76a

Color and Sound in Algorithm Animation

Marc H. Brown and John Hershberger

August 30, 1991

Abstract

Although systems for animating algorithms are becoming more powerful and easier for programmers to use, not enough attention has been given to the techniques that an algorithm animator needs to create effective visualizations. This paper reviews the techniques for algorithm animation reported in the literature thus far and introduces new techniques that we have developed for using color and, to a lesser extent, sound. The paper also presents six algorithm animations that illustrate the new techniques. A videotape of these animations is available.

Review by Lyle Ramshaw

A computer animation of an algorithm in action can clarify how and why that algorithm works. But designing enlightening animations is a tricky psychological and perceptual challenge. What information should be presented? How should it be arranged, in space and in time? What will help the viewer or listener to notice patterns? And how can different perspectives be tied together? This paper presents six example animations of clever algorithms and catalogs the techniques – some old and some new – used in those animations.

1 Introduction

Algorithm animation is a powerful approach for exploring a program's behavior. It has been used with success in teaching computer science courses [6], designing and analyzing algorithms [3], producing technical drawings [26], tuning performance [16], and documenting programs [23].

Although algorithm animation systems are becoming more powerful and easier for programmers to use, the task of using these tools to create effective dynamic visualizations of algorithms still remains a black art. Little attention in the literature has been devoted to the techniques that an algorithm animator must use to design dynamic graphics.

This paper adds to the collection of techniques developed by Brown and Sedgewick [10] using the BALSA algorithm animation system in the "Electronic Classroom" at Brown University in the mid-1980s. Our new techniques focus on the use of color and sound, previously unexplored areas in algorithm animation.

The techniques we describe evolved from our experience with the Zeus algorithm animation system over the past three years. Zeus provides a programmer with support so that it is almost as easy to animate a program as it would be to produce a textual trace of it. Zeus provides a user of the resulting program with ways to specify the input data, to select and manage the ensemble of active views, and to control the program's execution. Technical details about Zeus are given elsewhere [8]. By and large, the techniques we describe are independent of the particular algorithm animation system that we used, and are applicable to other algorithm animation systems.

The appendix contains screen dumps from six different animations that are representative of the ways we have used Zeus. Three static pictures of each of the examples cannot do justice to the interactive animations, and there is no sound, but we hope the the figure captions will provide enough information about the algorithms for you to imagine their dynamics. A color videotape, with sound, showing all six animations in action is available [11]. The figures in the appendix are not presented in any particular sequence; as we discuss a topic, we shall illustrate our points by referring to the relevant figure(s).

The next section briefly reviews the techniques reported by Brown and Sedgewick. Section 3 discusses the problem of choosing input data that best exhibits an algorithm's properties. This technique was alluded to by Brown and Sedgewick but never explored in depth. Finally, sections 4 and 5 describe the new techniques we have developed for using color and sound. Because the remainder of this paper refers extensively to the figures in the appendix, we recommend that you scan the appendix, or view the videotape, before proceeding with the rest of the paper.

2 Previous Work

In this section we review the techniques for algorithm animation that Brown and Sedgewick developed during their work with BALSA [10]. These techniques are also important in our Zeus animations, and the figures illustrate how we have incorporated them.

Multiple views. It is generally more effective to illustrate an algorithm with several different views than with a single monolithic view. A monolithic view concentrates all the information about an algorithm into one dynamic image. This is successful for showing simple algorithms, such as Quicksort in Fig. 3 (top)).

However, to depict a complicated algorithm in detail, or multiple aspects of even a simple algorithm, a single monolithic view must encode so much information that it quickly becomes difficult for the user to pick out the details of interest from the wealth of information on the screen. Our animations generally use multiple views, each displaying a small number of aspects of the algorithm. Each view is easy to comprehend in isolation, and the composition of several views is more informative than the sum of their individual contributions. The hashing animation in Fig. 1 (middle) and the polygon decomposition animation in Fig. 4 (top) exemplify this approach.

Another benefit of using several simple, easy-to-implement views is that it encourages the animator to experiment with different views and keep those that display the algorithm best.

State cues. Changes in the state of an algorithm's data structures are reflected on the screen by changes in their graphical representations. For example, in the quicksort partition trees of Fig. 3 (middle), a node is round while its associated set is being sorted, then changes to square when that set is finished.

State cues link different views together—a single object is represented the same way in every view in which it appears. For example, in the polygon decomposition animation of Fig. 4, polygon vertices change size as the algorithm processes them, and this change is applied consistently throughout the views.

Finally, state cues reflect the dynamic behavior of an algorithm. In the quicksort animation of Fig. 3, unsorted sets of elements are represented by horizontal boxes. When a set is partitioned, its box is replaced by a tree node at the splitting element with two smaller boxes as children. Watching the boxes split and the tree develop dynamically gives an excellent feel for the algorithm.

Static history. Especially when animation is used to explain an unfamiliar algorithm, it is helpful to present a static view of the history of the algorithm and its data structures. Such a view is similar to the way an example might be presented in a textbook; it allows the user to become familiar with the dynamic behavior of the algorithm at his own speed, and to focus on the crucial events where significant changes happen, without paying too much attention to repetitive events.

For example, the hashing animation of Fig. 1 (bottom) records the history of the dynamic hash tables of Fig. 1 (middle) in a left-to-right sweep. Table rehashings, which are the significant events in the algorithm, are clearly visible as sharp discontinuities in the historical record.

The idea of static history is also important in Fig. 4 (top). The Formula view shows the development of a Boolean formula over time, as parentheses and operators are added. The CSG Parse Tree view on the left also embodies a static history: it displays the planar region corresponding to every subformula ever constructed during the algorithm.

Amount of input data. If an animation is used to complement a textual description of an algorithm, it is important to introduce the animation on a small problem instance, preferably with textual annotation, to relate the visual displays to the user's previous understanding. Most of our animations follow this pattern (see Figures 2, 4, and 5). In Fig. 4 (top), we illustrate the algorithm on a seven-vertex polygon, complete with textual labels on all the edges and on the corresponding nodes in the parse tree. Because the example is small and well labeled, the user can easily understand the connections between the views. Once these connections are established, we can introduce larger, more interesting data sets in which the dynamic capabilities of the animation are more fully utilized (as in Fig. 4 (middle)). We omit the labels when displaying these large problem instances, since they would clutter the screen unnecessarily. Section 3 discusses other issues to be considered when choosing input data for an animation.

Continuous versus discrete transitions. When a change to a data structure is represented graphically, the change may be either continuous or discrete. For example, when two sticks exchange places in a sorting animation, it is easier to see the exchange if it appears as a smooth transition instead of an abrupt erase-and-repaint. Continuous change is most helpful for small data sets; for large enough amounts of data, small discrete changes look smooth, and any smoother motion would not be noticeable. (This observation confirms the findings of researchers studying the psychology of human-computer interaction, who suggest that in order to maintain the illusion of animation, the screen must be repainted at least every tenth of a second [14].)

Smooth motion should ideally be provided by underlying graphics software. The animator would specify the endpoints of the transition, a path between them, and the time to be spent moving along it; the graphics package would perform the in-betweening. The TANGO algorithm animation system provides an elegant framework to achieve this effect [28]. Our environment does not provide such support, and so we have used smooth motion in only a few cases, such as moving a sweepline across a polygon (not illustrated).

Multiple algorithms. Running several algorithms simultaneously allows a user to readily compare and contrast the different algorithms. See, for example, the multiple algorithms shown in Figures 2 (bottom), 3 (bottom), 4 (bottom), and 5 (middle).

Algorithm animation systems vary widely in the support they provide for running algorithms simultaneously. The BALSA system is the only system that provides tightly coupled synchronization: each algorithm is run as a coroutine (BALSA is single-threaded), and the coroutine relinquishes flow-of-control each time there is an "interesting event" in the algorithm that will cause a change in the display. The Zeus system provides no special support for synchronizing multiple algorithms. However, because Zeus is implemented on a multi-processor workstation [29] in a software environment that emphasizes concurrency [24, 25], we can easily run several animations simultaneously with their views on the same screen. This technique was used in Figures 2 (bottom), 3 (bottom), and 4 (bottom). The comparison of algorithms in Fig. 5 (middle) is implemented within the algorithm code, without support from the algorithm animation system, in order to achieve accurate relative execution speeds of the various algorithms.

3 Choosing Input Data

The choice of input data strongly influences the message that an animation conveys. In section 2, we discussed the role that the amount of input data can play. In particular, we noted that small amounts of data are good for introducing a new algorithm, whereas large amounts of data are good for developing an intuitive understanding of the algorithm's behavior. This section presents some additional observations on the importance of choosing input data for animations.

Pathological data. It is often instructive to choose pathological data to push the algorithm to extreme behavior.

For example, in the polygon decomposition animation of Fig. 4, we ran the algorithm using both perfectly convex polygons and tight spirals as input. Each input produced a characteristic parse tree (balanced or skewed). When we ran the algorithm on less contrived data, as in Fig. 4 (middle), we could easily pick out the unbalanced subtrees of the parse tree corresponding to the spirals of the input polygon.

In the sweepline animation of Fig. 2, we chose a regular pattern of lines (tangents to a parabola) to understand how different implementations of the algorithm work. The regular arrangement reveals the algorithm's structure better than the more chaotic example of Fig. 2 (middle).

In fact, running the polygon decomposition animation on regular data was instrumental in discovering a subtle bug, as mentioned in the caption for Fig. 4: A perfectly convex polygon as input should have generated a perfectly balanced tree, not merely a well-balanced tree. This bug was not noticed in the more "random" input polygons that we had been using.

Cooked data. Another example of choosing data for pedagogical purposes is shown in Fig. 1. The hashing algorithm of that animation is very effective—so effective that rehashings almost never occur in practice! To make the animation more interesting and instructive, we stacked the deck, so to speak, by filtering out some of the randomness in the input data. The "crippler" filter, whose control interface is shown at the top of the figure, runs in a separate thread and selects input data that hashes into a fixed subset of each hash table, thereby insuring enough collisions to force the tables to be rehashed.

4 Color Techniques

In this section, we outline the ways in which we have used color in Zeus. Color has the potential to communicate lots of information efficiently; however, it is not easy to achieve this goal. Graphics theorists, most notably Bertin [4] and Tufte [30, 31], offer excellent advice on the pragmatics and pitfalls of the use of color (and of displaying data graphically in general). We have tried to follow their principles, but the psychology of color in enhancing communication is beyond the scope of the current discussion.

Algorithm animation introduces additional problems that "classical" graphic designers do not face. Screens are smaller (especially when multiple views partition the screen) and have much lower resolution than paper. Views are dynamic; paper diagrams are not, though they may capture in a static picture data that has changed over time. Multiple views must be united and consistent, yet not interfere with each other; traditional graphic design is typically concerned with a single picture. Views in algorithm animation must be robust enough to handle many different sets of data, not necessarily known in advance. This contrasts with static graphic designs, which may be tuned to a particular data set.

Our use of color as an integral element in program visualization is novel. (Although color plates appear in previous articles on BALSA [7, 9], they are static images, not generated directly by that animation system.) We use color for five distinct purposes: encoding the state of data structures, tying views together, highlighting activity, emphasizing patterns, and making history visible. The figures in the appendix illustrate all these uses.

Color reveals an algorithm's state. For example, in the parallel quicksort algorithm of Fig. 3, the colors of dots and blocks indicate the partition of the elements among the sorting processors. In the robot algorithm of Fig. 6, green, pink, and gray indicate the suitability of different starting points for robot motion; the two key data structures, which are otherwise similar in appearance, are distinguished by pink and light blue coloring. In Fig. 1 (middle), different hash functions are indicated by a colored band around the hash table. The band changes color when a new hash function is chosen.

As an indicator of algorithm state, color enhances and complements the graphical techniques mentioned in Section 2 in at least three ways. First, it gives an extra dimension for state display—one can encode information in both the shape and the color of objects. Second, it allows denser presentation of information: fewer pixels are needed to make a color change visible than to make a change in the shape of an object visible. Third, color is good for displaying global patterns. For example, if a monochromatic group of small triangles changes to a monochromatic mixture of circles and squares, it will be much harder to perceive global patterns than if, say, a group of black dots changes to a mixture of red and blue dots.

Color unites multiple views. When multiple views show different aspects of the same data structure, or different representations of logically related objects, an application can create a smoother, more harmonious picture by painting corresponding features with the same colors in all the views. The polygon decomposition animation (Fig. 4) uses the colors blue, red, and black to denote objects that have been, are being, or have not yet been processed, respectively. This idea is applied uniformly; combined with the visual prominence of the color red, this makes it easy to see the connection between the active edges of the polygon and the corresponding active sites in the formula and parse tree views. The hashing animation of Fig. 1 associates keys with tables by painting them with the same color. In the sweepline animation in Fig. 2, the edges that cross the sweepline are painted red in all the views in which the sweepline is important; similarly, vertices where the sweepline can advance are marked with black disks.

Color highlights areas of interest. Many animations temporarily paint a small region with a transparent, contrasting color to focus attention on the painted area. Because the highlight color is transparent, it does not interact visually with the data elements on the screen, but simply draws the eye to them. For example, in Fig. 4 (top), the active polygon edge C is highlighted in brown. A second use of highlighting is to display transient computations without permanently altering the on-screen state. In Fig. 4 (middle), the brown convex hull is an essential part of the algorithm, but it changes too rapidly to belong to the relatively stable state displayed in the rest of the view. The convex hull is shown in a separate view—it would be distracting if it were always shown—and drawn as a temporary highlight. Similarly, the robot paths of Fig. 6 are drawn only in highlighting, since they don't change the underlying data structures.

The mechanism of highlighting is worth mentioning. Because a highlight is only temporary, we must be able to remove it and quickly restore the parts of the scene the highlight overwrote. In the absence of double-buffering, we accomplish this with color table trickery. The highlight is assigned one bit of the eight-bit color table index; painting or erasing a highlight means setting or clearing that bit. The colors in the table are carefully arranged so this gives the effect of painting with transparent color. Because each view appears in its own window installed in the window manger, the views must cooperate and use the same color table to ensure that the correct colors are seen in all views simultaneously.

The same highlighting technique is used in Fig. 2 to move the sweepline without overwriting the line arrangement below it. If we just used XOR on a standard color table, as is often done for highlighting, we would not get the uniform highlight color we want.

Color emphasizes patterns. In the polygon decomposition animation of Fig. 4 (middle), each deep subtree in the right view grows downward at the same time as the highlighted vertex runs inward along one of the spirals in the left view. The colors of the subtrees and the spirals also change in concert. The kinetic connection between the two views underlines the linkage between spirals and deep parse tree subtrees.

Color captures history. In many examples, we have used a spectrum to show the history of an algorithm. Tufte warns that humans do not perceive the rainbow color sequence as ordered [31]; nevertheless, a sequence of colors ordered by hue can be used to represent a linear time order, especially when there is also some spatial monotonicity to the regions to be painted. Figures 2 and 6 illustrate this point. In both cases the colored regions are roughly ordered by time, and so the viewer does not need a sense of the global color order—it is enough to perceive neighbor differences.

5 Audio Techniques

Although our use of sound in Zeus for algorithm animation is by no means sophisticated, our preliminary experiments have convinced us that sound will be a powerful technique for communicating information. We strongly concur with Gaver that

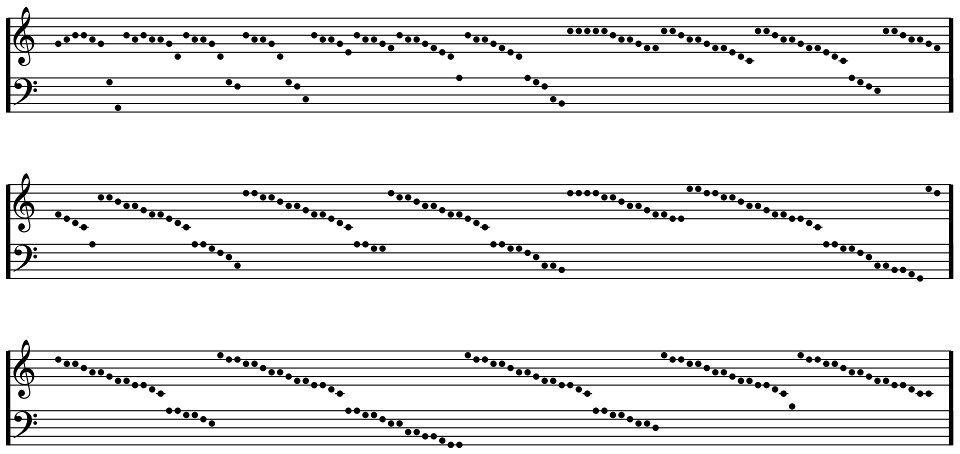
> Auditory displays have the potential to convey information that is difficult or awkward to display graphically. Sound can provide information about events that may not be visually attended, and about events that are obscured or difficult to visualize. Auditory information can be redundant with visual information, so that the strengths of each mode can be exploited. In addition, using sound can help reduce the visual clutter of current graphic interfaces by providing an alternative means for information presentation. [19]

With recent advances in workstation technology that make it easy to generate sound, it is not surprising that other researchers are also beginning to use sound for program comprehension [27] and monitoring performance of parallel programs [1]. In fact, a recently published mathematical textbook is packaged with a compact disc depicting many of the analytical functions discussed and presented graphically in the text [20]. An excellent survey of uses of sound for human-computer interaction is contained in the notes for Buxton, Gaver, and Bly's tutorial on *The Use of Non-Speech Audio at the Interface* [12].

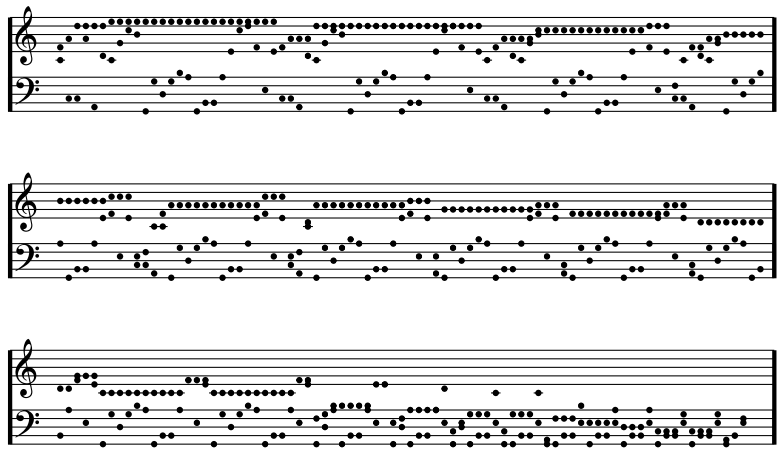
We have found sound to be much more difficult to use than color, primarily because most people do not have the same level of sophistication and training aurally as they do visually. Audio also raises a few logistical problems: What happens when more than one workstation is using audio in the same room? What happens when more than one "view" uses audio?

Nonetheless, we have had positive preliminary experiences using audio in algorithm animations for reinforcing visual views, conveying patterns, replacing visual views, and signaling exceptional conditions. We now elaborate on each of these uses.

Audio reinforces visual views. Our first foray into using audio, and perhaps its most obvious use, was simply to reinforce what was being displayed visually. For example, in the hashing animation of Fig. 1, each table has a pitch associated with it; inserting an element into a table produces a tone of the corresponding pitch. In the sorting animation of Fig. 3, each comparison or movement of an element produces a tone whose pitch is linearly related to the element's value. The musical score below shows the notes generated while sorting a file of 32 elements by insertion sort.

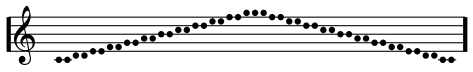


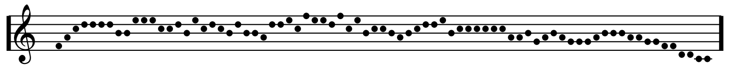
Audio conveys patterns. It became immediately obvious to us in "hearing" the sorting animation that sorting algorithms produce auditory signatures just as distinctive as the visual patterns of moving sticks or dots. It would be generous to use the term "music" to describe the signature, of course. Compare the signature of bubble sort, shown below, with that of insertion sort above.



It is possible for people to hear relationships in data that are never seen or displayed (and vice versa). Because sound intrinsically depends on the passage of time to be perceived, it is not surprising that sound is very effective for displaying dynamic phenomena, such as running algorithms.

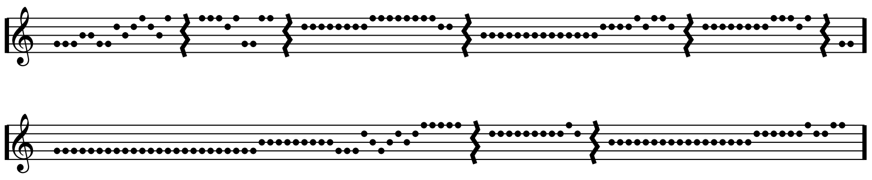
Audio replaces visual views. We've used audio views to replace what can easily be displayed in a visual view in order to allow the user to focus full visual attention on other visual views. For example, in the parallel quicksort algorithm in Fig. 3 and in the parallel topological sweep in Fig. 2, the sound effects "view" produces a tone whose pitch rises with the number of active threads (virtual processors). This number could easily be printed textually, or graphically displayed as a bar chart. However, because the user receives the thread information through a non-visual channel, he can focus full visual attention on the algorithm at work. The scores below compare the thread information for two sets of topological sweep input data. The top score corresponds to the regular arrangement of lines in Fig. 2 (bottom), and the lower score to the less orderly arrangement in Fig. 2 (middle).





Audio signals exceptional conditions. It came as no surprise to us to find that audio was very effective for signaling exceptional conditions. (After all, computers have long beeped at users.) However, an algorithm animation is different from an interactive program that, say, beeps when the user tries to do something illegal. This is because there are long periods of using algorithm animations when the user is passively watching the algorithm in action. In this situation, the visual input channel can easily be "turned off" by looking away, looking at the wrong part of a display, or being lulled into complacency by the normal case. It is harder (though certainly not impossible) to turn off people's audio input channel.

For example, in the hashing animation shown in Fig. 1, inserting an element into a table makes a tone whose pitch depends on the table. Thus, the executing algorithm sounds notes within some chord. However, when a new element collides with old elements in all tables, the sound of a violent car crash is heard, underlining the idea of a collision. The score below shows the same hash table history recorded visually in Fig. 1 (bottom). The vertical wavy bars denote the crash sounds.



(It is easy to discount this use of real-world sounds as cutesy. We refer the skeptic to Gaver for a thorough discussion of how and why everyday sounds can and should be integrated into computer programs [19].)

6 Conclusion

Using an algorithm animation system to create effective dynamic visualizations of computer programs is a craft, not a science. In practice, successful algorithm animations make use of a small "bag of tricks." This paper reviewed the techniques previously reported in the literature, and offered a number of new techniques, primarily for using color and sound, that we have utilized during the past three years. The figures in the appendix give examples of the techniques discussed.

We found that we use color for encoding the state of data structures, tying multiple views together, highlighting activity, emphasizing patterns, and making an algorithm's history visible in a single static image. We use sound for reinforcing visual views, conveying patterns, replacing visual views, and signaling exceptional conditions. Although our experimentation with sound is still very preliminary and relatively unsophisticated, it has proven to be very useful.

As workstation hardware and software become more powerful, more sophisticated graphics and audio techniques will become possible. For example, current workstation technology can support real-time 3-D views, ranging in sophistication from black-and-white stick figures to realistic, ray-traced images. Our intuition (supported by Lieberman [22] and recently corroborated at Xerox PARC [13]) is that 3-D views can give an "extra dimension of information" in the same way that color and audio convey more information than silent black-and-white.

Color and sound (and probably 3-D graphics) do not merely enhance the beauty of a presentation; they can be used to give fundamental information. Now, it is hard for us to imagine trying to convey sufficient information in algorithm animations without color or sound, as we used to do. We hope the new techniques presented in this paper for using color and sound will ease the task of future algorithm animators.

Acknowledgements

Lyle Ramshaw and Cynthia Hibbard improved the clarity of this paper. Harald Rosenberger implemented the topological sweep line algorithm (Fig. 2); Anna Karlin and Mark Manasse helped implement the spin block algorithms (Fig. 5); and Andrei Broder implemented the multilevel adaptive hashing algorithm (Fig. 1). Their assistance in designing the views for the animation of those algorithms is greatly appreciated.

References

[1] Larry Albright, Joan M. Francioni, and Jay Alan Jackson. Auralization of parallel programs. In *Proc. ACM SIGCHI '91 Conf. on Human Factors in Computing Systems*, page 497, April 1991.

[2] Gideon Avrahami, Kenneth P. Brooks, and Marc H. Brown. A two-view approach to constructing user interfaces. *Computer Graphics*, 23(3):137–146, July 1989.

[3] Jon L. Bentley and Brian W. Kernighan. A system for algorithm animation. *Computing Systems*, 4(1):5–30, 1991.

[4] Jacques Bertin. *Semiology of Graphics*. University of Wisconsin Press, 1983.

[5] Andrei Z. Broder and Anna R. Karlin. Multilevel adaptive hashing. In *Proc. of the 1st Annual ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 43–53, January 1990.

[6] Marc H. Brown. *Algorithm Animation*. MIT Press, Cambridge, MA, 1988.

[7] Marc H. Brown. Exploring algorithms using BALSA–II. *IEEE Computer*, 21(5):14–36, May 1988.

[8] Marc H. Brown. Zeus: A system for algorithm animation. In *Proc. IEEE 1991 Workshop on Visual Languages*, October 1991.

[9] Marc H. Brown and Robert Sedgewick. A system for algorithm animation. *Computer Graphics*, 18(3):177–186, 1984.

[10] Marc H. Brown and Robert Sedgewick. Techniques for algorithm animation. *IEEE Software*, 2(1):28–39, January 1985.

[11] Marc H. Brown, ed. An anthology of algorithm animations using Zeus. Research Report Videotape 76b, DEC Systems Research Center, 130 Lytton Ave., Palo Alto, CA, September 1991. A segment entitled "An Introduction to Zeus: Audio Visualization of Some Elementary Sequential and Parallel Sorting Algorithms" is part of the CHI '92 video program.

[12] William Buxton, William Gaver, and Sara Bly. The use of non-speech audio at the interface. CHI '90 Tutorial Notes, Seattle, WA, April 1990.

[13] Stuart K. Card, Jock D. Mackinlay, and George G. Robertson. Cone trees: Animated 3D visualizations of hierarchical information. In *Proc. ACM SIGCHI '91 Conf. on Human Factors in Computing Systems*, pages 189–194, April 1991.

[14] Stuart K. Card, Thomas P. Moran, and Allen Newell. *The Psychology of Human-Computer Interaction*. Erlbaum, 1983.

[15] David Dobkin, Leonidas Guibas, John Hershberger, and Jack Snoeyink. An efficient algorithm for finding the CSG representation of a simple polygon. *Computer Graphics*, 22(4):31–40, 1988.

[16] Robert A. Duisberg. Animated graphical interfaces using temporal constraints. In *Proc. ACM SIGCHI '86 Conf. on Human Factors in Computing Systems*, pages 131–136, April 1986.

[17] Herbert Edelsbrunner and Leonidas J. Guibas. Topologically sweeping an arrangement. In *Proc. of the 18th ACM Symposium on Theory of Computing*, pages 389–403, May 1986.

[18] J. Friedman, J. Hershberger, and J. Snoeyink. Compliant motion in a simple polygon. In *Proc. of the 5th ACM Symposium on Computational Geometry*, pages 175–186, 1989.

[19] William W. Gaver. The SonicFinder: An interface that uses auditory icons. *Human-Computer Interaction*, 4(1), 1989.

[20] Theodore W. Gray and Jerry Glynn. *Exploring Mathematics with Mathematica: Dialogs Concerning Computers and Mathematics*. Addison-Wesley, Reading, MA, 1991.

[21] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan Owicki. Empirical studies of competitive spinning for a shared memory multiprocessor. In *Proc. of the 13th Annual ACM Symposium on Operating Systems Principles*, 1991.

[22] Henry Lieberman. A three-dimensional representation for program execution. In *Proc. 1989 Workshop on Visual Languages*, pages 111–116, October 1989.

[23] Ralph L. London and Robert A. Duisberg. Animating programs using Smalltalk. *IEEE Computer*, 18(8):61–71, August 1985.

[24] Paul R. McJones and Garret F. Swart. Evolving the UNIX system interface to support multithreaded programs. In *Proc. Winter 1989 USENIX Technical Conference*, pages 393–404, 1989.

[25] Paul Rovner. Extending Modula–2 to build large, integrated systems. *IEEE Software*, 3(6):46–57, November 1986.

[26] Robert Sedgewick. *Algorithms*. Addison-Wesley, Reading, MA, 2nd edition, 1988.

[27] D. H. Sonnenwald, B. Gopinath, G. O. Haberman, W. M. Keese III, and J. S. Myers. InfoSound: An audio aid to program comprehension. In *Proc. of the 23rd Hawaii Int'l. Conf. on System Sciences*, pages 541–546, January 1990.

[28] John T. Stasko. A practical animation language for software development. In *Proc. IEEE Intl. Conf. on Comp. Lang.*, pages 1–10, 1990.

[29] Charles P. Thacker, Lawrence C. Stewart, and Edwin H. Satterthwaite Jr. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[30] Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, CT, 1983.

[31] Edward R. Tufte. *Envisioning Information*. Graphics Press, Cheshire, CT, 1990.

APPENDIX

Animation Examples

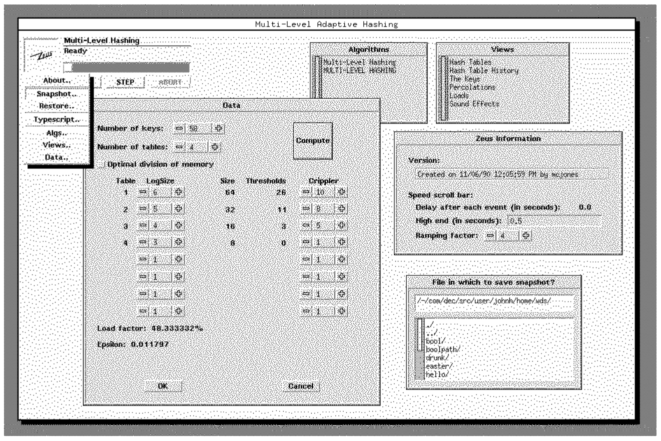
The following pages display screen images from six different Zeus animations. The animated algorithms span a wide variety of algorithmic disciplines, ranging from basic sorting algorithms to algorithms for hashing, competitive spinning, and computational geometry. The figure captions describe the algorithms and their animations in detail, illustrating the techniques described in the body of this report.

Figure 1. Multi-level adaptive hashing [5]. The screen image at the top shows the Zeus control panel. The control panel provides menus to select algorithms, views, and input data. The data menu may be customized to the application, as in this case; otherwise, the default menu is a browser to select an input file. Zeus provides a snapshot/restore capability for preserving settings between program runs. Finally, the control panel contains buttons for starting/stopping/stepping the algorithm, as well as a slider controlling its execution speed.

The middle screen image illustrates a hashing scheme that uses multiple hash tables to store an online dictionary. The tables are linked in a chain from left to right, so a collision in one table causes the colliding element to percolate into the next table. Whenever too many elements have percolated out of a table, that table and all its successors will be rehashed with new hash functions the next time that a new element collides in all the tables. This scheme provides constant-time lookup, provided that the hardware accesses the tables in parallel.

Elements in the "Keys" view are color-coded to indicate the tables to which they belong in the "Hash Tables" view. The color of the band surrounding each hash table corresponds to the hash function currently in use for that table. When an element is inserted, highlighting surrounds the location in each table to which it hashes. In the example shown here, the new element will be inserted in the third table, since it collides with elements in the first two tables. The two statistics views on the right show the number of percolations out of the table and the table load for each hash table; the third table has reached its percolation limit—the third and fourth tables will be rehashed after the next all-table collision. The "Sound Effects" view causes a table-specific tone to be sounded each time that an element is inserted into a hash table.

The bottom screen image shows the history of the hash tables over time. The four tables are shown stacked atop each other, and time runs from left to right. Keys are shown in the (permanent) color of their table; the background color of a table changes when it is rehashed. For example, half-way through this execution of the algorithm, all of the tables were rehashed; the tables were emptied and the elements reinserted. (The rehashings are not simultaneous because elements are reinserted into one table at a time.)



Data

Number of keys: 50

Number of tables: 4

Optimal division of memory

Table	LogSize	Size	Thresholds	Crippler
1	6	64	26	10
2	5	32	11	8
3	4	16	3	5
4	3	8	0	1
	1			1
	1			1
	1			1
	1			1

Load factor: 48.333332%

Epsilon: 0.011797

Zeus Information

Version:

Created on 11/06/90 12:05:59 PM by mcjones

Speed scroll bar:

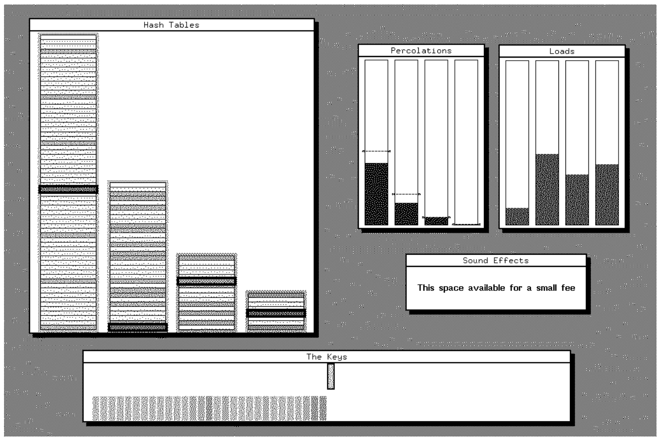
Delay after each event (in seconds): 0.0

High end (in seconds): 0.5

Ramping factor: 4

File in which to save snapshot?

```
/-/com/dec/src/user/johnh/home/wds/
./
../
bool/
boolpath/
drunk/
easter/
hello/
```



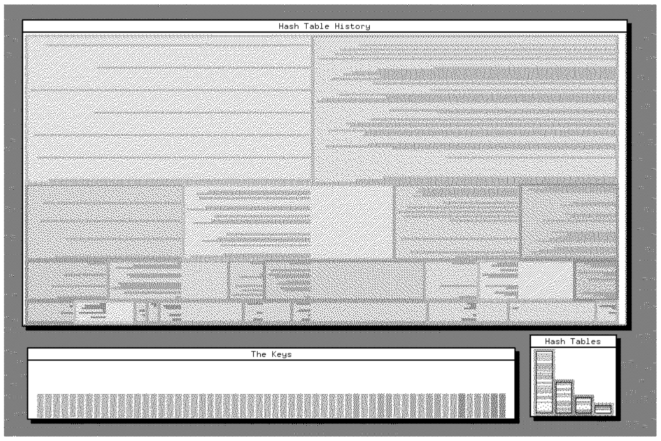
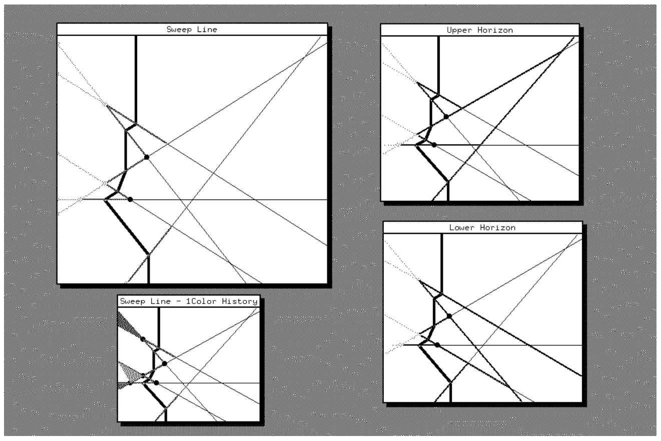
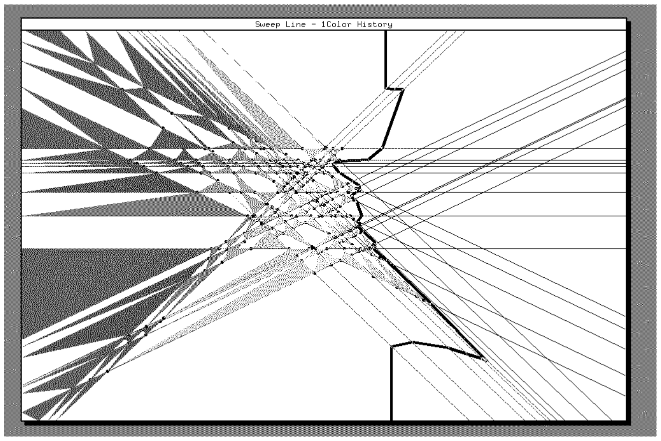


Figure 2. Topological sweepline [17]. An ordinary sweepline is a vertical line that visits the $O(n^2)$ intersections in an arrangement of n lines in the plane by sweeping across the arrangement from left to right. Such a sweepline uses only $O(n)$ working storage, but, because it sorts the intersections in x-order, it spends $O(n^2 \log n)$ time. In many cases the sorting is unnecessary; it is enough just to visit all the intersections in any order. A topological sweepline visits the intersections in optimal $O(n^2)$ time by sacrificing the straightness of the ordinary sweepline, while retaining the $O(n)$ space bound.

In between visiting intersections, the topological sweepline crosses n edges of the arrangement—an upper and a lower edge from each convex face it crosses. The active edges—those crossed by the sweepline—are shown in red in the "Sweep Line" view in the top screen image. The light blue edges have already been swept, and the thin black edges remain to be swept. The black dots show intersections that the sweepline could visit next (it could move from the two edges left of the intersection to the two edges right of it). The sweepline can choose arbitrarily which black dot to advance over next, or can even advance over all of them in parallel. The lower and upper "Horizon" views display the data structures that the algorithm uses to identify intersections that it can visit next. In the middle image, a color spectrum is used to display the history of the sweepline's movement. Each region of the arrangement is flagged with a triangle whose color tells when the sweepline passed the rightmost intersection on its boundary.

The bottom image juxtaposes two different algorithms—the left one sequential and the right one parallel. The sequential algorithm always advances the sweepline at the uppermost possible intersection, while the parallel algorithm advances over all possible intersections concurrently. This concurrent advance makes the sweepline look ragged.


Sweep Line
Upper Horizon
Lower Horizon
Sweep Line - 1Color History


Sweep Line - 1Color History

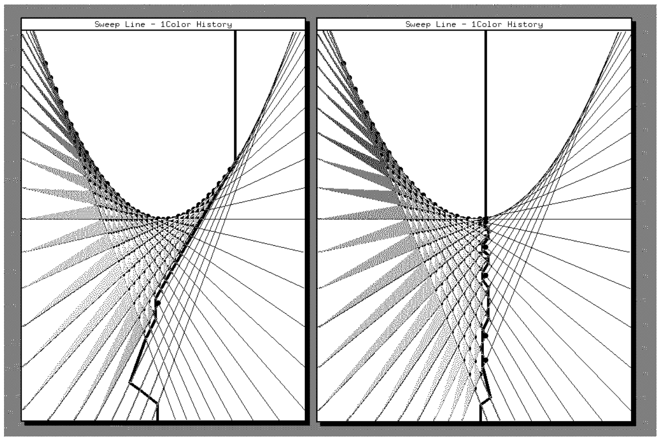
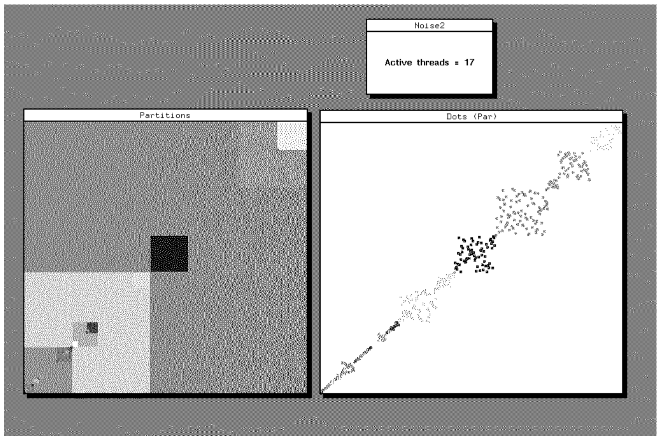
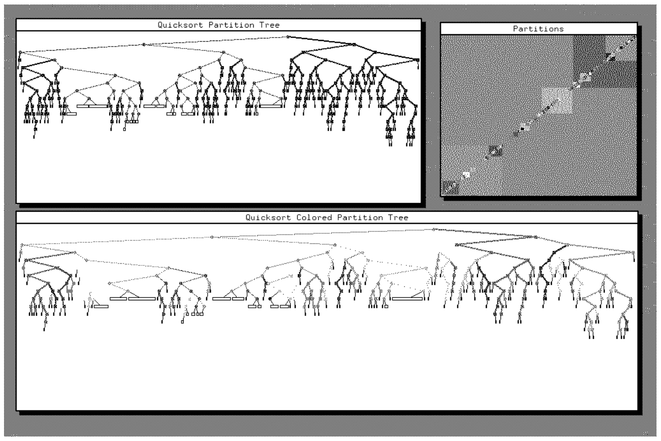

Sweep Line - 1Color History
Sweep Line - 1Color History

Figure 3. Quicksort. The top screen image shows a naïve parallel implementation of Quicksort. When the elements to be sorted have been partitioned into two groups, the partitioning thread recursively sorts one group, while a new thread is forked to sort the other group. Each thread has its own color, which is used to indicate the region and the elements on which it operates. The small window at the top of the screen, which displays the number of active threads, is the visual component of an "audio view"—the same information is represented by the pitch of a tone played through the workstation's speaker.

The middle image presents a binary-tree view of the partitioning process. Each node in the tree represents a position in the array being sorted. Each subtree represents a block of elements to be sorted recursively; the block is partitioned into two groups separated by the root of the subtree. In the lower tree, the colors of nodes and edges reflect the threads assigned to partition each block. In the upper tree, red and blue colors are used to distinguish active blocks from inactive ones. While a block is being sorted, its node and the edge to its parent are red; they change to blue when the block is finished. Horizontal boxes represent unexamined blocks.

The bottom image compares three different quicksort implementations using the red/blue tree view. The top tree shows a sequential version—exactly one root-to-leaf path is active at any time. The naïve parallel implementation is shown on the right—many threads are active at once. The left view shows a more sophisticated parallel algorithm. It sorts small blocks by selection sort (indicated by the unexpanded boxes), and forks a new thread only when the block to be sorted is large enough—in this view only two threads are active.


Noise2
Active threads = 17
Partitions
Dots (Par)


Quicksort Partition Tree
Partitions
Quicksort Colored Partition Tree

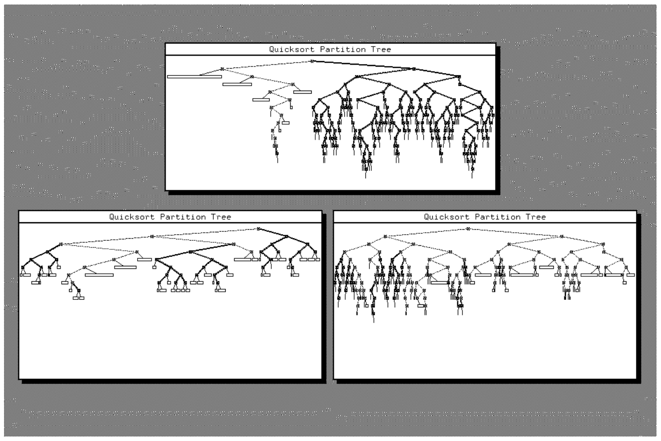
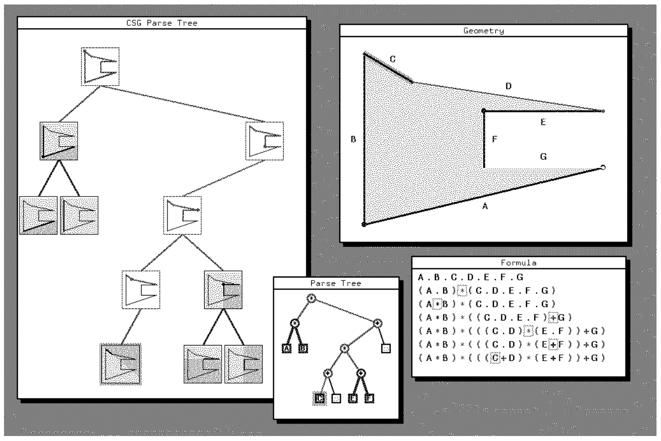

Quicksort Partition Tree
Quicksort Partition Tree
Quicksort Partition Tree

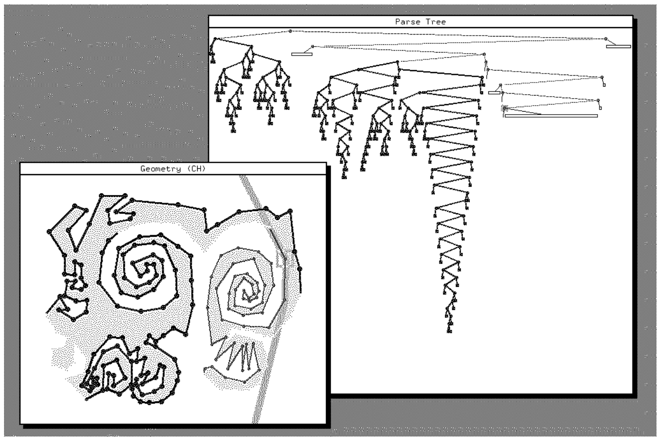
Figure 4. Boolean Formulæ for Simple Polygons [15]. Given a simple polygon, this algorithm represents its interior as a monotone Boolean combination of the halfplanes determined by its edges. (A *simple polygon* is a closed polygonal path, free of self-intersections; a *monotone Boolean combination* is a Boolean formula containing only unions ("+") and intersections ("*")—no negations are allowed.) The views display the polygon itself, the Boolean formula and its development, and the parse tree corresponding to the formula. Color is used consistently in all views: red represents the subpath of the polygon being processed, blue the parts already processed, and black the parts yet to be processed. Highlighting focuses attention on the edge or vertex where the formula is being changed. In the "CSG Parse Tree," each node is represented by the planar region that results from evaluating its subformula. The "Parse Tree" view is a compact version of the same tree that omits the CSG regions. It is especially appropriate for large examples like the one in the middle.

The middle example emphasizes a strength of algorithm animation—revealing hitherto unnoticed features of the algorithm. The deep, zigzag subtrees in the parse tree correspond to the spirals of the polygon, a fact underlined by the dynamic visualization. (The zigzag subtree corresponding to the red spiral is about to be constructed.)

The bottom image illustrates the helpfulness of graphical output in debugging. We expected the parse tree for the circle polygon to be balanced, like the one in the bottommost view. When the algorithm instead produced the parse tree in the upper view, which has leaves at four different depths, we investigated and soon turned up a subtle error in the implementation of the algorithm.



```
A.B.C.D.E.F.G
(A.B)*(C.D.E.F.G)
(A*B)*(C.D.E.F.G)
(A*B)*((C.D.E.F)+G)
(A*B)*(((C.D)*(E.F))+G)
(A*B)*(((C.D)*(E+F))+G)
(A*B)*(((C+D)*(E+F))+G)
```



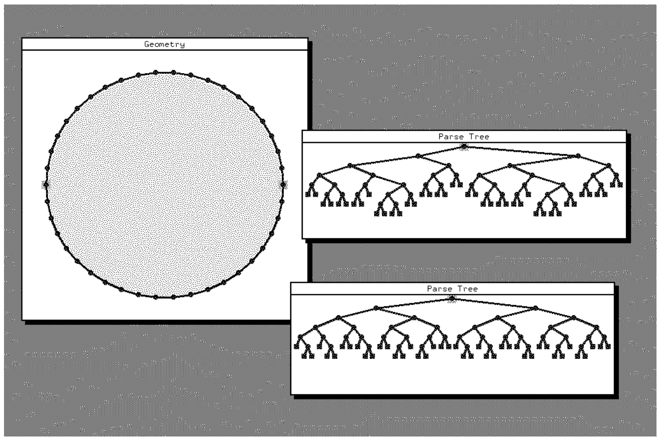
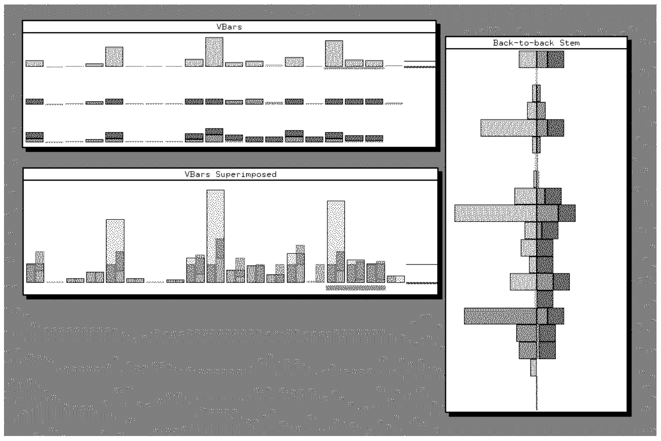
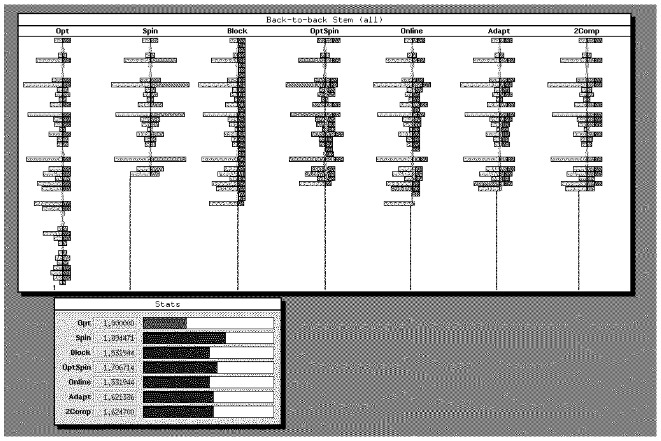


Figure 5. To Spin or to Block? [21]. When one thread of a multi-threaded program needs a shared resource (like a file server or a semaphore), it must wait until the resource is available before it can proceed. While waiting, the thread may *spin* (busy wait) or *block* (suspend itself). The former wastes processor time that might be used by other threads, while the latter incurs a fixed context-switch overhead. The best choice (spin or block) depends on the waiting time, which is not known in advance. The ratio of processor time spent to that actually needed is potentially unbounded. But a program that spins for a while and then blocks can come within a small factor of the optimal (clairvoyant) offline algorithm.

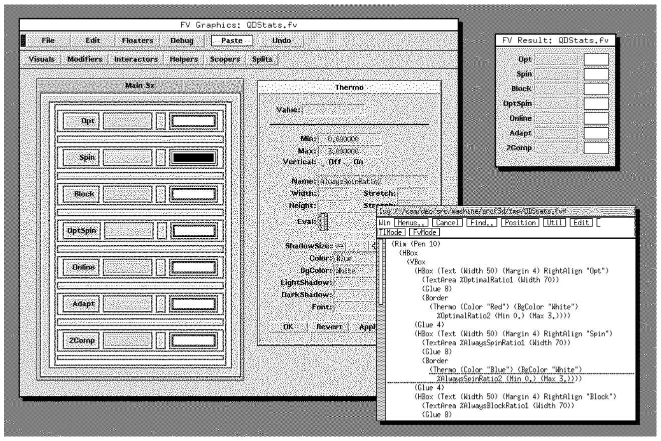
The top screen image compares an online algorithm that considers only the previous three waiting times against the optimal algorithm. The three windows present different views of the same data. The actual waiting times are shown in gray-blue, the remembered times in brown, algorithm spin time in green, and blocking time in red. In the "VBars" window, the top row shows the waiting times, the middle row represents the optimal algorithm, and the bottom row the algorithm being demonstrated. The "VBars Superimposed" view superimposes these three rows for easier comparison.

The middle image compares six different online algorithms with the optimal offline algorithm. The gray-blue bars on the left show the waiting time; the green and red bars show the spinning/blocking time used by each algorithm. The vertical red lines show the current thresholds at which each algorithm stops spinning and decides to block. The performance of each method is made visible graphically by the progress downward in each column. The view at the bottom of the screen shows the performance ratios numerically and graphically. This view was was constructed graphically using the FormsVBT multi-view editor for user interfaces [2], as shown in the bottom image. Here a thermometer gauge has been selected in the graphical editor, which pops up a property form and highlights the textual description of the gauge in the text editor.





Opt	1.000000
Spin	1.894471
Block	1.531944
OptSpin	1.706714
Online	1.531944
Adapt	1.621336
2Comp	1.624700



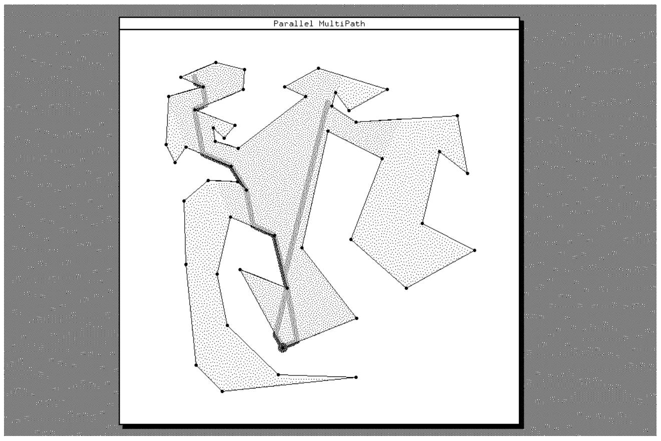
Value:
Min: 0.000000
Max: 3.000000
Vertical: Off On
Name: AlwaysSpinRatio2
Width:
Stretch:
Height:
Stretch:
Eval:
ShadowSize:
Color: Blue
BgColor: White
LightShadow:
DarkShadow:
Font:

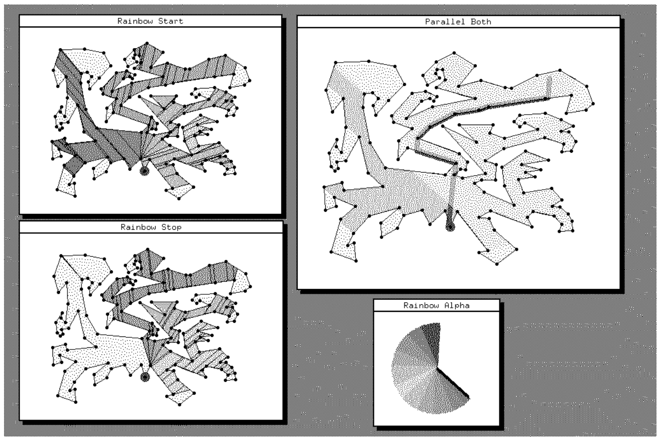
```
(Rim (Pen 10)
  (HBox
    (VBox
      (HBox (Text (Width 50) (Margin 4) RightAlign "Opt")
        (TextArea %OptimalRatio1 (Width 70))
        (Glue 8)
        (Border
          (Thermo (Color "Red") (BgColor "White")
            %OptimalRatio2 (Min 0.) (Max 3.))))
      (Glue 4)
      (HBox (Text (Width 50) (Margin 4) RightAlign "Spin")
        (TextArea %AlwaysSpinRatio1 (Width 70))
        (Glue 8)
        (Border
          (Thermo (Color "Blue") (BgColor "White")
            %AlwaysSpinRatio2 (Min 0.) (Max 3.))))
      (Glue 4)
      (HBox (Text (Width 50) (Margin 4) RightAlign "Block")
        (TextArea %AlwaysBlockRatio1 (Width 70))
        (Glue 8)
```

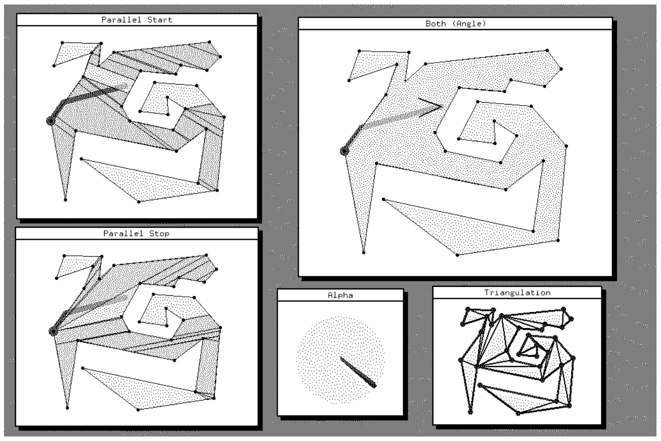
Figure 6. Compliant motion planning [18]. This algorithm plans the motion of a robot that, once started, always heads in the same direction. When the robot hits an obstacle—such as a wall—obliquely, it slides along that obstacle, continuing to head in the same direction as it slides. The resulting path of the robot may or may not reach the goal point, shown as a red dot. Paths from two different starting points that do reach the goal point are shown in the top image. The algorithm computes the set (shown in green) of all points from which the robot can reach the goal in a single programmed movement.

For any point, the set of directions in which the robot can reach the goal forms a single angular range. The algorithm finds these ranges by rotating a direction α through 360°, while maintaining the set of points R_α from which the robot can reach the goal in direction α. The directions at which a point enters and leaves R_α bound its angular range. When a point enters R_α, it is added to a *start subdivision*; when it leaves, it is added to a *stop subdivision*, as shown in the middle image. Color encodes the directions at which points are added to the two subdivisions, and hence records the history of the algorithm. The pink region in the top right view is the current set of points—points already in the start subdivision, but not yet in the stop subdivision.

After the two subdivisions are completed, the user can specify an initial robot position with the mouse (bottom image). In response to the query, the algorithm locates the point in the two subdivisions (the results are highlighted in green), combines the results to get the range of feasible directions (the black angle in the upper-right view), and computes the robot's path (highlighted in purple).


Parallel MultiPath


Rainbow Start
Parallel Both
Rainbow Stop
Rainbow Alpha


Parallel Start
Both (Angle)
Parallel Stop
Alpha
Triangulation