Techniques for Algorithm Animation

Marc H. Brown and Robert Sedgewick
Brown University

Allowing a user to interact with dynamically changing graphical representations of algorithms or data structures may help in teaching, research, or systems programming.

High-performance, graphics-based personal workstations, interconnected by a high-bandwidth, resource-sharing local area network, provide a rich communications environment. While such architectures are fast becoming the computing medium of choice in academic and research environments, all too often this modern equipment is used to support traditional modes of computing, and its capabilities are not fully realized. For some time, we have been exploring ways to overcome this tendency, and, instead, to exploit the architecture to increase the effectiveness of our research and teaching. We have over 100 such workstations in the Department of Computer Science at Brown University, 55 of which are housed in the Department's instructional workstation laboratory, a specially built auditorium/lecture hall. In this article, we describe some of our work in an application where we have concentrated most of our efforts: algorithm animation.

The Brown University Algorithm Simulator and Animator, or Balsa, is an integrated software environment designed to "animate" programs. On the one hand, Balsa is a system that supports a high-level user interface that allows a user to interact with dynamically changing graphical representations of his programs; on the other hand, it is a set of tools allowing one kind of Balsa user to build an environment for another kind of Balsa user. Technical details on Balsa may be found elsewhere,[1] and we briefly summarize some of this information next.

A similar version of this article was presented this year at the ACM 18th International Conference on System Sciences, January 2-4, Honolulu, Hawaii.

The Balsa environment provides three general types of facilities: *display* (a window manager), *interpretive* (an interpreter), and *shell* (a high-level command processor). Though these subsystems are relatively complete, they are not intended for general-purpose use; rather, they are tailored to be used for algorithm animation, while also emphasizing a quality user interface and performance. Runtime support routines display the animation, while a library of utility programs facilitate the construction of the animation.

This special-purpose software environment supports four kinds of activities. The *algorithm designer* provides the programs that are to be animated, identifies key "interesting events" in the programs, and also contributes to the design of graphical representations of the data structures. The *animator* implements *views* that comprise the graphical presentation; these views dynamically change in response to interesting events. The *scriptwriter* uses the high-level command facilities to produce scripts containing specific material for presentation to users. The *user* makes use of these scripts or directly interacts with the dynamic graphical representations of his algorithms.

Essentially, Balsa allows the construction of customized "movies" for illustrating or learning about properties of programs, which can then be viewed passively or experimented with actively. An important feature of this design is that there are distinct and independent phases: writing and debugging the programs that actually draw the pictures (the hard part); deciding how and when these programs should be invoked to get across a particular point (the easy part); and actually using all the software to discover properties of particular programs (the enjoyable part).

We have three primary applications in mind for Balsa: teaching, research, and systems programming. For teaching and research, the most natural application is in dynamically simulating how algorithms are executed, although we can also use Balsa to produce synthetic animations on unrelated subjects. Before Balsa can be used for systems programming, it must support some automatic help in the creation of graphical representations. Our approach has been to first build up a basis of fundamental material for use in teaching about algorithms and to use that material whenever possible in research on the design and analysis of algorithms. In the future, we plan to draw upon these experiences and this basis material to build a system that can be useful as a general tool in computer science research and systems programming applications.

In this article, we concentrate on our roles as algorithm designers, animators, and scriptwriters, while using Balsa as the fundamental communications medium in teaching a course on algorithms and data structures, and as an integral tool for algorithm design and analysis. In the next section we describe our innovative classroom environment and the concurrent development of a "dynamic book" to accompany the course textbook. The Balsa environment frees us to experiment with a variety of different techniques for animation and graphical presentation of material.

A dynamic book in an electronic classroom

Balsa provides a new medium for communication between instructor and student and between researcher and colleague. Based on our experience with prototype versions of Balsa and prototype applications,[2] we set out in the fall of 1983 to use this medium in an integral way in teaching a third semester course on algorithms and data structures in our instructional workstation laboratory. Most of the "courseware" was developed in the fall of 1983; it was refined when the course was taught again in the fall of 1984.

Basically, during each lecture all students would be Balsa users, so that

Typically, the instructor would base his lecture on a running commentary of the simulation.

the instructor could present material using dynamic graphical presentations rather than relying on traditional static media such as a chalkboard or viewgraphs. The modus operandi was for each student's workstation to replay a "script" of material that we had previously created and saved. At key points, the script would cause the system to pause and wait for the student to continue when signalled by the instructor (see the section on a high-bandwidth network below for more details). Typically, the instructor would base his lecture on a running commentary of the simulation. We used a traditional viewgraph presentation to supplement the dynamic script, to highlight central ideas, and to provide a focus for the class as a whole (instead of just having individuals interacting with their own machines).

We did not intend that students should be able to understand what was happening on the screen without help from the instructor. On the contrary, we used the system as a particularly rich new medium to teach concepts that might be difficult or impossible to explain any other way. We made the scripts used during lectures available for students to playback for review after the lectures. The scripts also provided a starting point for the students to interact with the material in their own way.

The text,[3] and therefore our dynamic simulations, covered basic algorithms in various fundamental areas of computer science: mathematical methods (random numbers, curve fitting, Gaussian elimination); sorting (internal and external methods); searching (trees, balanced trees, hashing, external methods); strings (string and pattern matching, parsing, file compression, cryptology); geometry (convex hulls, range searching, line intersection, closest points); graphs (basic searching techniques, connectivity, network flow, matching); and such areas as algorithm machines and dynamic and linear programming.

The figures in this article are examples of the material that was presented in the classroom. Of course, it is difficult to do justice to the material with a few static pictures because the essence of many of the presentations is their dynamic character. Specific descriptions of the displayed information are included with the figure captions; these are necessarily quite terse, but it is hoped that the reader will be able to visualize the dynamic presentation of familiar algorithms. The figures are not presented in any particular sequence; in fact, the figures and captions can be read independently of the body of this article. As we discuss a topic, we illustrate our points by referring to *all* the relevant figures.

The two major tasks that we faced when preparing the material for each lecture was: how to conceptualize a graphical representation of the program that would convey the essence of the algorithm and its data structures; and what would make a good script that could be integrated into the lecture. Each graphical simulation required perhaps 15 to 25 hours of programming time (by faculty and staff members) to implement the various algorithms and views, and about one to two hours to develop a script. Each script could be played back in about 15 minutes, but certainly filled a lecture when augmented with pauses and explanations. This ratio is quite reasonable and reflects the future importance and use of the two components: the algorithms and views form a basis for direct interaction or for the creation of scripts for other teaching and research applications; the scripts form a basis for a designated lecture and the subsequent student review, or for a researcher's illustration of a particular point to another researcher.

By the end of the semester, we found ourselves with an operational "dynamic book" to accompany the text, complete with a graphical table of contents comprised of icons from the various animations. By selecting one of the icons, the "reader" can invoke the corresponding lecture script, either for passive playback or for interaction after partial playback. There are other interesting possibilities. For example, it is nearly technically possible to integrate the page images from the text into the dynamic book, and allow the reader to browse through textual, graphical, and animated material online.[4]

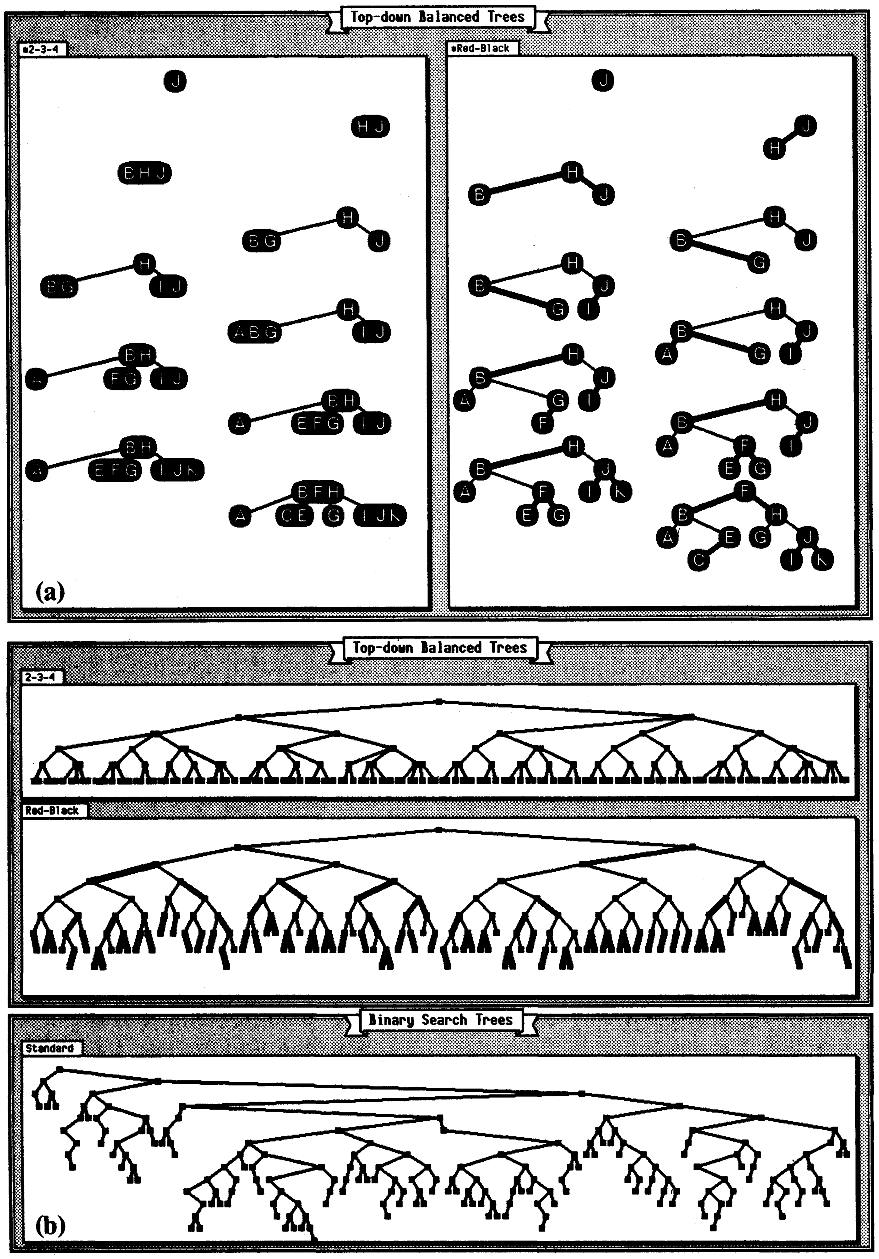


Figure 1. Binary search trees and balanced trees. (a) The image shows the "history" as each node is added to a balanced 2-3-4 tree. The *2-3-4 view at the left is the textbook representation of the 2-3-4 tree; the *Red-Black view at the right is the representation that is typically used for implementation (the red links are the thick edges). A full discussion of this can be found elsewhere.[6] (b) The image shows a comparison of the standard (unbalanced) binary tree algorithm and a balanced 2-3-4 tree algorithm on a moderately large file (about 200 keys). There are two views of the balanced tree: the top is a 2-3-4 tree representation; the bottom, a red-black tree.

A primary goal in the development of the dynamic book material was to create a basis upon which research applications could be built, so that the kind of dynamic interactive graphics used in the classroom could also be used in research. Certainly, the research applications with the most success potential are those that are closest to the fundamental algorithms covered. Already, Balsa has proven to be instrumental in the design and analysis of graph-searching algorithms,[5] sorting algorithms,[6] and priority queue algorithms.

Techniques

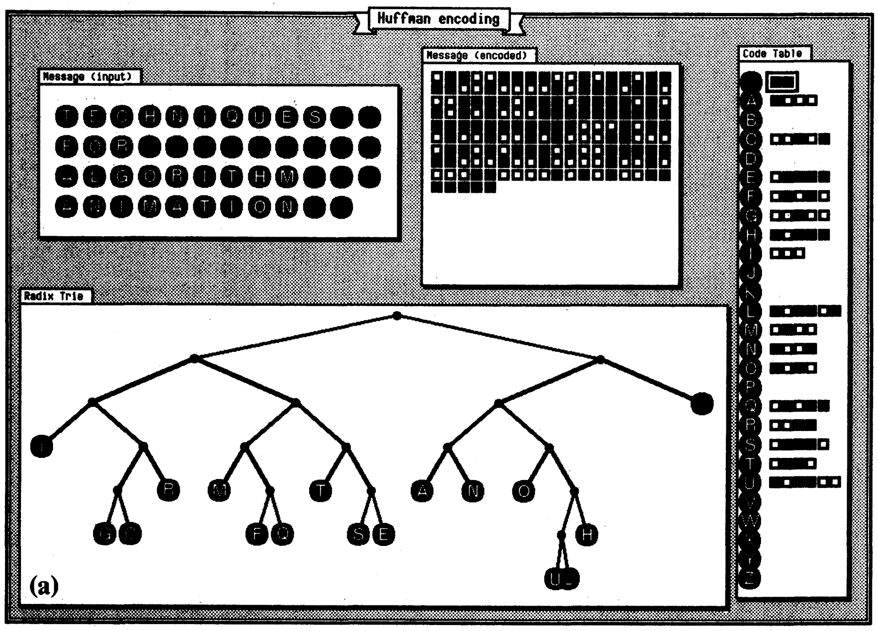
In this section, we describe the basic techniques that we found most useful for exposing the properties of programs through dynamic simulations. Most of the issues we discuss involved making use of basic capabilities in the initial design of the Balsa system. However, there were also several surprises, and our experience has given us many ideas for the design of future versions of Balsa.

Graphical artifacts. A major portion of our effort went into determining precisely how the various data structures and algorithms should be represented graphically. With Balsa, it is easy to experiment with several different representations before settling on a final design. Often, the dynamic nature of the algorithm would be the determining factor, so the capability to experiment was essential. For example, we learned quite early that different graphic representations were needed for different sorting algorithms, even though the underlying data structure (an array of numbers) was essentially the same. We chose the representation that best exposed the fundamental dynamic characteristics of the particular algorithm.

Initially, we planned to use only *graphical* representations for programs and data structures. This strategy was based on our preliminary experimentation that showed that it was essential to execute algorithms on a large set of input data to expose their true characteristics. Also, with a large amount of data, there was not enough screen real-estate to display it textually. We soon found out that users unfamiliar with the algorithms became quickly confused by an abstract representation, so that it was always necessary to first do a small case using a conventional textbook representation, then move into the more interesting and exciting dynamic views. Figures 1, 2, and 3 are examples of lectures that start out with a step-by-step presentation of a small case, and then progress into larger, more revealing views.

One principle that we quickly adopted was that it is best to stay "close" to the data structure, when possible, in designing views. Not only is this the path of least resistance, but also it tended to result in more revealing views. This principle is illustrated in the tree views in Figures 1 and 4. We began by displaying trees in the conventional manner, with the root appearing in the middle of the window, the left child of the root in the middle of the left half of the window, etc. This turned out to have the disadvantage that nodes at the bottom of the tree become crowded together after only six or seven levels, even if there were relatively few nodes in the tree. Accordingly, we turned to the method of determining the horizontal position of a node by counting the number of nodes that would appear before it. This is quite easy to implement, and it provides an alternative way of looking at the tree that is perhaps more revealing of its actual structure. Most importantly, it allows dynamic presentation of various algorithms without redrawing the whole tree after every modification.

For example, in Figure 1, it is easy to visualize how the standard binary search tree can be presented dynamically, because once a node has been inserted into the tree, its position is fixed. However, the red-black balanced search tree must be redrawn entirely each time a node is inserted because many nodes can potentially change positions. With more powerful hardware and more sophisticated graphics packages, we could show smooth transformations of the tree, such as those shown in the movie, *PQ-Trees.*[7] In Figure 4, each node in the binary tree is positioned using the Cartesian



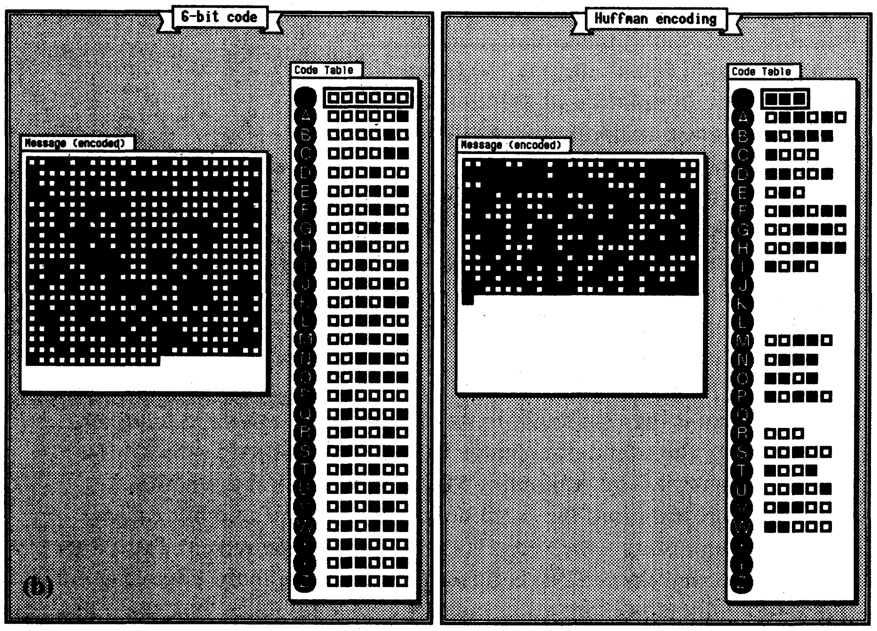
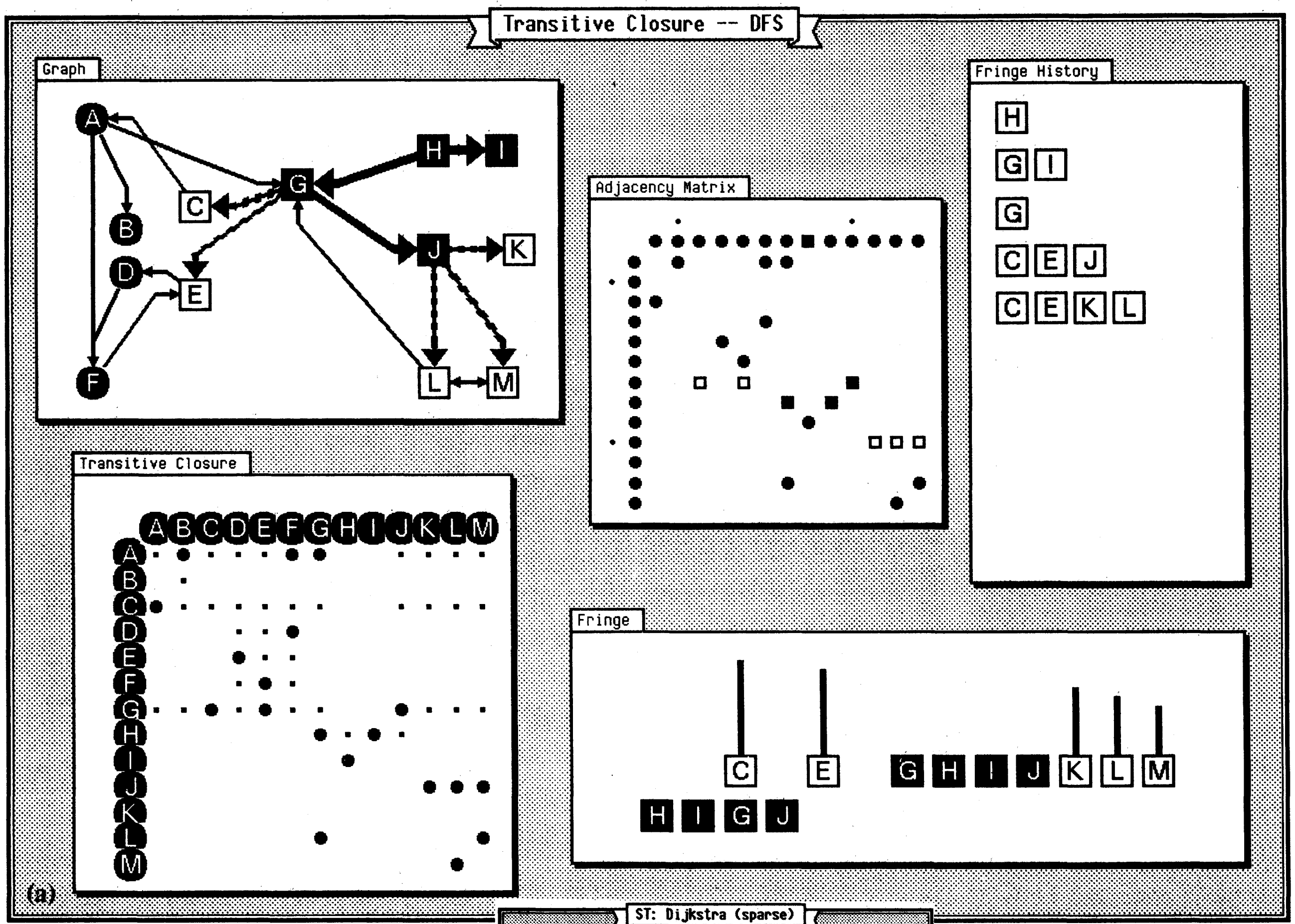


Figure 2. File encoding and compression. (a) The image shows a Huffman encoding for the given message. The Huffman encoding uses a variable number of bits to encode each letter and results in substantial space savings for most text files. The Radix Trie view shows the radix trie that is created from the set of letters comprising the phrase to be encoded. Each letter is stored in an external node of the tree, and the path from it to the root determines its Huffman code. The code for each letter is given in the Code Table view at the right. (b) The image shows a comparison of a 6-bit encoding for each letter of the alphabet with the Huffman encoding on the same message; clearly saving space



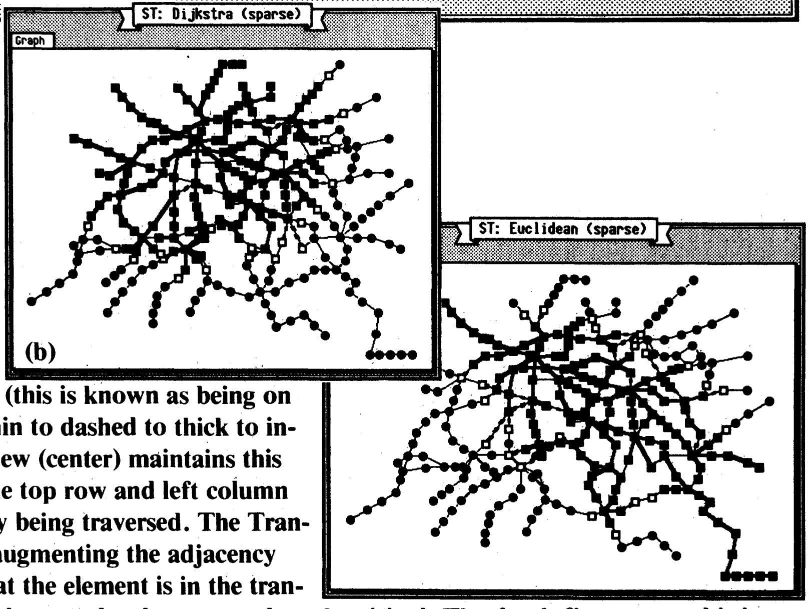


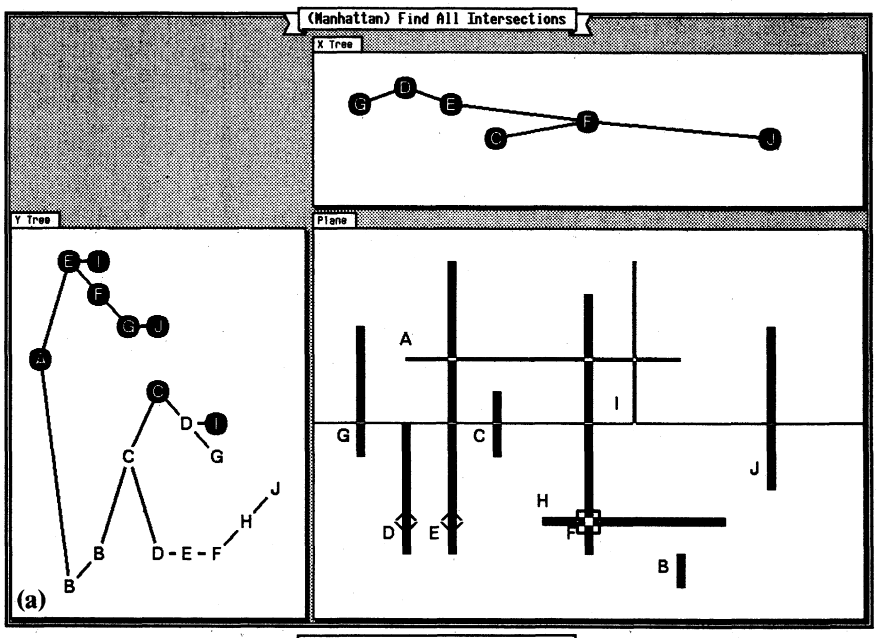
Figure 3. Graph algorithms. (a) The image illustrates the computation of the transitive closure on a small undirected graph. The transitive closure is found by making repeated depth-first traversals of the graph, using each node as the starting point. The Graph view (upper left) shows the depth-first traversal for the given node. The nodes in the graph are displayed as dark circles in their initial "unvisited" state; as dark squares after they have been "visited"; and as light squares if they are ready to be visited (this is known as being on the "fringe"). The edges also change from thin to dashed to thick to indicate the traversal. The Adjacency Matrix view (center) maintains this consistent representation; the small dots at the top row and left column are markers indicating which edge is currently being traversed. The Transitive Closure view (lower left) is formed by augmenting the adjacency matrix view with small squares to indicate that the element is in the transitive closure matrix. The Fringe view shows those nodes that are ready to be visited. The depth-first traversal is implemented using a priority queue to simulate the actions of a stack; the height of each stick above a node corresponds to its priority. The second row of nodes shows the nodes in the order in which they are removed from the queue. The Fringe History view is a history of the queue, at each time that a node is visited. (b) The image shows a comparison of two shortest path algorithms on a graph representing the Paris Metro. The algorithm at the upper-left is the classical Dijkstra's algorithm, while the one at the lower-right is based on a heuristic.[9] The classical algorithm must in essense perform a breadth-first traversal of the entire graph. The heuristic (looking at the distance from the start to the current spot in the tree plus the Euclidean distance to the target vertex) does not need to examine all of the nodes. The image is taken just after the heuristic algorithm has completed; Dijkstra's algorithm still has a ways to go.

coordinate of the point that it represents. We have also experimented with using two-dimensional representations to display and learn about characteristics of very large trees.

Note that the tree representation above requires some advance knowledge of the input. This was not difficult to arrange for the lecturing application, nor is it unreasonable to arrange for many research applications. Several of our animations would involve running an algorithm twice; once to calculate size parameters for the graphic representation, and a second time to actually fill in the display.

Many of our displays involve using graphical cues for "state" information about atomic pieces of data structures. This is best exemplified in Figure 3, which shows different representations used for nodes in graphs (solid box, open box, solid circle, etc.) as they pass through different states while being processed by the algorithm. Other examples are found in Figures 4 and 5. These graphical cues were useful for several reasons. First, in stepping through a program, data structure changes are immediately reflected in the display. Second, by using a consistent representation, we are able to provide a common thread connecting elementary textual views and advanced graphical ones (and a common thread among multiple views; see below). For example, in Figure 3, once one learns about "tree" and "fringe" nodes, etc., the labels of the nodes are not particularly relevant. Third, on large cases, the state changes (if the icons are chosen properly) dynamically expose the character of an algorithm. For example, the difference in the dynamic presentation corresponding to Figure 3 between a breadth-first search sweeping through the graph and a depth-first search probing uncharted areas is striking. (Of course, because the figures in this article are static, some imagination is necessary.)

Multiple views and algorithms. A fundamental property of the Balsa system is its capability for running multiple algorithms simultaneously and for



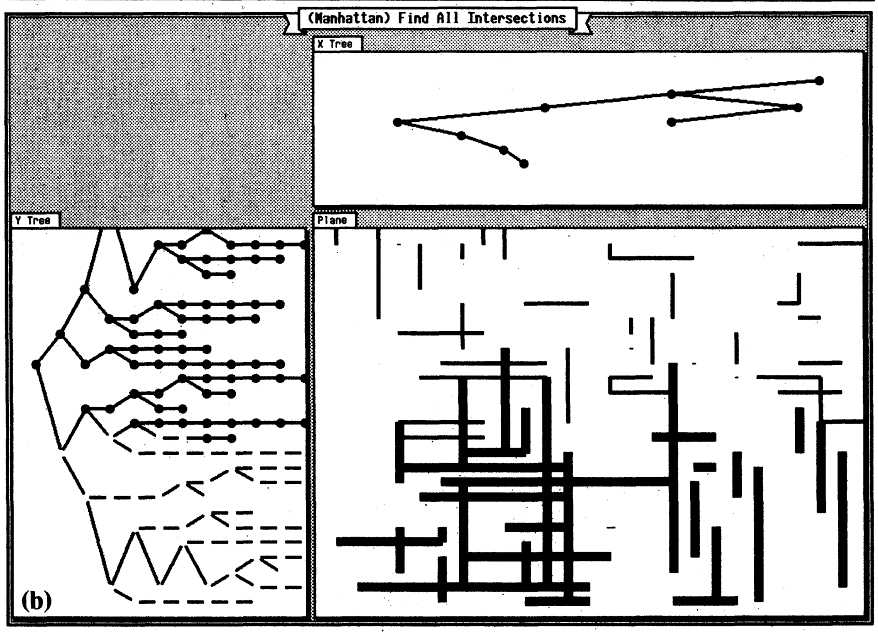
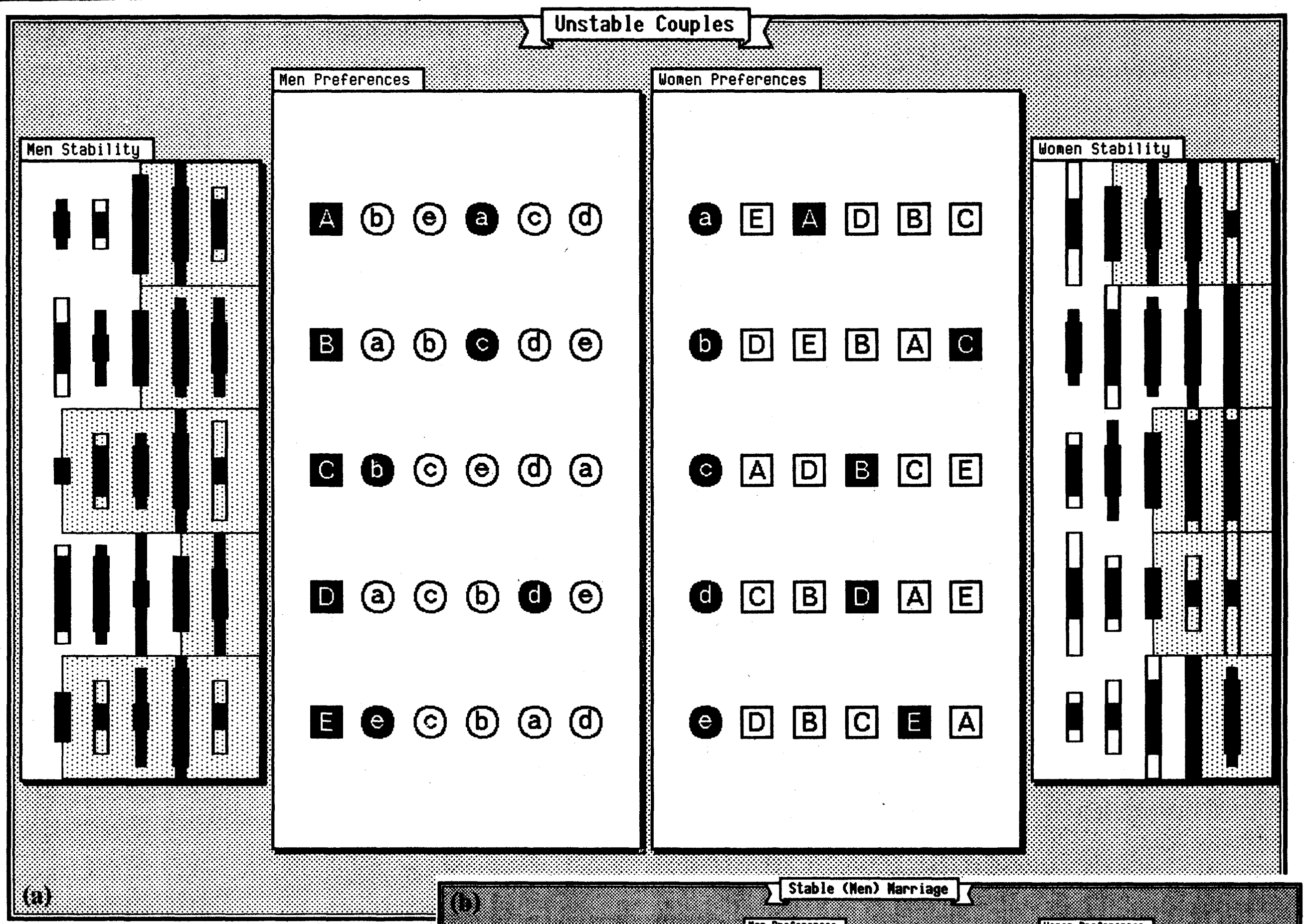


Figure 4. Line intersection. (a) The image illustrates an algorithm for finding points of intersection in a given set of vertical and horizontal lines. The idea of the algorithm is to imagine a horizontal scan line starting at the bottom and moving up. Whenever a horizontal line is encountered, look at each vertical line from a small set of possible candidates and see if its *x* value lies within the endpoints of the horizontal line. If an intersection is found, a small square is drawn; otherwise, a diamond is drawn. The *y*-coordinate of the endpoints of each line is stored in a binary tree, as shown in the Y Tree view at the left. Then, a standard in-order traversal of this tree gives the stopping positions for the scan line, from bottom to top. Note how each node in the tree changes after it has been visited. When the scan line reaches the bottom of a vertical line, the algorithm adds that line to the set of possible candidates. When the top of a vertical line is reached, the algorithm removes that vertical line from the candidiate set. These operations are implemented with a binary tree, using the *x*-coordinates as shown in the X Tree view at the top in each image. Notice that both the *y*-tree and the *x*-tree views display each node using the coordinate of the corresponding line in the plane. (b) The image illustrates the algorithm on a much larger data set.



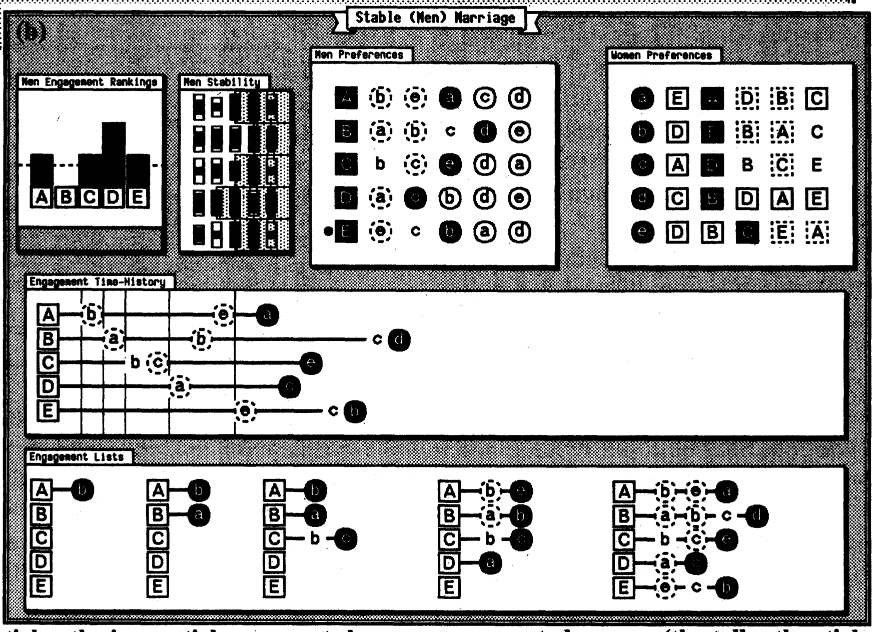


Figure 5. The Stable Marriage problem. (a) In the image, the center views show how each of *N* men (capital letters, square nodes) rate each of the *N* women (lower case letters, circular nodes) as possible mates (the left view) and how each woman rates the set of men (the right view). A set of *N* marriages is performed (shown as darkened circles and squares); a configuration is called unstable if there is a man *m* who prefers a woman *w* to his current mate, and if *w* prefers *m* to her current mate. The graphical view on both sides aid in determining whether a marriage configuration is stable. Consider man *m* and his *k*th choice, woman *w*. In the sticks view at the far left, row *m*, column *k* has two vertical sticks: the inner stick represents how woman *w* rated man *m* (the taller the stick, the better the rating); the outer stick represents how that woman rated her current husband. A "hot-dog" (the inner stick is larger than the outer stick) indicates that *w* would prefer *m* to her current spouse; if *m* also prefers *w* to his spouse the hot-dog, will be in the non-shaded area of row *m*. The far right view is analogous, but from the women's perspective. (b) The image illustrates the dynamics of the algorithm: we go through the men in order, trying to marry each to his best remaining choice. In order to keep everybody stable, we might need to divorce an existing couple (and marry the man to his next choice) or use a man's next choice (if the woman prefers her current mate). At each stage of the algorithm, another couple is married. The middle view in these images is a time-line of all marriages, divorces (dashed nodes), and refused-proposals (clear nodes). The bottom view shows a compressed view of the time-line, but with a "history." Note how the same representation of man, woman, marriage, divorce, etc., is used in all views.

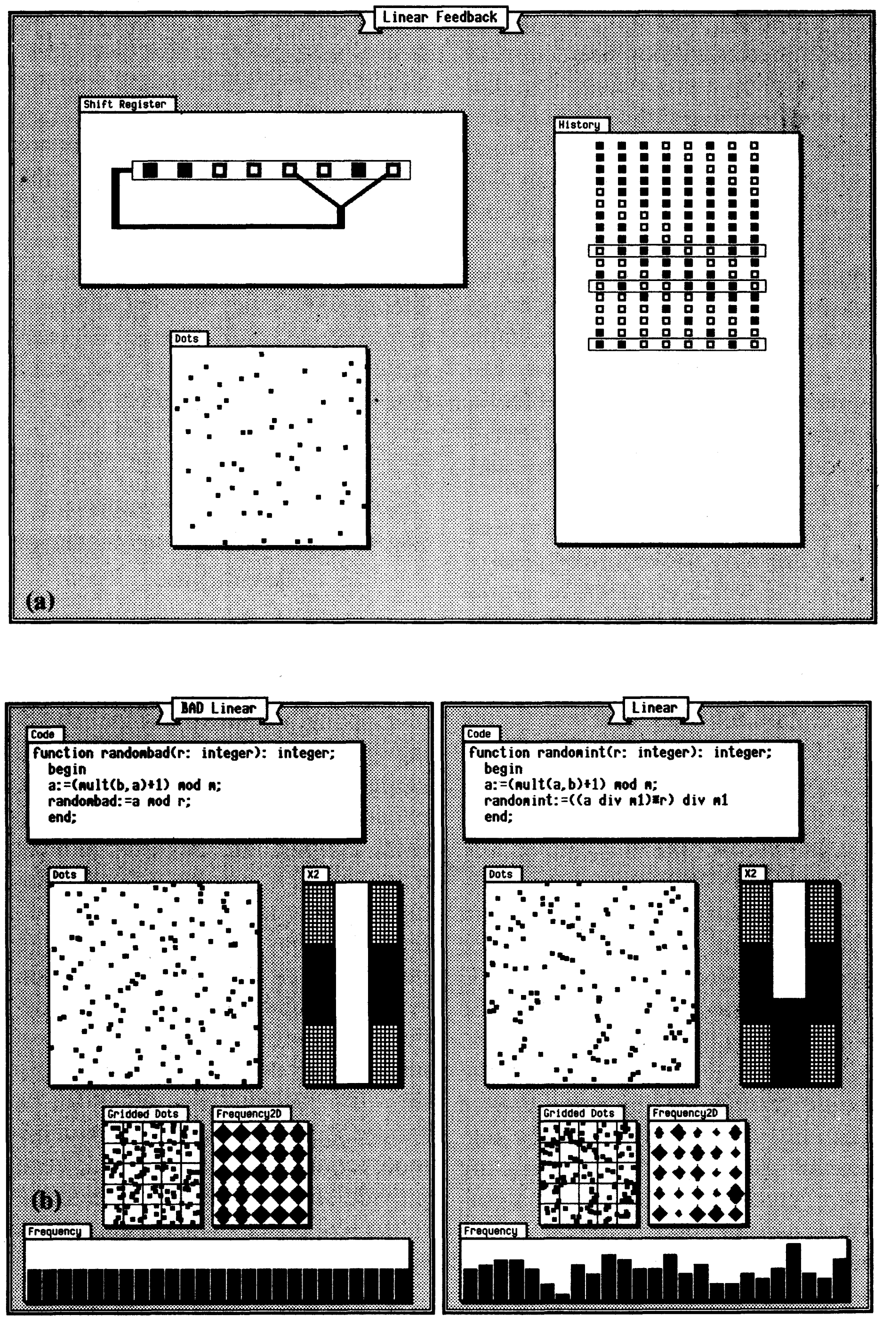
supporting multiple views of the same data structure. This was an important part of many dynamic presentations.

First, as illustrated in Figures 1, 3, and 6, it was useful to simply run two algorithms on the same data for the purpose of comparison. Not only would this type of "algorithm race" usually make a lasting impression on students, but also the contrasting dynamic properties of the algorithms would often make it easier to understand each one's particular properties.

Multiple views of the same data structure provide a thread connecting different graphic representations, thus providing a convenient mechanism for moving to more advanced representations. For example, Figure 1 shows two different representations for balanced trees: one appropriate for considering the high-level algorithm (the 2-3-4 view), and one appropriate for considering low-level implementations (the red-black tree view). When both views are displayed simultaneously as the algorithm is running, the semantics of the representation become obvious.

In many cases, it was surprisingly easy to use multiple views of a trivial nature to put together an ensemble of dynamic images that together show the operation of an algorithm. Examples of this may be found in Figures 3, 4, and 6. A simple graphical display showing the value of a crucial control variable (or an array of such variables) sometimes can make the difference in the effectiveness of a presentation. For example, in Figure 4, the position of the nodes in both trees matches the coordinates of the line that they are representing when displayed in the view of the plane. These three independent simple views together provide the *gestalt* of the way that the algorithm is progressing. (This example also raises difficult questions about the amount of coordination appropriate between particular instantiations of views, which we have not yet fully addressed.)

Scripts. In early prototypes of the Balsa system, we found it convenient to include a crude capability for saving and invoking sequences of basic operations, called *scripts*. Its main applica-



```
function randombad(r: integer): integer;
  begin
  a:=(mult(b,a)+1) mod m;
  randombad:=a mod r;
  end;
```

```
function randomint(r: integer): integer;
  begin
  a:=(mult(a,b)+1) mod m;
  randomint:=((a div m1)*r) div m1
  end;
```

Figure 6. Random number generators. (a) The image shows a generator based on a linear feedback shift register. The Dots view uses the numbers generated to plot points in Cartesian space.

(b) The image compares two linear congruential methods: the algorithm on the left uses the least significant bits, whereas the algorithm on the right uses the most significant bits. The implementation code is displayed. Note that the points produced by both random number generators appear equally "random" and well-distributed in the square.

Below the dots is another instantiation of the dots, but with grid markers partitioning the square. The distribution of points within each partition is shown both two-dimensionally (as diamonds) and as a more conventional histogram in row-major order (the bottom view). The rightmost view is the x^2 (chi-square) test to measure how "random" the numbers are, with the center dark region indicating the "passing" zone. The algorithm on the left produces numbers that are distributed too perfectly, an indication of non-randomness.

tion was in demonstrating the system to interested parties; we wanted to be able to set up a preassigned sequence of commands showing off the capabilities of the system, then let it run unattended. Including this capability was indeed fortunate, because it turned out that scripting was a fundamental ingredient in all of the material that we developed. The script is what brings the views out from the system to the user. For the naive user, it provides worked-out examples designed by the programmer; for the advanced user, it provides a catalog of available visualizations and dynamic modes.

Scripting is currently done at a primitive level: a keystroke history is saved into a file which can then be played back. A number of additional facilities exist: saving a keystroke "FutureFreeze" will cause the script to pause in the playback phase and wait for the user to press a "Continue" key; the keystroke "FuturePause" will break out of a script in order to let the user use the current worldspace of the system and then return to the script later.

As a digression, it is interesting to note that the Balsa user-interface (modeled after the Xerox PARC Smalltalk system[8]) was designed to be self-disclosing through pop-up menus. All interaction is done through the mouse (except for the rare instance when text must be entered at the keyboard). However, when creating the script, repeatedly searching for a button on a menu proved to be quite cumbersome. Thus, we added a programmable keyboard feature that allowed us to quickly enter the common commands such as "Go," "FutureFreeze," or "SingleStep" as single keystrokes, such as control-G, control-F, and control-S, respectively. Since a script may consist of hundreds of such commands, this facility proved to be a timesaver for sophisticated users.

Our procedure for building scripts was as follows. First, we would become familiar with the algorithms, views, and types of input that were available. Next, we would storyboard a series of *scenes*. Each scene would comprise a configuration of algorithm and view windows, ideas of what size and type of input would be best, and ideas of what places in the execution of the algorithm(s) would be appropriate places to insert "FutureFreeze" or "FuturePause" commands. Then, we would build a small script for each scene (in its own file) by restoring a previously saved set of windows, and run the algorithm(s) using all of Balsa's interactive facilities. Finally, we would merge these small scripts for each scene into one large script, and make that script available to users.

Scripting was a fundamental ingredient in all of the material developed.

Now that we realize the importance of scripting, we have plans for extending the available capabilities in future versions of the system. For example, research needs to be done on editing these scripts. Although a readable transcript of keystrokes is saved, editing the transcript file has severe side effects that may not be obvious from just looking at the keystrokes. Certain syntactic changes may lead to undefined semantics. One approach to this might be to use a generalized undo-skip-redo facility,[9] but this might be cumbersome for long scripts. Another more promising possibility would be to provide some graphic representation of the script itself, with corresponding editing aids.

Static history. One surprising fact that we did not discover until we had some actual experience was that a visual *history* of the execution of an algorithm was essential for studying the algorithm on "small cases." Moreover, as mentioned above, for the small cases, we were better off with textual examples rather than graphical representations (Figures 1 and 3, as noted above).

It is relatively straightforward to attach a "history view" to any existing Balsa view. Unfortunately, this must currently be done by the animator when the view is actually coded. It would be better to provide the user with interactive tools to specify how a history should be kept.

The code view. Programs themselves are displayed by a versatile part of the view library called the *code view.* In general, a user may open a window containing a code view of an algorithm to watch its line-by-line execution. Upon entry to a procedure, a new subwindow containing code for the new procedure overlays the code of the calling procedure (offset to the right and down). When a procedure exits, its code window is removed, and the caller's state reappears, exactly as it had been before the call. This scenario has proved successful for several lectures (especially those dealing with recursive algorithms) as well as for advanced researchers debugging new algorithms.

Because the code view is triggered by interesting events rather than interpreting the code, we have been able to use it in diverse and surprising ways. For algorithms whose displays are effective primarily because of the speed of execution, it is rare that the code view with lines flashing by is of any use. Thus, we can have a streamlined version of the algorithm, with perhaps a handful of judiciously chosen lines highlighted. Moreover, we can choose not to highlight any lines of code, but to show only procedure calls. Alternatively, we can show multiple textual versions of the algorithm, each with a different level of detail. This separation of the view from the actual code provides flexibility which has proven to be helpful in many teaching and research applications.

We have experimented with a variety of styles for highlighting lines of code. The most effective style has been to invert the area of screen containing the text; the less effective styles were to change the font (from say, bold to italics) and to draw a box around the screen area. We have also experimented with displaying the procedures side-by-side (vertically or horizontally) so they didn't overlap; displaying all procedures simultaneously; and replacing the caller procedure by the caller. Overlapping boxes seem to be the most effective because they provide a context separate from the sequence of callers, yet they do not use an excessive amount of screen real estate. Another method that we plan to try is the "fisheye" view[10] of large files. This is analogous to a very wide angle camera lens that gives full details of the area in focus, while retaining the global high-level structure of the remainder.

Current work is underway to investigate graphical representations of a program,[11] and also graphic design techniques to produce very pretty (though textual) listings of a program.[12] For large systems, it might be nice to display each procedure in a box of a module hierarchy chart (using a topological sort and other constraints to determine the screen layout). Our framework allows these and other views, as well as their animation.

These textual views have proven to be helpful to a wide range of users. For programs of any complexity, the capability of the personal workstation to display and change many lines of text quickly at arbitrary positions on the screen is crucial.

High-bandwidth network

Balsa is designed to exploit the high-bandwidth, local-area network interconnecting all of the machines in the instructional workstation laboratory. To date, we have used the system only in two relatively straightforward ways: either all machines run independently, playing back Balsa scripts, with students proceeding from pauses on cue from the instructor; or the system is in "broadcast" mode with all nodes slaved to the instructor node. In broadcast mode, students often found the instructor's sudden changes of context disconcerting (in playback mode, students were able to linger, for at least a few seconds, on difficult images). On the other hand, the instructor had much more control of the class in broadcast mode.

From a system standpoint, broadcasting is analogous to scripting. In broadcasting, the master's (usually the instructor's) keystrokes are saved in a multiprocessed file that the slaves (usually the students) are simultaneously reading in playback mode; in scripting, user keystrokes are saved in a file to be read at a later time in playback mode. We considered broadcasting an image of the master bitmap, or tapping the broadcasting at a very low level (for example, at the graphics display primitive), but rejected these because of the potential load on the network. Our method of broadcasting also gives us the potential to have an instructor monitor several students simultaneously, by having the students become the master and displaying their output in a separate window on the instructor's screen.

For programming assignments outside of class, the network is vital.

The broadcast mode has a number of intrinsic synchronization problems. First, if the instructor on the master node interrupts the executing program, how do the viewers know this? Our solution is for the master to set a flag at each potential interruption point (which would roughly correspond to each line of code), and broadcast that flag to all viewers. Another problem is the relative speeds of the machines. Obviously, no viewer can execute at a faster rate than the master. However, what if the slave machine is too slow (perhaps it has less memory, an older and slower disk, etc.)? Should the slave send a "wait-for-me" message and response to each message sent by the master? If so, that means that the class will go only as fast as the slowest machine, which might be intolerably slow.

When the instructional workstation laboratory is not being used as a classroom, machines in it are used for programming assignments for several courses. Here, the network is vital. The computer supports network-wide, demand-paged, virtual memory, with a hierarchical file system naming-space transparent over the network.[13] The practical significance of this is that a student accesses his files in the exact same way regardless of which machine he is on, and students have access to the lecture scripts outside of class in the same way as during class. Generally, explicit file transferral is avoided.

We have only begun to consider ways in which the network can be used to provide new paradigms for education in the classroom. It is important to note that most of our effort—which went into writing the views for the algorithms—will be unaffected by any added functionality. Only the scripts —which represent a comparatively small amount of effort—will need to be updated.

Certainly, there is a close match between the algorithms/data-structures material discussed here and the orientation and capabilities of our hardware and software systems. It might legitimately be questioned whether one might expect to be as successful in animating some other body of material. Actually, the instructional workstation laboratory has already been used for teaching several other courses:[14] an introductory Pascal course has been taught three times using Balsa demos; a differential equations course has been taught using Balsa demos for displaying mathematical surfaces; and a neural science course has been taught using a special software system to see cross-sections of the brain. Several more uses are planned.

When we first began developing Balsa, we had several prototype simulations of sorting algorithms in operation, motivated by the movie, *Sorting out Sorting*.[14] It was, legitimately, questioned at that time whether we could do anything beyond sorting. In sorting, we saw the potential to do algorithms; in doing algorithms, we see the potential to tackle other domains.

We have plans for extending the system's capabilities by assessing the most successful presentation techniques discussed above and adding capabilities to them. Examples include support for creating static history views at the user level and extended support for scripting.

One of our long-term goals is to be able to automatically animate programs. For the near-term, we are developing tools to simplify the animation process. For example, it seems possible to eliminate the animator from the loop, and allow the animations to be done by an algorithm designer (by just specifying interesting events and names of library views), or even by a user (by pointing to a data structure declaration in a program and then picking a view from a list of available views for that type of data structure).

Many of our images suffer from a lack of resolution or become unacceptably slow on large cases, so we are interested in investigating the use of more advanced hardware and software systems for Balsa-like activities. In particular, preliminary studies indicate that color will play an important role in the future, especially in dynamic displays.

On the other hand, we can apply many of the techniques that we have learned to much less powerful systems, so we are also interested in investigating the use of less powerful personal computer systems for Balsa-like activities. While this will mean giving up many of the advanced features to which we have become accustomed, we feel that our experience will enable us to utilize personal computers, such as the IBM PC or the Apple Macintosh, for algorithm animation. Furthermore, we have the capabilities of Balsa available as a test-bed for alternative animation techniques that might be appropriate in less rich environments.

Finally, we believe that the wide range of dynamic images that we have created and the ways we provide users to interact with them suggest a number of intriguing possibilities on using personal computer systems as a supplement, or even as an alternative, to traditional printed media. We expect to explore this area much more deeply in building upon the material described here, and in developing more advanced versions of our system. □

Acknowledgments

Many people have participated in the implementation of Balsa and the underlying software. Without trying to mention all of the participants, we do want to acknowledge the efforts of Steve Reiss, Joe Pato, and Mike Strickman. Special thanks to Janet Incerpi and Steve Feiner for their usual meticulous readings and insightful comments on drafts of this article.

Partial support for this research was provided by the Exxon Education Foundation, and by the ONR and DARPA under Contract N00014-83-K-0146 and ARPA Order No. 4786. Partial equipment support was provided by NSF Grant SER80-04974 and by Apollo Computer, Inc. Partial support for the second author was provided by NSF Grant MCS-83-08806.

References

1. Marc H. Brown and Robert Sedgewick, "A System for Algorithm Animation," *Computer Graphics*, Vol. 18, No. 3, July 1984, pp. 177-186.
2. Marc H. Brown Norman Meyrowitz, and Andries van Dam, "Personal Computer Networks and Graphical Animation: Rationale and Practice for Education," *ACM SIGCSE Bulletin,* Vol. 15, No. 1, Feb. 1983, pp. 296-307.
3. Robert Sedgewick, *Algorithms,* Addison-Wesley, Reading, MA, 1983.
4. Steven Feiner, Sandor Nagy, and Andries van Dam, "An Experimental System for Creating and Presenting Interactive Graphical Documents," *ACM Trans. Graphics,* Vol. 1, No. 1, 1982, pp. 59-77.
5. Robert Sedgewick and Jeffrey S. Vitter, "Shortest Paths in Euclidean Graphs," *Proc. 25th Ann. Symp. Foundations Computer Science,* Nov. 1984.
6. Leo Guibas and Robert Sedgewick, "A Dichromatic Framework for Balanced Trees," *Proc. 19th Ann. Symp. Foundations Computer Science,* Oct. 1978, pp. 8-21.
7. Kellogg Booth, *PQ Trees,* 16mm color silent film, 12 minutes, 1975.
8. Larry Tesler, "The Smalltalk Environment," *BYTE,* Vol. 6, No. 8, Aug. 1981, pp. 90-147. (The interested reader should refer to the Smalltalk books by Adele Goldberg et al., published by Addison-Wesley, 1983.)
9. Jeffrey S. Vitter, "US&R: A New Framework for Redoing," *IEEE Software,* Vol. 1, No. 4, Oct. 1984, pp. 39-52.
10. George W. Furnas, "The FISHEYE View: A New Look at Structured Files," unpublished manuscript, Bell Laboratories, Murray Hill, NJ, 1984.
11. Steven P. Reiss, "PECAN: Program Development Systems that Support Multiple Views," *Proc. 7th Int'l Conf. Software Engineering,* Mar. 1984.
12. Ronald Baecker and Aaron Marcus, "On Enhancing the Interface to the Source Code of Computer Programs," *Proc. Computer-Human Interface,* Dec. 1983, pp. 251-255.
13. David L. Nelson and Paul J. Leach, "The Evolution of the Apollo DOMAIN," *Proc. 17th Hawaii Int'l Conf. System Sciences,* Jan. 1984, pp. 470-479.
14. Marc H. Brown, and Robert Sedgewick, "Progress Report: Brown University Instructional Computing Laboraory," *ACM SIGCSE Bulletin,* Vol. 16, No. 1, Feb. 1984, pp. 91-101.
15. Ronald Baecker, *Sorting out Sorting,* 16mm color sound film, 25 minutes, 1981.

OPENINGS FOR ASSISTANT PROFESSOR

The Department of Computer Science of the Swiss Federal Insitute of Technology in Zurich has two openings for assistant professors. We are looking for young computer scientists actively engaged in experimental research and interested in teaching.

Research areas of particular interest to the department include:
- programming techniques
- computer networks
- operating systems
- VLSI design
- CAD/CAM
- computers in education

Candidates are invited to submit an application (including vita, list of publications, references, research interests) before March 1, 1985 to the President of ETH Zurich, Prof. H. Ursprung, ETH-Zentrum, 8092 Zurich, Switzerland.

For further information write to the Chairman of the Department of Computer Science, Prof. J. Nievergelt, ETH-Zentrum, 8092 Zurich, Switzerland.

Further Reading

Ronald Baecker, "Two Systems Which Produce Animated Representations of the Execution of Computer Programs," *ACM SIGCSE Bulletin*, Vol. 7, No. 1, Feb. 1975, pp. 158-167.

Alan Borning, "ThingLab—A Constraint Oriented Simulation Laboratory," report SSL-79-3, Xerox PARC, Palo Alto, Calif., July 1979.

John M. Chambers, William C. Cleveland, Beat Kleiner, and Paul A. Tukey, *Graphical Methods for Data Analysis,* Duxbury Press, Boston, MA, 1983.

Mark S. Dionne and Alan K. Mackworth, "ANTICS—A System for Animating LISP Programs," *Computer Graphics and Image Processing,* Vol. 7, 1978, pp. 105-119.

Laura Gould and William Finzer, "Programming by Rehearsal," report SCL-84-1, Xerox PARC, Palo Alto, Calif., May 1984.

Robert F. Gurwitz, Read T. Fleming, and Andries van Dam, "MIDAS: A Microprocessor Display and Animation System," *IEEE Trans. Education,* Feb. 1981.

Christopher F. Herot, et al., "An Integrated Environment for Program Visualization," *Automated Tools for Information Systems Design,* H. J. Schneider and A. I. Wasserman, eds., North Holland Publishing Co., 1982, pp. 237-259.

Kenneth C. Knowlton, *L6: Bell Telephone Laboratories Low-Level Linked List Language,* two black and white sound films, 1966.

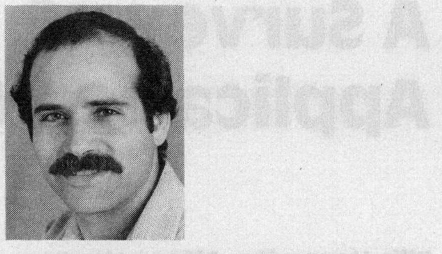
Henry Lieberman and Carl Hewitt, "A Session with TINKER: Interweaving Program Testing with Program Design," *Proc. Lisp Conf.,* Aug. 1980, pp. 90-99.

Brad A. Myers, "Displaying Data Structures for Interactive Debugging," technical report CSL-80-7, Xerox PARC, Palo Alto, CA, 1980. (A summary of this report appeared as "Incense: A System for Displaying Data Structures," *Computer Graphics,* Vol. 17, No. 3, July 1983.)

Mark Noriconi and Amy L. Lansky, "Representation and Refinement of Visual Specifications," *Software Validation,* ed. H. L. Hausen, Elsevier-North Holland, 1984, pp. 229-251.

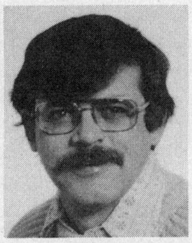
Bernhard Plattner and Jurg Nievergelt, "Monitoring Program Execution: A Survey," *Computer,* Vol. 14, Nov. 1981, pp. 76-93.

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



Marc H. Brown is currently a candidate for his PhD at Brown University under an IBM Graduate Student Fellowship. His research interests include program visualization and computer science education. He serves as the director of the Association of Computer Science Leagues, Inc., a non-profit corporation that administers monthly computer science contests to junior and senior high school students in the US and Canada.

Brown received his ScB and ScM degrees in computer science from Brown University in 1980 and 1982, respectively. From 1980 to 1984, he was responsible for the Instructional Workstation Laboratory at Brown.



Robert Sedgewick is currently professor of computer science at Brown University where he has been a member of the faculty since 1975. His research interests include mathematical analysis of algorithms, design of data structures and algorithms, and program visualization. He has published widely in these areas, and is the author of the textbook *Algorithms*.

Sedgewick received his PhD degree from Stanford University in 1975. He has held visiting research positions at Xerox PARC, the Institute for Defense Analyses, and INRIA in France. He is a member of the ACM, the EACTS, and SIAM. He is an editor of the *Journal of the ACM* and the *Journal of Algorithms.*

The authors may be reached at the Department of Computer Science, Brown University, Providence, RI 02912. Electronic mail correspondence may be sent to mhb%brown@csnet-relay.

TOTAL CONTROL:

***FORTH:* FOR Z-80®, 8086, 68000, and IBM® PC**

Complies with the New 83-Standard

GRAPHICS • GAMES • COMMUNICATIONS • ROBOTICS DATA ACQUISITION • PROCESS CONTROL

- ***FORTH*** programs are instantly portable across the four most popular microprocessors.
- ***FORTH*** is interactive and conversational, but 20 times faster than BASIC.
- ***FORTH*** programs are highly structured, modular, easy to maintain.
- ***FORTH*** affords direct control over all interrupts, memory locations, and i/o ports.
- ***FORTH*** allows full access to DOS files and functions.
- ***FORTH*** application programs can be compiled into turnkey COM files and distributed with no license fee.
- ***FORTH*** Cross Compilers are available for ROM'ed or disk based applications on most microprocessors.

Trademarks: IBM, International Business Machines Corp.; CP/M, Digital Research Inc.; PC/Forth+ and PC/GEN, Laboratory Microsystems, Inc.

FORTH Application Development Systems include interpreter/compiler with virtual memory management and multi-tasking, assembler, full screen editor, decompiler, utilities and 200 page manual. Standard random access files used for screen storage, extensions provided for access to all operating system functions.

Z-80 FORTH for CP/M® 2.2 or MP/M II, $100.00; **8080 FORTH** for CP/M 2.2 or MP/M II, $100.00; **8086 FORTH** for CP/M-86 or MS-DOS, $100.00; **PC/FORTH** for PC-DOS, CP/M-86, or CCPM, $100.00; **68000 FORTH** for CP/M-68K, $250.00.

FORTH + Systems are 32 bit implementations that allow creation of programs as large as 1 megabyte. The entire memory address space of the 68000 or 8086/88 is supported directly.

PC FORTH +	$250.00
8086 FORTH + for CP/M-86 or MS-DOS	$250.00
68000 FORTH + for CP/M-68K	$400.00

Extension Packages available include: software floating point, cross compilers, INTEL 8087 support, AMD 9511 support, advanced color graphics, custom character sets, symbolic debugger, telecommunications, cross reference utility, B-tree file manager. *Write for brochure.*

VISA LMI ***Laboratory Microsystems Incorporated***
Post Office Box 10430, Marina del Rey, CA 90295
Phone credit card orders to ***(213) 306-7412*** MasterCard