Sorting Out Sorting: A Case Study of Software Visualization for Teaching Computer Science

Ronald Baecker
University of Toronto

Appears in *Software Visualization: Programming as a Multimedia Experience.* MIT Press, 1998, 369-381.

Like professional programmers, teachers and students of computer science frequently use pictures as aids to conceiving, expressing, and communicating algorithms. Instructors used to cover blackboards and themselves with chalk in drawing intricate diagrams of data structures and control flow, yet often made errors such as improperly estimating the space needed for a proper layout. Students try to improve their visualization of a program's behaviour by sketching representations of nesting of procedures, scope of variables, memory allocation, pointer chains, and organization of record structures.

Unfortunately, a program's behaviour cannot be described by a static drawing; it requires a dynamic sequence. We must trace control flow, bind variables, link pointers, and allocate memory. We must execute processes which mimic those of the machine. It is difficult for us to enact these dynamic sequences directly. Our drawings are inaccurate. Our timing is bad. We make major mistakes, such as skipping or rearranging steps. Thus it would be useful to have animation sequences portraying the behaviour of programs constructed automatically as a by-product of their execution, and therefore guaranteed to portray this execution faithfully.

Animation is a compelling medium for the display of program behaviour. Since programs are inherently temporal, executing through time, they can be vividly represented by an animated display which portrays how they carry out their processing and how their essential state changes over time. Furthermore, many algorithms employ repetitive computations, whether expressed iteratively or recursively. These can be viewed efficiently when displayed as a motion picture. In so doing, we can perceive structure and relationships of causality, and ultimately infer what the program is doing.

Software visualization can therefore be a powerful tool for presenting computer science concepts and assisting students as they struggle to comprehend them. This chapter traces the author's early investigations of this concept, presents a detailed description of the contents and development of a successful 30-minute teaching film *Sorting Out Sorting*, and outlines other work in pedagogical uses of program animation since *Sorting Out Sorting*.

Animating programs for pedagogical purposes is not a trivial endeavor. There are numerous intricate details in most computer programs. To be effective, algorithm animation must abstract or highlight only the essential aspects of an algorithm. We must decide which program text and which data to represent, how they are to be visualized, and when to update their representations during the execution of a program, Most importantly, we must try to enhance relevant features and suppress extraneous detail, to devise clear, uncluttered, and attractive graphic designs, and to choose appropriate timing and pacing.

Early Work

In 1971, having completed GENESYS (Baecker 1969a,b; Baecker, Smith, and Martin, 1970), a pioneering interactive computer animation system for artists, I turned my attention to the role of computer animation for computer science (Baecker, 1973). I was surprised that computer scientists had not reacted more enthusiastically to Ken Knowlton's (1966a,b) dramatic early computer animation explaining the instruction set of a low-level list processing language. Two notable exceptions were Bob Hopgood (1974) and Kellogg Booth (1975), whose early films are reviewed in the history chapter by Baecker.

We carried out a number of program animation experiments in the next seven years. Students in computer graphics classes developed animations of specific algorithms, such as bubble sort, recursive merge sort, hash coding, and hidden line elimination. Ed Yarwood (1974) explored the concept of *program* *illustration*, focusing specifically on the integration of program source text with diagrams of program state. Several students built extensions to language processors to aid the animation of programs in specific languages such as Logo and PL/I (Baecker, 1975). Finally, in 1978, we decided to use this experience to produce a teaching film on the subject of *sorting algorithms*.

Sorting Out Sorting

From 1978 to 1981 we produced a 30-minute colour sound film, entitled *Sorting Out Sorting*, which uses animation of program data coupled with an explanatory narrative to teach nine different internal sorting methods. The film, now primarily distributed as a videotape (Baecker, 1981), was generated with a 16mm computer-output film recorder.

Sorting Out Sorting explains the nine sorting algorithms sufficiently well so that a student who has watched the animation carefully could program some of them herself. It also illustrates the differences in efficiency of the various algorithms.

The movie has been used successfully with computer science students at various levels. As a motivational aid, and to introduce the concept of the efficiency of different solutions to the same problem, it is shown in introductory computer science courses at the university, community college, and high school level. As an explanation of solution methods, it is shown in first or second courses on programming. It can also be used in introductory courses on data processing, algorithms, or complexity.

Internal Sorting Methods

Internal sorting methods are algorithms for rearranging items within a data structure into some predefined order. Typically this order is that of increasing numerical value, decreasing numerical value, or alphabetical order of a field within the data structure.

There are over one hundred internal sorting algorithms (Knuth, 1973; Lorin, 1975; Wirth, 1976). All have both advantages and disadvantages. Typical tradeoffs are between the algorithm's *difficulty of programming* and *complexity*, or *speed of execution*, and between its *time* and *space* requirements.

Internal sorting methods compare items with other items to determine if they are in the correct order or if items need to be moved. Items are then moved zero or more times until they reach their final and correct positions. Once all items have reached their final positions, the data is sorted and the algorithm is finished.

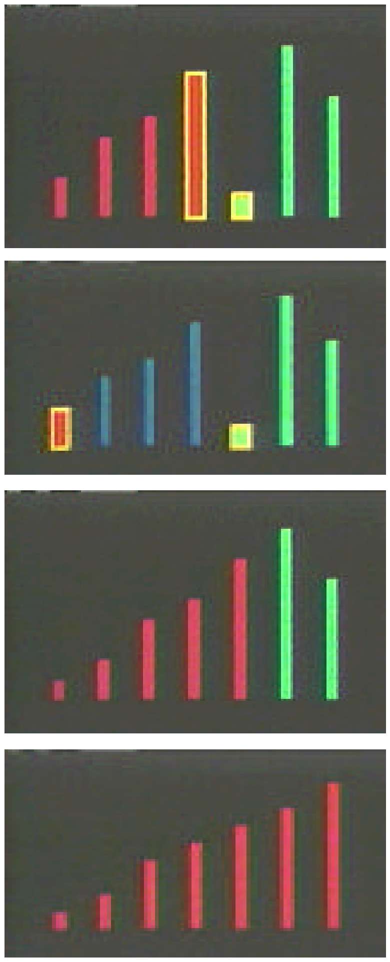


Figure 1a-d. (a is at the top.) Linear Insertion: a) first comparison of the 4th pass, with the first 4 items already correctly ordered; b) final comparison of the 4th pass; c) end of the 4th pass, after the 5th item has been moved to the front; d) data is sorted. Colours (shown here as gray scale) denote "unsorted" and "sorted," i.e., in the correct position thus far. Borders indicate that two items are being compared.

Execution time is therefore primarily determined by the times required for *data comparisons* and for *data movements*. The execution time of internal sorting algorithms over *n* items of data is, in most cases, proportional to either n^2 or *n log n*. As n becomes large, differences between the times required become very significant. However, the simplest algorithms to design, program, and understand are those whose execution time is a function of n^2 .

The film deals with three distinct groups of algorithms — the *insertion sorts*, the *exchange sorts*, and the *selection sorts*. Other techniques, such as the *merge sorts* and the *distribution sorts*, are not covered. Our treatment is closest in content and spirit to that found in Section 2.2 of Wirth (1976).

Structure and Content of the Film

A program may be viewed as a mechanism for transforming one set of data into a new set of data. If we consider a program's state to be determined by its data, then one way of animating and portraying the program is to show how the data is transformed over time. By viewing these sequences of transformations, or possibly several such sequences resulting from different initial sets of data, one can induce the algorithm upon which the program is based.

For example, assume that we wish to order a single array of numerical data into increasing order. We can portray each data item as a vertical bar (Fig. 1), whose height is proportional to the value of the item. Initially, the heights of successive items will vary upwards and downwards. Successive steps of a sorting method will produce rearrangements of the data, until ultimately we should have the elements arrayed in order of increasing height with the smallest one on the left and the largest one on the right.

The movie begins with the *insertion sorts*, in which successive items of data are inserted into their correct position relative to items previously considered. The process is analogous to picking up cards of a bridge hand and inserting them into their correct positions relative to the cards already in one's hand.

The Linear Insertion Sort (Fig. 1) is the simplest of the insertion sorts. For each new item, we scan through the array sorted thus far, looking for the correct position; having found it, we move all the larger items one slot to the right and insert the new item. The Binary Insertion Sort speeds up this technique by using a binary search to find the item's correct position.

In the Shellsort (Fig. 2), we first perform insertion sorts on subsequences of the data spaced widely apart, thus moving items closer to their ultimate destination more quickly. We then perform insertion sorts on subsequences of the data spaced more closely together. We continue in this way until we do as the final pass a regular insertion sort. Because items have already been moved close to where they belong, this pass is extremely efficient.

In the *exchange sorts*, we interchange pairs of items until the data is sorted.

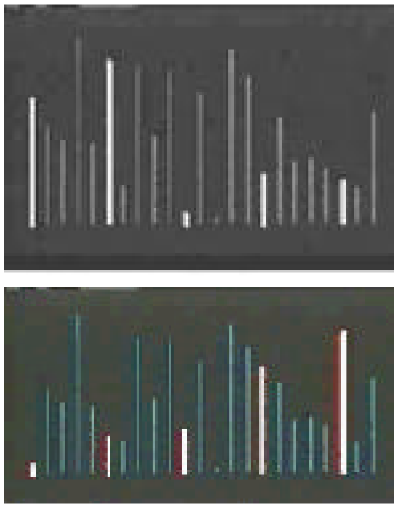


Figure 2a-b. Shellsort. The two frames show the beginning and end states of the 1st pass, which performs an insertion sort on a subsequence of the data consisting of every 5th item.

In the Bubblesort (Fig. 3), we pass through the data from one end to the other, interchanging adjacent pairs of items which are ordered incorrectly relative to each other. Each such pass "bubbles" one more item into its correct position, as for example the smallest items shown at the top in Figure 3. The Shakersort improves on this technique by alternating passes in both directions, and by keeping track of when and where no exchanges were made in order to reduce the number of comparisons on future passes.

The Quicksort (Fig. 4) selects an item at the beginning of the data (the "pivot"), and proceeds by exchanging items from that end that are larger than the pivot with items from the other end that are smaller than the pivot. The pivot is then moved between the two sets of data, so that it is in its correct final position, correctly dividing the set of smaller items from the set of larger items. The Quicksort is then applied recursively to each set. The Quicksort is one of the most efficient of those presented in the film.

The *selection sorts* are those which the algorithm selects, one by one, the data items and positions them in the correct order.

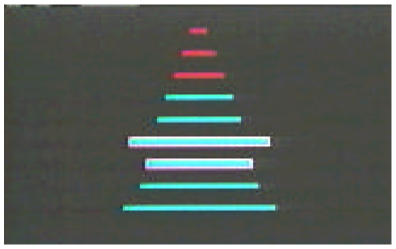


Figure 3. Bubblesort. The two highlighted items are about to be swapped. The top three items have reached their correct positions.

In the Straight Selection Sort, the data is scanned for the smallest item, which is then inserted, with a single data movement, into its final position in the array. Each pass selects the next smallest item from the remaining data. Tree Selection significantly reduces the number of comparisons by organizing the data into a tree, at the cost of requiring more storage.

Heapsort (Fig. 5) preserves most of Tree Selection's efficiency gains without using extra storage. It does this by repeatedly organizing the data into a special kind of binary tree called a *heap*, in which each parent is greater than its children, and then moving the top of the tree into its correct final position.

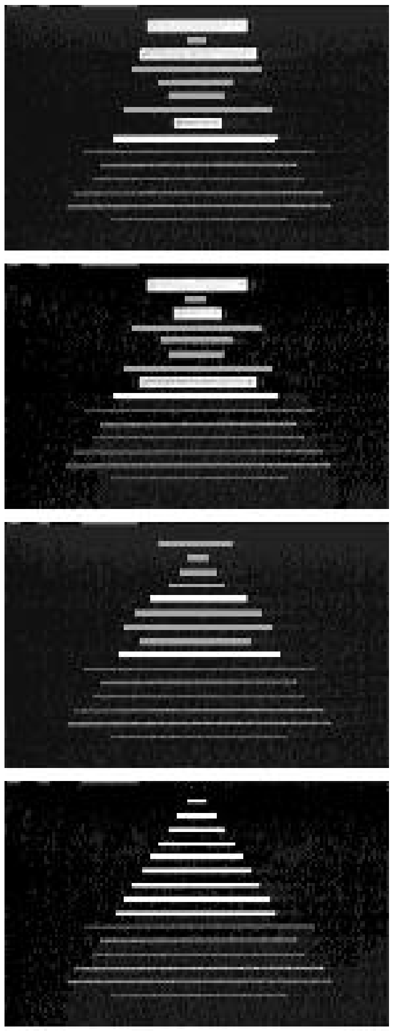


Figure 4a-d. Quicksort: a) The 9th topmost item is in its correct position. The top item is the new pivot. We have found an item larger than it (the 3rd), and one smaller (the 8th), and interchange them to achieve b). Soon, we reach c) in which this pivot has been moved into its correct final position (#5). In d), the top 9 elements have been sorted; we will begin again recursively on the bottom 6.

Problems

A problem in early drafts of the film was a lack of consistent visual conventions. The student who is presented with the animation of several algorithms at once should be able to forget about the *technique* of presentation and concentrate instead on what is being taught. Without an appropriate set of visual conventions, such as one colour to denote "sorted" items and another for "items still to be considered," the viewer may spend more energy trying to figure out what the picture means than she will expend in trying to follow the algorithm.

A central problem is that of *timing*. The steps of the algorithms must first be presented slowly, to give time both for the narrator to explain what is happening and for the student to absorb it. However, once the algorithm is understood, later steps may be boring. This is particularly true in the case of the insertion sorts, which appear to slow down as they go along, whereas the exchange and selection sorts begin slowly and appear to speed up towards the end.

We needed a visually interesting and convincing way to convey the message that some simple algorithms which are easy to code are nonetheless not appropriate for sorting large data sets. One way to do this is by means of animated performance statistics. Yet if we also wish to do this by showing the algorithms operating upon large amounts of data, then we have new representation problems. To fit the desired information legibly onto the screen and to compress the animation into a reasonable span of time requires the design of different methods of portraying the data and different conventions for illustrating the progress of the algorithms.

To summarize, we are faced, throughout the film, with the problem that totally literal and consistent presentations can be boring. Consistency is required so that changes made for visual purposes not be interpreted falsely as clues relevant to understanding the algorithm. Being literal and explaining things step-by-step is required to aid initial understanding, but we must go beyond this to add visual and dramatic interest as we present more advanced material.

Solutions

The presentation of nine algorithms, grouped into three groups of three, lends itself to a pleasing symmetry. We present each group as a separate act of the film. In each case, we present the algorithms within each group in increasing order of efficiency, and hence increasing order of complexity of explanation.

With each group, we adopt a different set of visual cues, while retaining the same underlying conventions. Thus, in each group, one colour is used to indicate items “yet to be considered”; a second colour denotes those items which are “already sorted”; and a third is used to form borders around items which are currently being compared. Whenever items are dimmed and faded into the background, they are “not currently being considered” within the context of the algorithm.

The items themselves are represented by vertical bars in the first group; by horizontal, centered bars in the second group; and by single-digit numbers in the third. In each case the final increasing order attained by the algorithm is from left to right or from top to bottom.

Only the data appears on the screen. There are no pointers, no labels, no gimmicks of any kind. Attributes of the data and processes affecting them are conveyed entirely by the visual clues described in the last two paragraphs, by the motion of the data, by the accompanying narrative, and to a lesser extent by the music track, which is not directly driven by the data but conveys the feeling of what is going on.

Each of the nine parts begins with an animation sequence of the title of the algorithm. The letters of the title appear initially in a scrambled order, and are then rearranged by the algorithm about to be taught until they spell the title correctly. The same colour conventions apply to “sorted items” and “yet to be sorted items” as will apply later. The whole process takes from ten to twenty seconds. It is not intended that the first-time viewer be able to understand from this sequence how the algorithm operates. Yet it does provide a feel for the flow of the sorting method, and for the order in which the items become sorted.

There follows a presentation of the algorithm itself, on a sufficiently small and well-chosen set of data to illustrate at a slow pace how the method works. Where necessary, the pace of the sort is decreased to allow time for complex narration and for the viewers to digest what it is going on. The pace is sometimes increased after a few initial passes have provided a clear explanation and the scene starts to become boring.

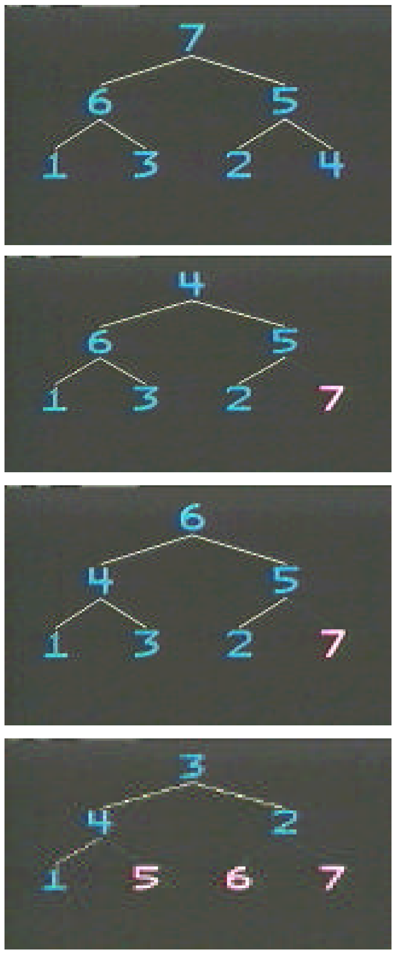


Figure 5a-d. Heapsort: a) The data is organized into a heap, in which every parent item is greater than its children. We then move the top, largest item into its correct position at the end of the array, shown in b). We then need to fix up the tree so it once again is a heap, shown in c). A somewhat later stage is shown in d).

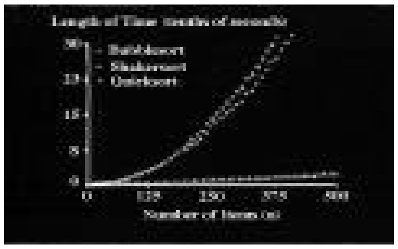


Figure 6. Total execution time of the three Exchange Sorts. Each curve plots growth in execution time as the size of the data set grows. The difference between the two n^2 sorts, Bubblesort and Shakersort, and the n log n Quicksort, represented by the single low curve on the graph, is evident.

After all three algorithms of each class have been presented, we illustrate their efficiency with three line graphs, comparing their efficiency on sorts of *n* items, where *n* ranges from 10 to 500. The three graphs, presented in succession, show number of data comparisons, number of data movements, and actual execution time of a DEC PDP-11/45. Rather than being static, each graph begins with just the labeled axes and "grow" three coloured lines, one for each algorithm (Fig. 6). This technique permits the narrator to focus on each algorithm in turn, and comment on why its line in the graph is shaped as it is. It also depicts clearly the difference between the *n log n* and n^2 algorithms.

To illustrate the difference in efficiency even more dramatically, we then run a "race" of all three technique simultaneously on the screen, sorting 250 items of data. Each algorithm is accompanied by a digital clock measuring film time, which stops as soon as the data is sorted (Fig. 7). A title for each algorithm appears as soon as the data is sorted. The slowest algorithms take over two minutes to run, while the *n log n* sorts are finished in five to fifteen seconds.

After all three groups of sorts have been presented, we close with a "grand race" of all nine algorithms, sorting 2500 items of data each (Fig. 8). Each item of data is represented by a coloured dot. The value of the item is represented by its vertical position, and its position in the array by its horizontal position. Thus unsorted data appears a cloud, and sorted data appears as a diagonal line.

The fastest algorithm, Tree Selection and Quicksort, finishes in 20 seconds each; the other *n log n* algorithms within another 20. Their sorted data then fades out, leaving room for the final credits, while the n^2 sorts plod along, until they, too, fade out. This happens long before they are finished, for, as the narrator notes, it would take another 54 minutes for Bubblesort to complete.

The grand race not only illustrates performance, but illuminates the algorithms. We see how Shellsort moves all the data close to its final position, then finishes the job on the final pass. We see the recursive behaviour of Quicksort as it picks up rectangular regions of the array and squashes them into a line. We see the peculiar way in which Heapsort organizes and shapes the data into a funnel as it picks off successive largest remaining elements.

As an Epilogue to the film, we replay the entire film at 12 times normal speed. This provides an opportunity for review, and shows visual patterns unique to each method that are not obvious at normal speed.

The Success of Sorting Out Sorting

Sorting Out Sorting has been both successful and influential. It is a significant contribution to the pedagogical tools available for teaching sorting methods to computer science students. It encapsulates in 30 minutes of usually vivid and occasionally compelling imagery the essence of what written treatments require 30 or more detailed pages to convey. Interviews with students and an informal, unpublished experiment make it clear that the film communicates effectively both the substance of the algorithms and the concept of their relative efficiency.

More than 600 copies have been sold over the past 15 years, mostly by word-of-mouth and with no effective marketing. I still routinely encounter individuals whom I have never met before who are effusive in their praise of the film. Two letters written to me are particular interesting:

> "I was impressed by the amount of careful thought that was evident in all aspects of the production; the different presentations were always dramatically well-timed, and the visual logic of each section flowed well... the film said visually exactly what needed to be said..." (Scott Kim, 20 July 1981)

and

> "I am profoundly deaf... I was not able to understand ANY of the voice-over commentary on the film. I could hear someone talking; I could hear music, and beeps; but I could not understand WHAT was being said. Interestingly enough, this was not any particular disadvantage. I felt that I could understand most of the film... without hearing any of the commentary." (Bev Biderman, 12 March 1982)

The film was also instrumental in stimulating further work in algorithm animation, most notably that of Marc Brown (1988, also see chapter by Brown), which together with SOS in turn inspired much of the work in the field.

The film also goes beyond a step-by-step presentation of the algorithms, communicating an understanding of them as *dynamic processes*. We can see the programs in process, running, and we therefore 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 which are difficult to express in words.

The film does have weaknesses. The typography is atrocious. Colour is mediocre. Timing is not always optimal, for it is hard to find the right speed for a diverse audience. The film stresses average efficiency too strongly, ignoring best cases and worst case analysis, and also the subtleties that enter into a real-

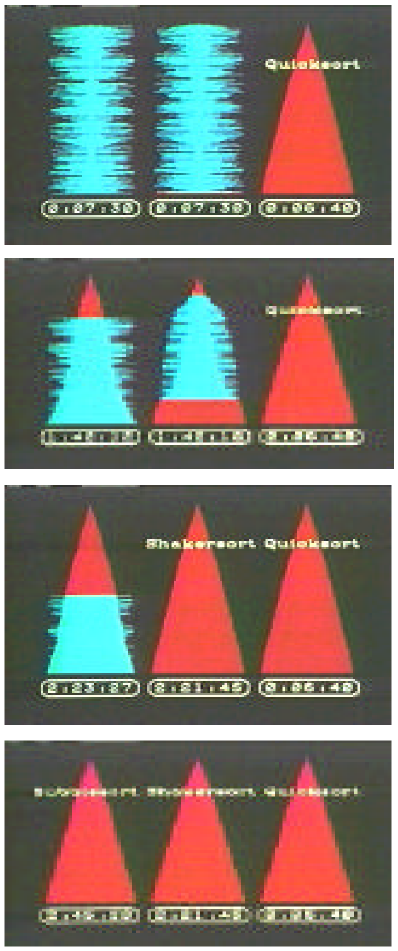


Figure 7a-d. The race of the three Exchange Sorts. a) The Quicksort completes after roughly 7 seconds. b) It then takes over 1 minute 40 seconds for the Shakersort to approach completion. c) At 2 minute 21 seconds, it completes. d) The Bubblesort finally completes at just over 2 minutes 45 seconds. Notice that Bubblesort works from the top down, and Shakersort works from both the top and the bottom.

world choice of technique for a particular problem. Pedagogically, it is regrettable that it omits merge sorts and distribution sorts. Yet it works, and works very well, even today, 15 years later.

Work on the film has taught us a number of lessons about algorithm animation:

• Effective symbolism depends upon the size of the subject and its scale within the total composition, and the context within which the image is displayed.
• Significant insights into algorithm behaviour can be gained while only viewing the data, if the illustrations and the timing are designed carefully, and are accompanied by appropriate narration.
• Control over motion dynamics must be powerful and flexible to produce effective animated communications.
• Timing is the key to effective motion dynamics and algorithm animation — fancy rendering and smooth motion are not necessarily required.

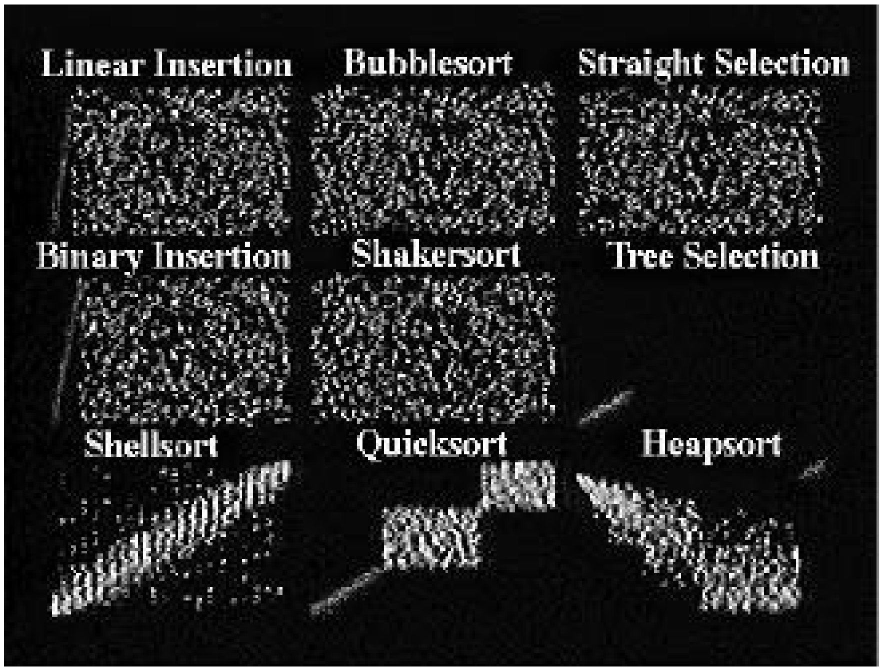
Recent Developments

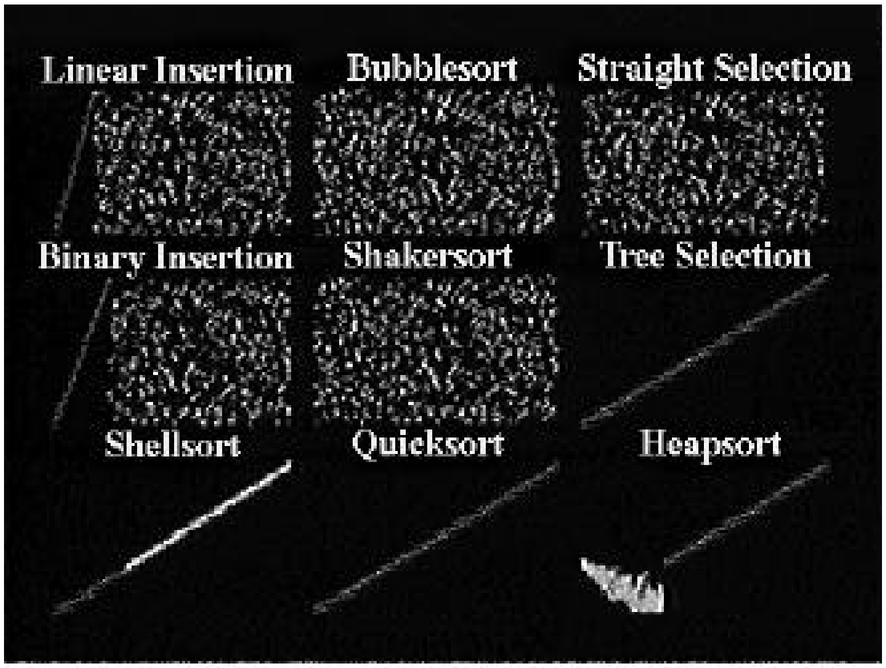
After completing SOS, I took a long hiatus from this research, turning my attention to typographic enhancements of the appearance of source code, and to issues of computer program publishing (see Baecker and Marcus, 1990; also the chapter by Baecker and Marcus).

More recently, we developed two generations of a Logo environment for novice programmers that incorporates tools for software visualization and auralization. The work on LogoMotion is documented in Buchanan (1988) and Baecker and Buchanan (1990). The work on the successor system, LogoMedia, is documented in DiGiano (1992) and DiGiano and Baecker (1992).

In addition to the tight integration of capabilities for program visualization and auralization, two aspects of this work are particularly exciting. The LogoMedia system introduce a novel *probe* metaphor, which students can attach unobtrusively to locations in the program or to data items used by the program. This allows one to specify and tailor visualizations of an algorithm without modifying the program. DiGiano also carried out a ethnographic study of three programmers using LogoMedia on a variety of debugging tasks. The very encouraging results from this study are reported in DiGiano (1992).

Figure 8.a-b The "grand race." Unsorted data appears as a cloud; sorted data becomes a diagonal line. The difference between the n log n sorts (Shellsort, Quicksort, Treesort, and Heapsort) and the n^2 *sorts are clearly visible. Notice Shellsort's pushing of the data towards the line, Quicksort's recursive subdivisions, and Heapsort's strange data funnel.*


Linear Insertion
Bubblesort
Straight Selection
Binary Insertion
Shakersort
Tree Selection
Shellsort
Quicksort
Heapsort


Linear Insertion
Bubblesort
Straight Selection
Binary Insertion
Shakersort
Tree Selection
Shellsort
Quicksort
Heapsort

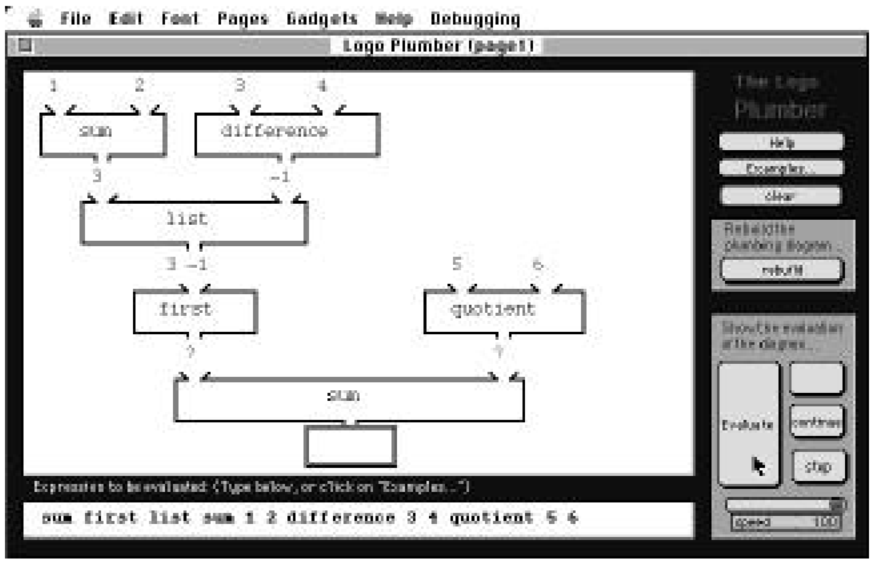
In the last three years, we have also been engaged in developing a novel computer literacy course (Baecker, 1995) and in applying software visualization tools in this course. The course makes use of the Logo Microworlds environment (LCSI, 1993). We have developed a number of experimental animated program execution machines which illustrate and bring to life the underlying syntax and semantics of the execution of Logo programs (Fig. 9). The top image in Fig. 9 shows the Logo Plumber, a Microworlds program that displays the execution of Logo expressions in terms of a plumbing metaphor developed in Harvey (1997). The bottom image in Fig. 9 shows the Logo Visualizer, a Microworlds program that displays the execution of Logo statements and procedures in a subset of the Logo language. Just as *Sorting Out Sorting* brings to life and makes visible how nine sorting algorithms work, these machines bring to life and make visible how the Logo language works.

Conclusions

We have described the development and principles governing the success of the computer animated teaching film *Sorting Out Sorting,* and have also mentioned two recent projects in the educational uses of software visualization. Although this work is very promising, it is distressing to see how difficult it still is to describe and control algorithm animations, how hard it is to get these techniques to scale (but see papers by Eisenstadt and Brayshaw, by Eick, and by Kimelman, Rosenberg, and Roth), and how little the work represented in this volume has been adopted by the mainstream of computer science education and practice.

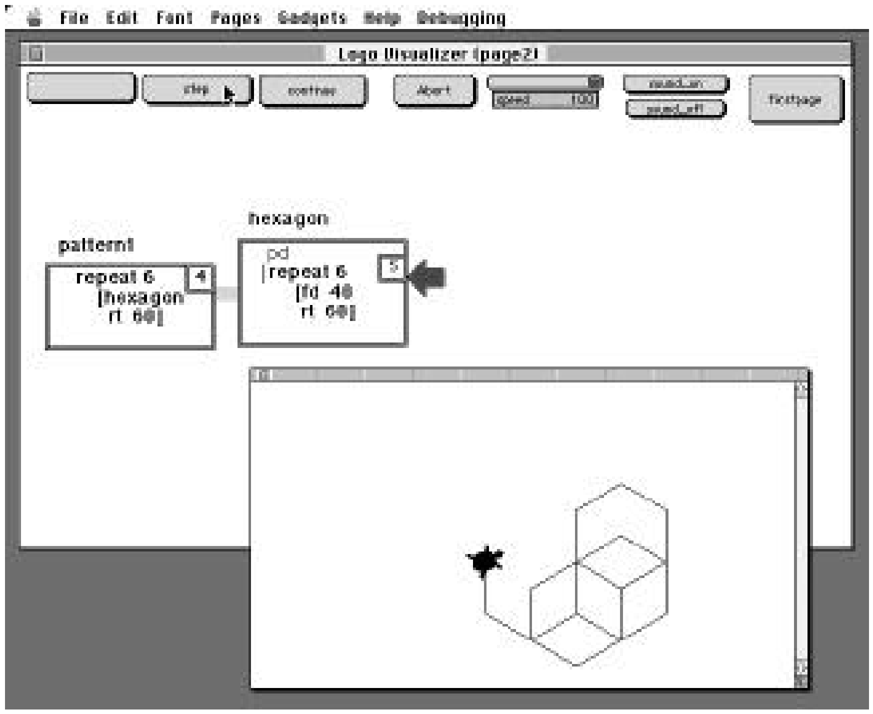
Acknowledgments

Sorting Out Sorting was designed by Ronald Baecker, with assistance from David Sherman, and programmed by David Sherman, with assistance from Anthony Ayiomamatitis, David Gotlib, Thomas O'Dell, and Richard Outerbridge. Martin Tuori at the Defense and Civil Institute of Environmental Medicine graciously contributed the use of a film plotter. Initial versions of the animated program execution machines are due to Diba Bot and Isabel Jevans. The versions shown here are the work of Abba Lustgarten and Alexandra Mazalek. Jason Chang helped with the figures for this paper. The Natural Sciences and Engineering Research Council of Canada provided financial support. *Sorting Out Sorting* may be obtained in Canada from the Information Commons, University of Toronto, 130 St. George St., Toronto Ontario M5S 3H1 Canada, (416)978-6049, fax (416)978-0440, and elsewhere in the world from Morgan Kaufmann Publishers, 340 Pine Street, Sixth Floor, San Francisco CA. 94104 USA, (415)392-2665, mkp@mkp.com.



Expression to be evaluated: (Type below, or click on "Examples...")

```
sum first list sum 1 2 difference 3 4 quotient 5 6
```



```
repeat 6
  [hexagon
   rt 60]
```

```
pd
repeat 6
  [fd 40
   rt 60]
```

Figure 9a-b. Two animated program execution machines.

The top image displays the execution of Logo expressions in terms of a plumbing metaphor. The `sum`, `difference`, *and* `list` *procedures have been evaluated; the* `first` *procedure will be done next. Data flows through an interconnected set of procedures like water flows through a set of pipes.*

The bottom image displays the execution of Logo statements and procedures in a subset of the Logo language. We are in the 4th iteration of calls to a `hexagon` *procedure within the* `pattern1` *procedure. The image drawn by the turtle geometry commands within the procedures is shown in the enclosed window. The execution of successive statements and their effects in the drawing are displayed in synchrony, helping to convey how the Microworlds Logo interpreter executes a program.*

In both cases, we provide single step, pause and continue, and speed controls to aid usability. The graphics is augmented with sound effects to engage the user.

References

Baecker, R.M. (1969). Interactive Computer-Mediated Animation, Ph.D. Thesis, M.I.T. Department of Electrical Engineering, April 1969. Reprinted as M.I.T. *Project MAC TR-61*. Also available as AD 690 887 from the Clearinghouse for Federal Scientific and Technical Information.

Baecker, R.M. (1969). Picture-Driven Animation, *Proceedings 1969 Spring Joint Computer Conference*, 273-288 (Reprinted in H. Freeman (Editor), *Tutorial and Selected Readings in Interactive Computer Graphics*, IEEE Computer Society, 1980, 332-347).

Baecker, R.M. (1973). Towards Animating Computer Programs: A First Progress Report, *Proceedings Third NRC Man-Computer Communications Conference*, May 30-31, 1973, 4.1-4.10.

Baecker, R.M. (1975). Two Systems which Produce Animated Representations of the Execution of Computer Programs, *SIGCSE Bulletin*, Vol. 7, No. 1, February, 1975, 158-167.

Baecker, R.M. (1981). With the assistance of Dave Sherman, *Sorting out Sorting*, 30 minute colour sound film, Dynamic Graphics Project, University of Toronto, 1981. (Excerpted and “reprinted” in *SIGGRAPH Video Review 7*, 1983.) (Distributed by Morgan Kaufmann, Publishers.)

Baecker, R.M. (1995). A New Approach to a University “Computer Literacy” Course. *Proceedings 12th International Conference on Technology and Education*, March 1995, 247-249.

Baecker, R.M. and Buchanan, J. (1990). A Programmer's Interface: A Visually Enhanced and Animated Programming Environment, *Proceedings 23rd Hawaii International Conference on System Sciences*, Jan. 2-5, 1990, 531-540.

Baecker, R.M. and Marcus, A. (1990). *Human Factors and Typography for More Readable Programs*. ACM Press, Addison-Wesley.

Baecker, R.M., Smith, L., and Martin, E. (1970). *GENESYS -- An Interactive Computer-Mediated Animation System*, 17 minute colour sound film, M.I.T. Lincoln Laboratory.

Booth, K.S. (1975). *PQ Trees*, 12-minute colour silent film.

Brown, M.H. (1988). *Algorithm Animation.* MIT Press.

Buchanan, J.W. (1988). LOGOmotion: A Visually Enhanced Programming Environment, M.Sc. Thesis, Dept. of Computer Science, University of Toronto.

DiGiano, C.J. (1992). Visualizing Program Behaviour Using Non-speech Audio, M.Sc. Thesis, Dept. of Computer Science, University of Toronto.

DiGiano, C.J., and Baecker, R.M. (1992). Program Auralization: Sound Enhancements to the Programming Environment, *Proceedings Graphics Interface '92*, Vancouver, B.C., 11-15 May 1992, 44-52.

Harvey, B. (1997, in press). *Computer Science Logo Style. Volume 1: Symbolic Computing*. Second Edition. MIT Press.

Hopgood, F.R. (1974). Computer Animation Used as a Tool in Teaching Computer Science. *Proceedings IFIP Congress*, 889-892.

Knowlton, K. (1966a). *L6: Bell Telephone Laboratories Low-Level Linked List Language.* 16-minute black-and-white film, Murray Hill, N.J.

Knowlton, K. (1966b). *L6: Part II. An Example of L6 Programming*. 30-minute black-and-white film, Murray Hill, N.J.

Knuth, D.E. (1973). *The Art of Computer Programming, Volume 3: Searching and Sorting*. Addison-Wesley.

LCSI (1993). MicroWorlds Reference. Logo Computer Systems Inc., Montreal PQ. Canada.

Lorin, H. (1975). *Sorting and Sort Systems.* Addison-Wesley.

Wirth, N. (1976). *Algorithms + Data Structures = Programs*. Prentice-Hall.

Yarwood, Edward (1974). Toward Program Illustration, M.Sc. Thesis, Dept. of Computer Science, University of Toronto.