZA BOOK. CHAPTER 1 IS USER DOCUMENTATION; CHAPTER 2 IS A SERIES OF DIAGRAMATIC AND GRAPHICAL OVERVIEWS OF ELIZA; CHAPTER 3–CHAPTER 10 IS ELIZA SOURCE TEXT; CHAPTER 11 IS PROGRAMMER DOCUMENTATION; AND CHAPTER 12 IS A SERIES OF INDICES TO ELIZA.
- (D) THE FIRST OF TWO TABLE OF CONTENTS PAGES SHOWING THE ELIZA SOURCE TEXT; EACH FILE APPEARS AS A SEPARATE CHAPTER.

tion designers but must be constructed opportunistically by software developers. To demonstrate the easy interactive specification of custom presentations, we developed a Logo programming environment called LogoMedia [6].

LogoMedia facilitates the visualization of programs with sounds as well as images. Connected to a musical instrument digital interface (MIDI) synthesizer, it can be used to orchestrate acoustic feedback [8]

that informs programmers of key control and dataflow events. Audio has been used in software visualization (e.g., [5]) to augment, enhance, or replace graphical or textual portrayals for several reasons:

- While attending to auralizations, programmers are free to interact simultaneously with the graphical interface (e.g., by scrolling through code listings, inspecting output windows, or editing bug lists).
- As a modality distinct from the typically visual output of today's programs, audio can provide programmers with debugging and profiling feedback without disturbing the integrity of the graphical interface.
- Sound may be a more salient

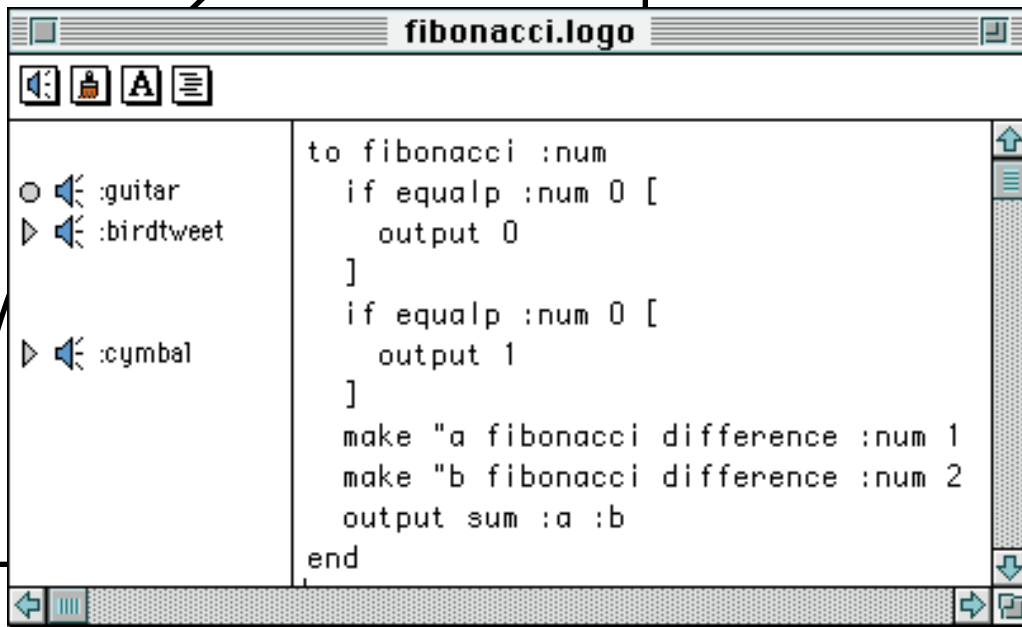


Figure 7. A LogoMedia code window containing a buggy Fibonacci procedure and three audio control probes

THE POP-UP MENU ICONS FOR LOGOMEDIA'S FOUR PROBE TYPES ARE AT THE TOP OF THE WINDOW. THE THREE PROBES ON THE LEFT MARGIN OF THE CODE ARE ASSOCIATED WITH THE LINES IN THE FIBONACCI PROCEDURE⁴ THEY ARE MONITORING. FOR EXAMPLE, THE TWO SYMBOLS AND THE WORD ":guitar" DENOTE THAT A GUITAR NOTE IS ASSOCIATED WITH EACH CALL TO fibonacci. HERE, THE PROGRAMMER IS USING THE DEFAULT CONTROL PROBE ACTION, WHICH IS TO PLAY A NOTE MAPPED TO THE DEPTH OF RECURSION (THE DEEPER THE RECURSION, THE LOWER THE PITCH). THE OTHER TWO CONTROL PROBES IN THE FIGURE PLAY UNMAPPED SOUNDS WHEN EXECUTION REACHES THE FIRST AND SECOND OUTPUT STATEMENTS. (A LOGO OUTPUT COMMAND TERMINATES THE PROCEDURE CALL.) WHEN fibonacci IS RUN, THE PROGRAMMER HEARS A RAPID SERIES OF GUITAR NOTES OF DECREASING PITCH AS THE PROGRAM CALLS ITSELF RECURSIVELY ON DECREASING VALUES OF num. A BIRD TWEET IS THEN HEARD, INDICATING THAT THE ARGUMENT FOR ONE OF THE CALLS HAS REACHED 0. HOWEVER, BECAUSE OF A BUG IN THE PROCEDURE (THE SECOND BASE CASE SHOULD CHECK FOR A VALUE OF 1, INSTEAD OF 0), THE GUITAR NOTES CONTINUE PLAYING INDEFINITELY WITHOUT THE CYMBAL SOUNDING.

⁴The last few lines of this example could be expressed more elegantly using the statement `output sum fibonacci difference :num 1 fibonacci difference :num 2`. However, the intermediate values a and b are useful for illustrating auralization techniques.

representation for certain types of program information than graphical forms. Two examples include repetitious patterns in control flow and nonlinear sequences of variable values.

- As a new output modality complementing the visual interface, audio offers an attractive design solution for programming on smaller devices with limited screen space, such as personal digital assistants.

Audio Probes. Audio renditions, or "auralizations," of LogoMedia programs are specified via "probes" programmers attach to their source code unobtrusively, that is, without modifying the code. Programmers can configure or design probes to turn on synthesized musical instruments, play back sound samples, or adjust a sound's pitch or volume. LogoMedia supports two types of probes: *control*

probes for monitoring execution flow and *data probes* for tracking variable values. A graphical interface was designed to make it easy to attach these probes to running programs.

Programmers install control probes by selecting expressions they wish to monitor in one of LogoMedia's code windows (see Figure 7) and then choosing probes from a pop-up menu at the top of the window. The menu options specify which MIDI instruments or effects to turn on or off prior to executing a line of code. In addition to the audio probe, LogoMedia offers other probe types that provide a consistent interface for creating graphical feedback as well as audio. These include graphics probes for assisting animation, text probes for printing tracing messages, and "generic" probes that allow users to execute their own Logo expressions for generating completely customized visualizations. Three examples of audio probes for debugging a program to compute the Fibonacci series are shown in Figure 7.

In addition to using control probes, programmers can install data probes for monitoring the values of variables through a Probe Sheet window (see Figure 8). In the Probe Sheet, programmers enter arbitrary Logo expressions as "triggers" for audio feedback. Probes can then be linked to the triggering expressions using pop-up menus. Like control probes, data probes can generate MIDI notes and/or adjust properties of sounds already being played. These actions occur whenever a probe's triggering expression is modified. Figure 8 shows two audio data probes for monitoring variables in the Fibonacci program.

User Evaluation of LogoMedia.

We conducted an observational ethnographic study to observe how programmers could make use of sound while coding and debugging and to gain feedback on LogoMedia's interface for specifying auralizations. Three study subjects—with modest Logo or Lisp programming experience and varied musical backgrounds—participated in a series of three sessions, each lasting approximately two hours. The first session was a review of basic Logo programming constructs and an introduction to LogoMedia's audio

probes. In the second session, subjects composed procedures for the Game of Life population simulation algorithm invented by John H. Conway; they then used sound to help evaluate their code. Finally, in the third session, subjects were asked to fix a different implementation of Life and encouraged to use auditory feedback for debugging purposes. A think-aloud protocol was used to elicit comments from subjects as they worked.

We were particularly interested in how subjects would use sound in their "test runs," that is, Logo evaluations that checked for bugs or tested a theory. On average, the subjects conducted 43 such test runs per session; in 55% of them, they used auditory feedback. Without explicit suggestions from the experimenter, they developed a variety of interesting ways to extract information from the test runs.

Subjects generated audio feedback via control and data probes in order to complement visual feedback,

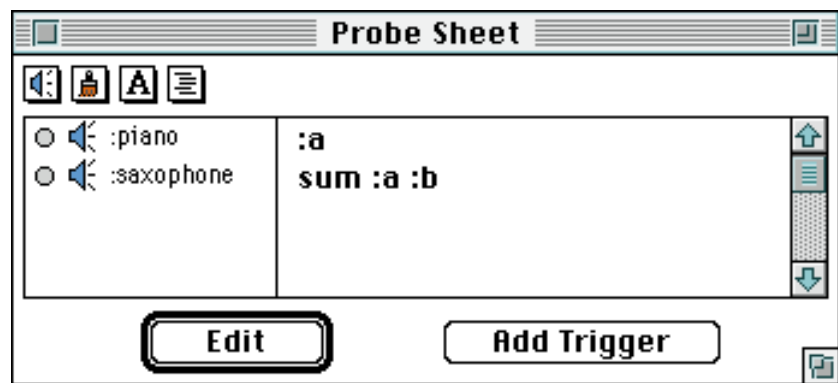


Figure 8. The LogoMedia Probe Sheet and two audio data probes

THE FIRST DATA PROBE CAUSES A PIANO NOTE TO PLAY WHEN THE LOGO VARIABLE "a" CHANGES; THE SECOND GENERATES A SAXOPHONE SOUND WHEN THE SUM OF "a" AND "b" CHANGES. THESE DATA PROBES PLAY DESCENDING SERIES OF PIANO AND SAXOPHONE NOTES AS THE ARGUMENTS TO RECURSIVE CALLS TO `fibonacci` ARE DECREMENTED. THESE AUDIO CUES MIGHT HELP THE PROGRAMMER REALIZE THAT THE PROBLEM IS NOT THAT THE DECREMENT OPERATIONS ARE FAILING BUT THAT THE SECOND BASE CASE IS FLAWED.

identify errors in the code, and verify their bug fixes. They chose "iconic" sounds, such as an explosion and a bell, to denote important events, such as the birth or death of a game piece. More musical sounds, such as that of a piano, were used to indicate such ongoing events as traversal of the simulation game board.

Audio control probes were used heavily by subjects to test whether execution reached a certain part of their code and to help answer other fine-grained flow

control questions. To assess the values of variables at particular points in programs, they installed audio control probes at these points to map characteristics of the data to pitch; subjects also used audio data probes to monitor arbitrary changes to variables by mapping their values to sound. Trends were evident when state changes occurred rapidly in some patterned way. Subjects also used data probes to monitor acoustical variables, such as loop indices, lists, and counters (e.g., for the numbers of neighboring pieces in a region of Life's board). While listening to their programs execute, subjects took advantage of their unburdened visual processing abilities to manipulate the graphical interface. Listening activities included scrolling through code, manipulating windows, and adjusting the audio portrayal.

Just as the timing of visualizations was important for students learning from *Sorting Out Sorting*, the speed of auralizations was critical for subjects trying to make sense of acoustic feedback. Typically, they would create audio probes, run a program, adjust the speed of execution using LogoMedia's speed controller, and then run the program again. Full execution speed seemed suitable for noticing general trends, as when subjects were tracking down infinite loops and wanted to know which procedures were running too long. However, when the relative pitch was more important, subjects tended to slow execution by a factor of two or more in order to hear changes from one value to the next more distinctly. One subject seemed to consider execution speed very important; on average, he adjusted the speed controller once every four test runs in the final two-hour session.

Subjects, on their own initiative, began developing an acoustic vocabulary for describing their running programs. For example, one said a loop in his program was "clicking." Another, when describing a procedure he had just fixed for calculating the next generation in Life, commented that it "sounds like it's counting right now."

Subjects also encountered a few problems with LogoMedia's auralization facilities. For example, sometimes they wanted to track the dataflow in the program, but because of the functional nature of Logo, the datum they wished to monitor was not represented by a variable and could not be entered in the Probe Sheet. They worked around this limitation by assigning the results of Logo expressions to temporary variables.

Another problem was avoiding cacophony. Subjects complained that certain simultaneous sounds, such as the piano and the guitar, tended to merge into one. Even when more distinctive sounds were

Effective representation and presentation aid thought, the management of articulate expression and complexity, and problem solving.

used, such as the piano and the explosion, too many audio probes tended to cause confusion. Future versions of LogoMedia should allow users to temporarily disable probes to reduce noise. Subjects also suggested that LogoMedia offer more specific domain-oriented sounds and allow them to play back their own voices.

Because there were only three subjects, our results are suggestive, not definitive. Yet several incidents suggest that audio feedback was more than a novelty for these users. For example, after identifying the missing loop index incrementer in Life, one subject said, "That was neat. That was very helpful. I think if I hadn't had the sound, I would still be banging away at this."

More subtle but perhaps more convincing evidence can be seen in how all three subjects reacted to the system failures that caused their audio probes to be lost. This happened five times during the study, at least once to each subject. In four cases, the subjects elected within 5 minutes of LogoMedia's being restarted to reinstall the probes almost identically to the way they were before the failure—without prompting from the experimenter or the written instructions.

Conclusions


The three software visualization approaches described here are useful for debugging. Carefully crafted algorithm animations can show how programs work; enhanced typographic representations can improve the readability and comprehensibility of source code; and an interactive environment can let programmers specify software visualizations, including audio portraits of running programs.

A number of implications for the design of debugging technology can be derived from our work:

- The need for graphic design expertise in developing visualization technology;
- The importance of variety, cleanliness, and simplicity in the choice of visual representations;
- The desirability of using sounds as well as images;

- The importance of speed controls for program playback;
- The desirability of being able to attach debugging probes unobtrusively to source code; and
- The need for tools for enhanced static display of source code and for dynamic display of program execution.

These tools must be powerful enough so programmers can configure effective special-purpose visualizations with little effort in response to particular bugs.


Debugging environments can usefully employ visualization tools and techniques. To develop what is needed, computer science must learn and apply the lesson that graphic design (see [9]) teaches so vividly: Form matters and is not just an issue of aesthetic appeal. Effective representation and presentation aid thought, the management of articulate expression and complexity, and problem solving. As debugging is a particularly challenging form of problem solving, software visualization should play an increasingly important role in future programming environments. 



Department of Computer Science

Summer ReNEWal 1997

Six Ways to Stay Current - Leading Edge Technology Seminars

<p>Security and Intrusion Detection for Network Infrastructure</p> <p>Felix Wu May 19-23</p>	<p>Design Principles for Concurrent Systems</p> <p>Rance Cleaveland June 2-6</p>	<p>The Technology of Distributed Multimedia</p> <p>Douglas Reeves June 9-13</p>
<p>Knowledge-Based Multimedia Learning Environments</p> <p>James Lester Patrick FitzGerald June 16-20</p>	<p>Object-Oriented Languages and Systems</p> <p>Vicki Jones Edward Gehringer Jonathan Rossie June 23-27</p>	<p>Cooperative Information Systems: Agents Meet Databases</p> <p>Munindar Singh July 7-11</p>
<p>For more information please contact</p> <p>Ms. Bobbi Baird Industrial Extension Service, NCSU Raleigh, NC 27695-7902 Phone 919-515-3954 Fax 919-515-1325 bobbi_baird@ncsu.edu http://www.csc.ncsu.edu/summer_renewal_97</p> 		

ACKNOWLEDGMENTS

Many individuals, all acknowledged in [2, 3, 6], contributed to this research. We are especially grateful to Michael Arent, Ilona Posner, Hugh Redelmeier, Alan J. Rosenthal, and David Sherman and to the Advanced Research Projects Agency (U.S.) and the Natural Sciences and Engineering Research Council (Canada) for financial support.

REFERENCES

1. Baecker, R.M. (assisted by Sherman, D.) *Sorting Out Sorting*, color/sound film, University of Toronto, 1981. Distributed by Morgan Kaufmann, San Francisco.
2. Baecker, R.M. *Sorting Out Sorting: A case study of software visualization for teaching computer science*. In *Software Visualization: Programming as a Multimedia Experience*, J. Stasko, J. Domingue, M. Brown, and B. Price, Eds. MIT Press, Cambridge, Mass., 1997.
3. Baecker, R.M., and Marcus, A. *Human Factors and Typography for More Readable Programs*. ACM Press, Addison-Wesley, Reading, Mass., 1990.
4. Brown, M.H. *Algorithm Animation*. MIT Press, Cambridge, Mass., 1988.
5. Brown, M.H., and Hershberger, J. Color and sound in algorithm animation. *IEEE Comput.* 25, 12 (1991), 52-63.
6. DiGiano, C.J. Visualizing program behavior using non-speech audio. M.S. Thesis, Univ. of Toronto, 1992.
7. Knuth, D.E. Literate programming. *Comput. J.* 27, 2 (May 1984), 97-111.
8. Kramer, G., Ed. *Auditory Display: Sonification, Audification, and Auditory Interfaces*. Addison-Wesley, Reading, Mass., 1994.
9. Marcus, A. *Graphic Design for Electronic Documents and User Interfaces*. ACM Press, New York, and Addison-Wesley, Reading, Mass., 1992.
10. Oman, P.W. and Cook, C.R. The book paradigm for improved maintenance. *IEEE Software* (Jan. 1990), 39-45.
11. Oman, P.W. and Cook, C.R. Typographic style is more than cosmetic. *Commun. ACM* 33, 5 (May 1990), 506-520.
12. Price, B.A., Baecker, R.M., and Small, I.S. A principled taxonomy of software visualization. *J. Visual Lang. Comput.* 4, 3 (Sept. 1993), 211-266.
13. Stasko, J., Domingue, J., Brown, M., and Price, B., Eds. *Software Visualization: Programming as a Multimedia Experience*. MIT Press, Cambridge, Mass., 1997.

RON BAECKER (rmb@dgp.toronto.edu) is Professor of Computer Science, Electrical and Computer Engineering, and Management, and Director of the Knowledge Media Design Institute at the University of Toronto. CHRIS DIGIANO (digi@acm.org) recently earned a Ph.D. in computer science from the University of Colorado at Boulder. AARON MARCUS (Aaron@amanda.com) is President of Aaron Marcus and Associates, Inc., Emeryville, Calif., a user interface design and development firm.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© ACM 0002-0782/97/0400 \$3.50