Progress Report: Brown University Instructional Computing Laboratory†

Marc H. Brown
Robert Sedgewick

Dept. of Computer Science
Brown University
Providence, RI 02912

Abstract: *An instructional computing laboratory, consisting of about 60 high-performance, graphics-based personal workstations connected by a high-bandwidth, resource-sharing local area network, has recently become operational at Brown University. This hardware, coupled with an innovative courseware/software environment, is being used in the classroom in an attempt to radically improve the state of the art of computer science pedagogy. This paper describes the current state of the project. The hardware and courseware/software environments are described and their use illustrated with detailed descriptions, including sample screen images. Some comments are included on our initial reactions to our experience to date with the environment and on our future plans.*

1. Introduction

Several years ago, in anticipation of rapidly developing improvements in available hardware and software environments, the Department of Computer Science at Brown University embarked on a project designed to put the best possible technology to good use in the classroom. Projections at that time indicated that it would be feasible to put a large number of very powerful graphics-based computer systems in a lecture hall, and we began to plan for the use of such a system in our teaching programs.

Based on previous experience, we felt that high-resolution graphics could be used to provide detailed pictures of programs, data structures, and representations of other concepts commonly taught in computer science; that powerful computer systems supporting the displays could be used to expose the dynamic characteristics essential to such concepts; and that a very high bandwidth connection among the systems could facilitate exploration of new interactive teaching modes. More details on the rationale for the development of such an environment may be found in last year's proceedings of this conference [Brow83].

Actually we have been able to achieve many of the things that we planned: in the fall of 1983, we opened an Instructional Computing Laboratory with about 60 high-performance Apollo computer systems that is used throughout our curriculum, including classroom use for our introductory programming course and for our algorithms course. The purpose of this paper is to give some details on the development of this laboratory and on its use in classroom teaching.

The next section describes the laboratory hardware in detail. Section 4 describes our current software/courseware environment. Section 5 presents two typical lectures: the first one is on binary trees and is suitable for an introductory programming course or a course in algorithms and data structures. The second example is on range searching, an advanced use of a binary tree structure. It is suited for either an algorithms or computational geometry course. We also briefly outline how other courses are using the lab. In section 6 we summarize what we've learned from the first semester of production use. We conclude with a description of our future plans both for the lab and for educational computing on campus in general.

†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. Equipment support was provided by NSF Grant SER80-04974 and by Apollo Computer, Inc. Support for the second author was provided by NSF Grant MCS-83-08806.

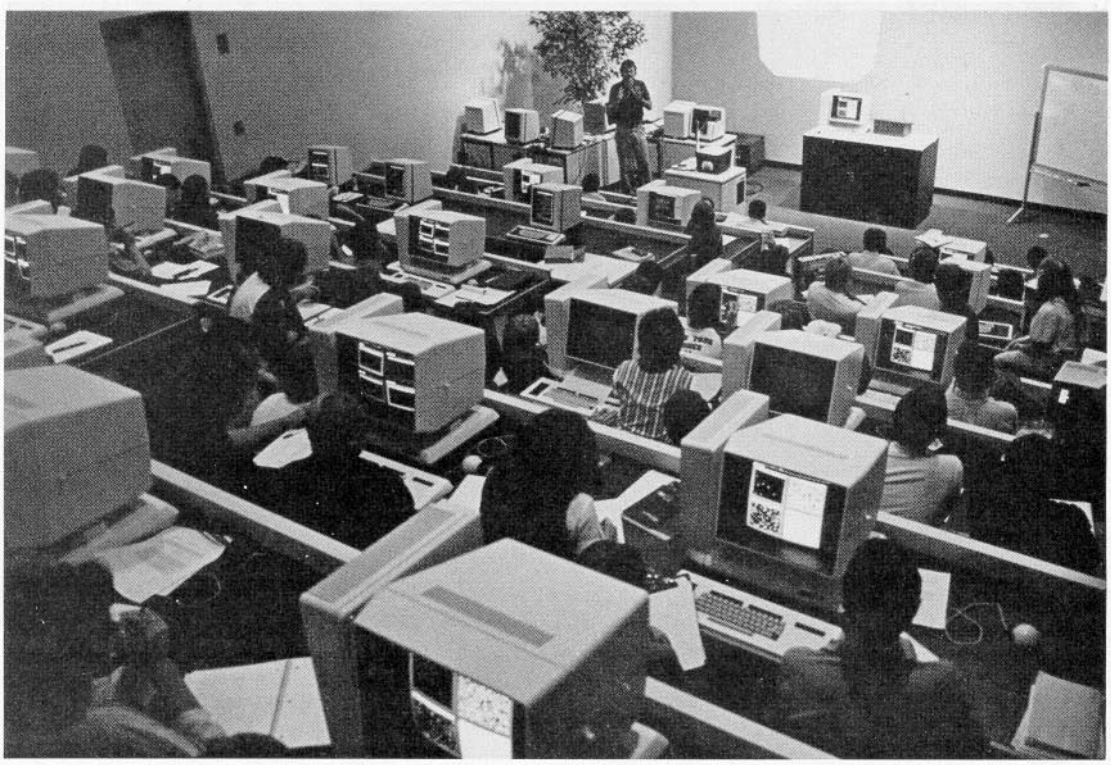

Figure 1 – *Foxboro Auditorium at Brown University*

2. Foxboro Auditorium

Foxboro Auditorium, shown in Figure 1, was designed and specially built in 1982 to house the Department's Instructional Computing Environment. There are eight tiers containing seven personal workstations each, plus the front podium containing the instructor's machine. During class, two students sit at each workstation; outside of the classroom, students use machines individually.

Each personal workstation is a powerful Apollo DN300 featuring a Motorola 68010 processor with 1.5MB of main memory. There is an additional 128KB of memory dedicated to the 1024x800x1 resolution bit-mapped display, and a hardware "bit-blt," capable of moving rectangular regions of memory at the rate of 32 Mbits per second. Each machine has a 3-buttoned mouse for graphical input. Groups of five nodes share a 158 MB Winchester disk, which is attached to a DN400 server. All nodes are connected by the Apollo proprietary token-passing ring network running at 12 MHz. The operating system is Unix System III, running on top of Apollo's Aegis OS. There is a 96-bit network-wide virtual address space with a 32-bit virtual address space for each system object (e.g., file) built on a demand paging system. There is a completely transparent naming space over the network, complete with both hard and soft links.

The Department owns a total of sixty-three DN300's dedicated to the instructional environment with almost 2 gigabytes of disk storage. The Department is also using Apollo computers for much of its research efforts into graphics-based workstation environments (see [Pato83] for details). Each of the eleven faculty members and many staff and graduate student offices are equipped with an Apollo DN300. These machines are more powerful than those in the lab, as they each have 2MB of memory and a private 33MB Winchester disk. Each is equipped with either a mouse, data tablet, or touch pad. We own about twenty-five such research machines.

We have two full-time employees and one part-time student who are responsible for the maintenance and installation of all hardware. On the software front, we have a full-time system administration

and four part-time student helpers. Full-backups are done weekly, and incremental-backups are done daily by our staff. Files to be printed are transfered to the University's main computing facilities via the Brown University Network, BRUNET, a 300 MHz broadband cable connecting all 125 buildings on campus. The Apollo network communicates to the Department's research computers (Vax 11/780) by various 9600-baud serial links and an Ethernet hi-speed link.

The laboratory is open to students daily from 9:00 am to 2:00 am, and for the entire night during "crunch" periods. Whenever the lab is open, there is a student consultant on duty (to answer non-course-specific questions and to do first-level trouble shooting of faulty hardware) and a student waitlist monitor (to check student id badges and to administer the waiting list). We have used a simple "first-come, first-serve" policy with a one hour maximum time limit rather successfully, although we spent many days debating the relative merits of a sundry of more complex algorithms.

With the lab operating in "production mode" during the fall of 1983, it was used during lectures for our two largest courses, and just about all student programming assignments in the Computer Science Department were done using the equipment in the laboratory. The artificial intelligence course (about 70 students) uses a dialect of LISP called T which was developed by Yale University. The systems programming course (about 50 students) uses both Pascal and C, in addition to much of the local software developed for facilitating the creation of interactive graphics programs. The operating systems course (about 60 students) use a Motorola 68000 simulator developed locally. In addition, an advanced mathematics course on differential geometry and the introductory neural science course use the lab on an regular basis for interactive classroom demonstrations.

The two courses which actually use the laboratory for interactive demonstrations during the class are the first semester introductory programming course (about 200 students) and the third semester algorithms and data structures course (about 175 students). In the introductory course, the laboratory is used in the majority of the lectures, and demos cover all programming constructs and data structures taught. In the algorithms course, the laboratory is used integrally in all of the lectures. Both of these courses meet in two sections of about 90-100 students. The students in these courses also use the laboratory for their programming assignments.

3. The Courseware/Software Environment

BALSA (Brown University Algorithm Simulator and Animator) is our software environment for the instructional computing laboratory. Our basic design philosophy was to develop a system that would allow a user to see and manipulate his programs by providing pictures of his data structures that mimic the representation that he uses himself. Since computer programs are inherently dynamic objects, dynamic pictures must be used to illustrate the operational characteristics of programs.

We set out to develop a comprehensive environment that would support the systematic animation of programs in a classroom environment. Our primary goals were are follows:

- It was desired that the instructor animating a program just write code for the specific algorithm being animated. The demo writer should not need to write any utilities, user-interaction routines (other than specifying a menu table to a front end module), or even pictorial representations of common objects.
- It was desired to have one common set of familiar, "standard" controls. Thus, once students learned how to interact with the system in one course, they would be able to use the system in the same way for all other courses.
- It was desired to build a rich set of common tools for displaying and interacting with common data. For example, after some basic primitives are developed for displaying a binary tree, the myriads of programs which use a binary tree data structure could readily be animated. Primitives might include displaying a static tree, pruning and moving subtrees, and adding new nodes.
- It was desired to research various techniques for animating algorithms. For example, what is the best way to animate a convex hull algorithm? matrix multiplication? file compression?
- It was desired to build a very robust front end that would be able to accommodate all user-interaction and program control with modern techniques, exemplified by the various Xerox environments.
- It was desired to have a rich set of common "utilities." Utilities include getting hardcopy of screen, saving a snapshot of the "world," and playing back a previous session.

BALSA graphically displays multiple views of the user's program and data structures. Each view is automatically maintained during execution to give the user a motion picture of his program in action in

the terms which make it easiest to understand. The user controls which views are active, as well as the size and location of windows in which views are displayed. Windows are rectangular regions of the screen that can overlap without interference. The user can step through a program both forward and backward at any speed he sets, mark a point during the execution to return to later, set and clear breakpoints, and interrupt an executing program.

The user can both zoom and scroll through the graphical contents of a view. In addition, it is easy for an experienced programmer to write additional views to provide alternative representations. Thus, by having multiple views of a given data structure with different panning and zooming factors, one can simultaneously study it from both global and local perspectives. (See Figures 5 and 6, in particular.)

One very useful utility in BALSA is the ability for one user to "broadcast" his display to one or more other users; the designated recipients then see on their screen exactly what the broadcaster is doing on his screen. This can be used in the laboratory in a number of modes: first, the instructor can show his screen to students to introduce concepts. Then, students can continue using the screen that was broadcast to their station to work further on the new concepts. Finally, the instructor (or teaching assistants) can "monitor" onto students' displays, to get feedback on how each student is mastering the subject matter. Other utilities include saving a replayable, user-editable history of a user session, and making a hardcopy bitmap of the screen.

4. Examples

In this section, we illustrate the use of the laboratory through several examples. The first, on binary trees, is suited for either an introductory programming course or a course on algorithms and data structures. The second sample lecture is on range searching using an extension of binary trees called 2-D trees and is geared for a beginning or advanced course on algorithms and data structures or a graduate course on computational geometry. The third example illustrates the computation of the transitive closure of a directed graph. It is also suited for an algorithms or combinatorics course. These three examples, Figures 2–9, have all been developed in BALSA (we've deleted the logo and prompt areas from most of the figures in order to conserve space). The fourth example is from a course on differential geometry, and the last example is from an introductory neural science course.

In the following descriptions, we've concentrated on explaining what the screen images are and what can be done with them, rather than how the displays are actually used in the classroom. Typically, concepts are introduced with each student's computer mimicking the instructor's; then students are given an opportunity to execute the programs at their own pace and with their own data or preassigned data, often including the sample data presented in the textbook.

4.1 Sample Lecture – Binary Trees

The motivation for this lecture is to develop a program which will maintain the information in a phone book containing 1 million entries. One must be able to insert new entries and search for entries relatively efficiently. We first illustrate what *efficient* means by comparing the values of N, $N/2$, and $\log_2(N)$ for $N = 1,000,000$ on a computer capable of performing 100 operations per second. We present three algorithms: (1) maintain an unordered array, and then search sequentially through it; (2) maintain a sorted array, and also search it sequentially; and (3) maintain a sorted array, but use a bisection algorithm for the search. Figure 2 shows the computer screen

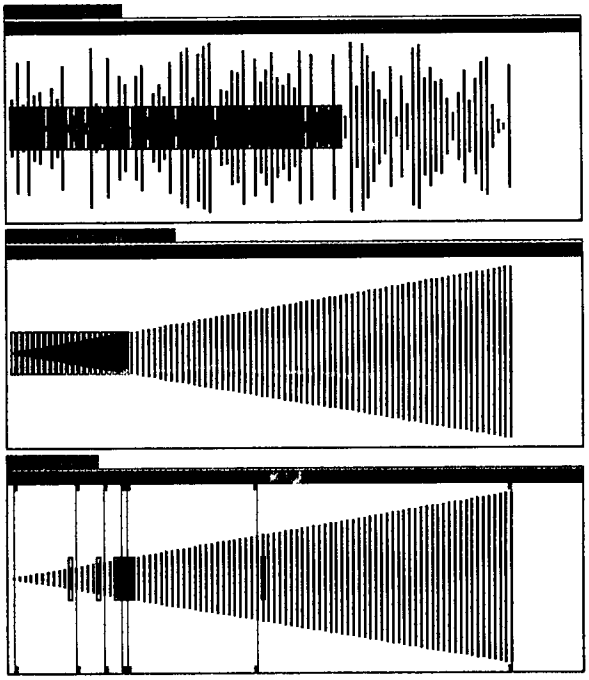

Figure 2 – *Array Searching Methods*

after a simulation of the searching aspect of the three methods.

There are three large rectangular regions on the screen, corresponding to the three different algorithms. Each algorithm window contains a graphical view of the algorithm's data structure, the array of keys. Each element of the array is displayed as a stick; the larger the stick, the larger the element of the array. Each array contains 90 elements; the white space at the right of each view indicates that the array has been defined to hold at most 105 elements. The thin box surrounding a stick indicates that we "probed" that element of the array in searching for the desired key. The size of the surrounding box indicates the value of the element for which we are searching. We are searching for an element which is about one-quarter of the way between the minimum and maximum values. The bottom view, of bisection, has some large thin vertical lines. These lines represent the left and right "pointers" in the search. Each "pointer" line has a little box at the top and at the bottom indicating whether it is a left or right pointer.

The reader must bear in mind that this figure, like all the others we will show, is a static snapshot of a dynamic processes. When the simulation is actually run, one sees the three algorithms running in parallel. The surrounding boxes are drawn as each corresponding element is probed. By watching these three searching methods students get an intuitive feel for the relative speeds of the methods.

An informal analysis of the relative running times, for an array containing N elements, is as follows: the top algorithm (sequential search in an unsorted array) takes N probes if the key is not already in the array, and about $N/2$ if the key is in the array; the middle algorithm (sequential search in a sorted array) requires about $N/2$ probes for both a successful and an unsuccessful search; the bottom algorithm (bisection searching of a sorted array) takes about $\log_2(N)$ probes for both a successful and unsuccessful search.

We next run a simulation of the insertion aspect of these three algorithms to illustrate the amount of data movement required. The unsorted array requires no data movement because the new key is inserted at the end of the array, whereas the sorted array (bottom two algorithms) requires $N/2$ elements (on the average) to be moved to the right in order to make room for the new key.

The students quickly realize that bisection is the appropriate searching method, but an unordered array is best for inserting. Since linked lists were used in a previous class to maintain order but eliminate data movement, we desire a linked list type of a structure which can supported bisection. We present Figure 3 as a solution. This diagram clearly illustrates some of the basic properties of binary trees. Each node has three parts: the key (the letter of the alphabet), a left subtree (which may be null for nodes such as R and a), and a right subtree (which may also be null for nodes such as a and g). All nodes follow the rule that the left son is lexicographically less than the node and the right son is lexicographically greater than the node.

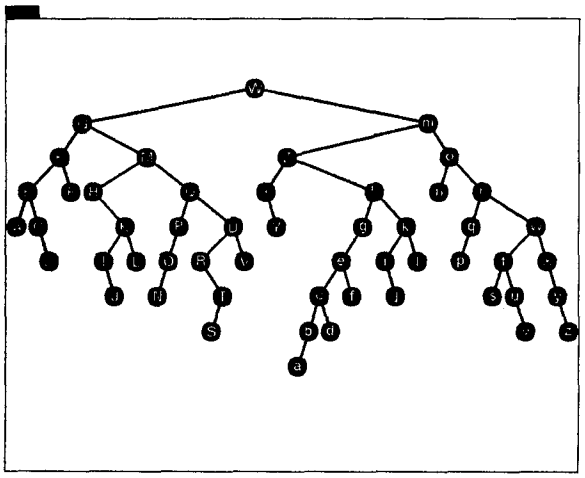

Figure 3 – *Binary Tree Built with 52 Random Keys*

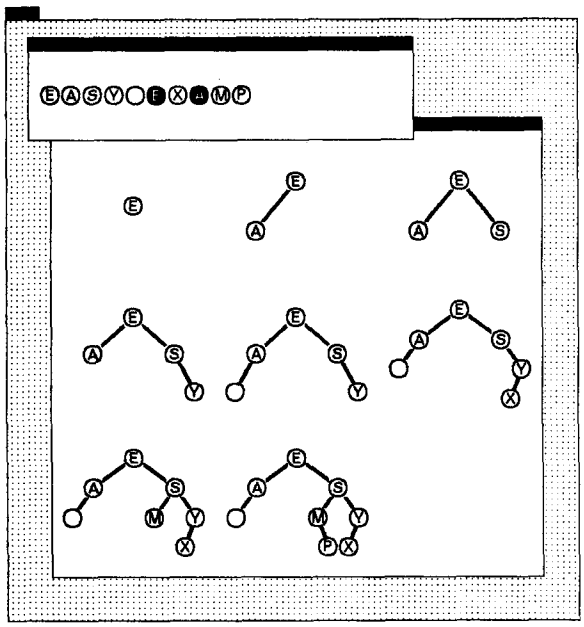

Figure 4 – *History of Building a Binary Tree*

Figure 3 also illustrates the fact the binary trees are not perfectly balanced. For example, the right son of Z contains many more elements than its left son.

Figure 4 illustrates the history of the tree as it is built from the letters E, A, S, Y, , E, X, A, M, P, L, E (note the blank space following the letter Y). The view in the upper left corner is a view of the input. As a letter to be added to the tree is read, it is displayed in this window. If the letter is successfully added to the tree, it is encircled; otherwise (the letter already appeared in the tree), the node is displayed as a filled circle.

The next part of the lecture is to explain the code needed to search for a node and to add a node to a tree. In an upper level algorithms course, it would suffice to describe the method at the pseudo-code level; in an introductory programming course, we would use Figure 5 to illustrate the actual Pascal implementation.

The current line of code that is being executed is highlighted in the large window on the left. Each procedure is displayed in its own overlapping window. Thus, in our example, we see that the `Mainline` called procedure `Insert` which then called the current procedure, `Lookup`. As each line of code is executed, the graphical results of that line are displayed in each of the other windows. The view of the tree with nodes in rectangular boxes also displays the current values of the two key variables, `PrevPtr` and `CurrPtr`. Below that, there is another view of the tree in order to give students a better perspective. The window in the upper left displays a status message describing the state of the program.

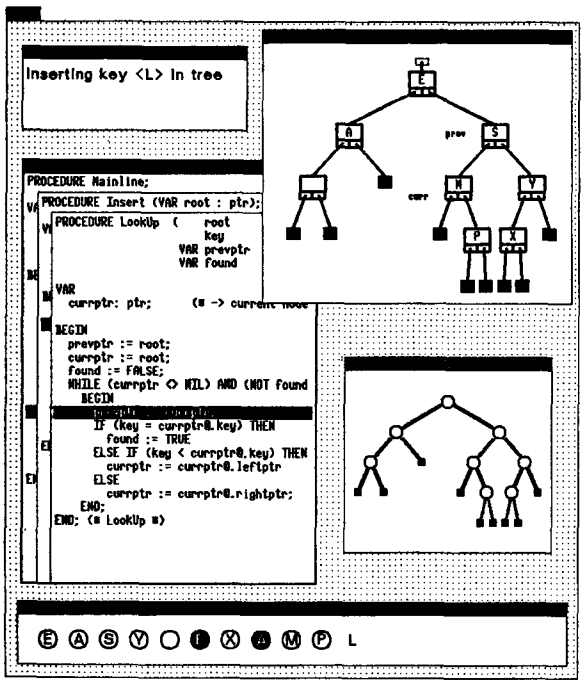

Figure 5 – *Inserting* L *into a Binary Tree*

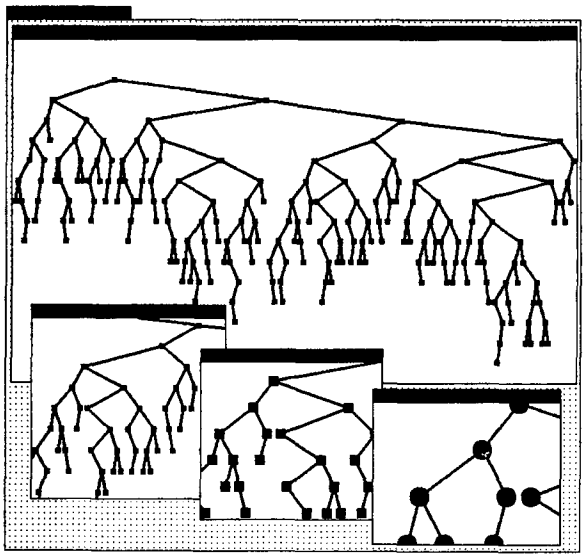

Figure 6 – *Multiple Views of a Large Binary Tree*

When the algorithm is run, the pointers chase down the tree, as expected. When procedure `Lookup` returns, `CurrPtr` will disappear because it is a local variable, and `PrevPtr` will change its name to the name of the parameter used by `Insert`. In addition, the window containing the code for `Lookup` will be removed and the code for procedure `Insert` will then be visible. The dark rectangle in the `Insert` procedure, next to the `BEGIN` statement in `Lookup` is the actual line in `Insert` which called `Lookup`.

The lecture concludes by running the algorithm on a large set of data. Figure 6 shows the tree which is build after 200 random elements are inserted into it. The three small views at the bottom show various levels of zooming into the the left subtree of the root's rightmost grandchild.

4.2 Sample Lecture – Range Searching

The problem that this lecture addresses is to develop a data structure (and appropriate algorithms) to efficiently determine which points from a set of N points fall within an arbitrary rectangular region in the plane. (See Chapter 26 in [Sedg83] for more details.) The strategy we'll use is to build a binary

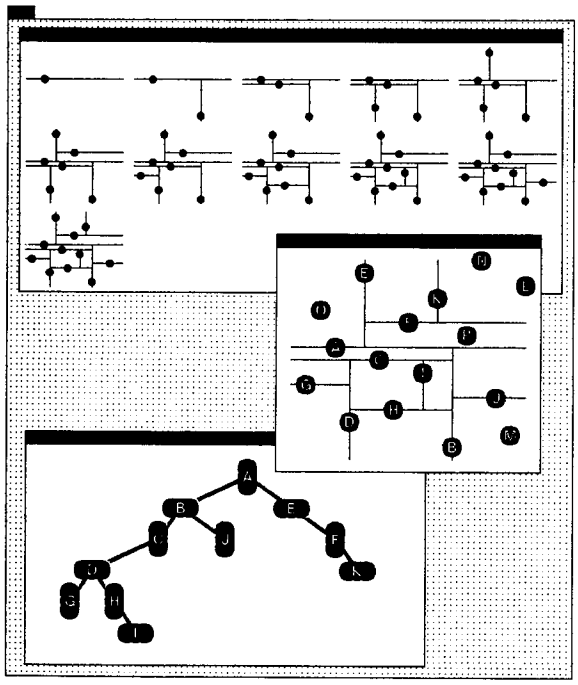

Figure 7 – *Building a 2-D Tree*

tree with the following property: at the root, use the y value to discriminate between the two sons. At the next level, use the x value. At the next, the y value. And so on. Figure 7 displays the resulting tree after the first eleven points have been inserted. The view in the middle is of the points in the plane, with thin horizontal and vertical line indicating which points serve to discriminate vertically and horizontally. The bottom view is the 2-D tree as it is being built. The top view is a history of the plane view after each point has been considered.

This diagram illustrates dramatically the properties of a 2-D tree: the right subtree of root node A contains only points above A; the left, those points which fall below A. The horizontal line at point A shows that it divides the plane into two horizontal slabs. There is a vertical line through point B to indicate that it is a horizontal separator. Nodes in the left subtree of B correspond to points which are below A and to the left of B (namely, points C, D, G, H, and I). After the tree is completely built, the right subtree of F will contain nodes corresponding to those points which are above A, to the right of E, and above F (namely, points K, L, and N). The nodes in the tree are horizontal and vertical to indicate whether the corresponding point is a horizontal or vertical separator.

The algorithm for performing the range search is a simple extension of an elementary recursive tree walk. The idea it to traverse the tree but prune it judiciously along the way. We test the point at each node in the tree against the range along the dimension that is used to divide the plane of the node. For example, in Figure 7, when we come to node A, we would test whether the range to be searched is above point A or below it. If above, we only look at the right subtree; if below we only look at the left subtree. If point A were inside the search range, then we would need to look at both sons. (This happens infrequently for reasonably small ranges in a large set of points). Figure 8 shows the searching algorithm after it has finished running on a larger set of points. Note that each node changes shape to indicate its status. Initially, all points in the plane are circular and all nodes in the tree are oval (see Figure 7). When a node has been visited in the traversal it becomes rectangular in the tree and square in the plane. Rectangles and squares are filled if the point is within the desired range.

The view above the tree illustrates the actual probes. Each point is drawn in the same style as in the plane view (i.e., initially circles, and becoming filled and hollow squares). Each point has a vertical line sticking out: the top stick represents the value of the x coordinate of the point; the bottom, the y coordinate. For each point, one stick is thin and the other stick is thick. The thick stick indicates whether

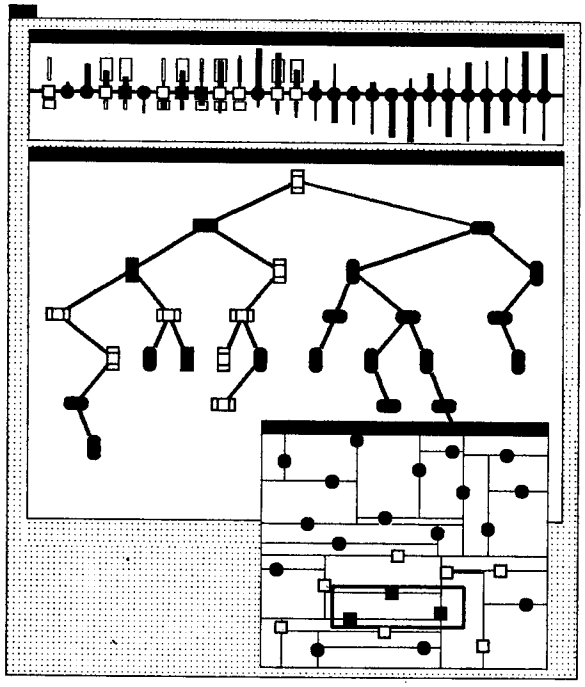

Figure 8 – *Range Searching in a 2-D Tree*

the point is used in the 2-D tree as a horizontal or a vertical discriminator. The desired range is shown as a surrounding box: the upper box indicates the desired horizontal range; the lower, the vertical range. The reader may be amused to compare this view with those in Figure 2.

4.3 Sample Display – Transitive Closure

The transitive closure of graph is the matrix which indicates whether there is a path (of any length) between each pair of vertices. Figure 9 shows the computation of the transitive closure of a directed graph containing thirteen points. The view in the lower left is of the adjacency matrix representing the graph. A non-white space in the row for point R and the column for point C indicates that there is an edge in the original graph from point R to point C. The algorithm that is illustrated here is this: consider each point on the graph in turn. Perform a depth first search (DFS) of the graph, starting at the given point. Each vertex that can be reached in the DFS belongs in the transitive closure matrix for the given point. The view of the graph in the top half illustrates the DFS starting at point I. Already, points H, G, J, and M have been visited (indicated by a thick edge and a square point rather than a circular point). The I row in the transitive closure matrix view (lower right) has a dot in the columns corresponding to points G, H, I, J, and M, indicating that these points can be reached from I (note that I can be reached from H). In both the adjacency matrix views, points C, E, K and L are hollow squares indicating that in the DFS, it hasn't yet been visited, but is scheduled to be visited. This is known as a point being in the "fringe."

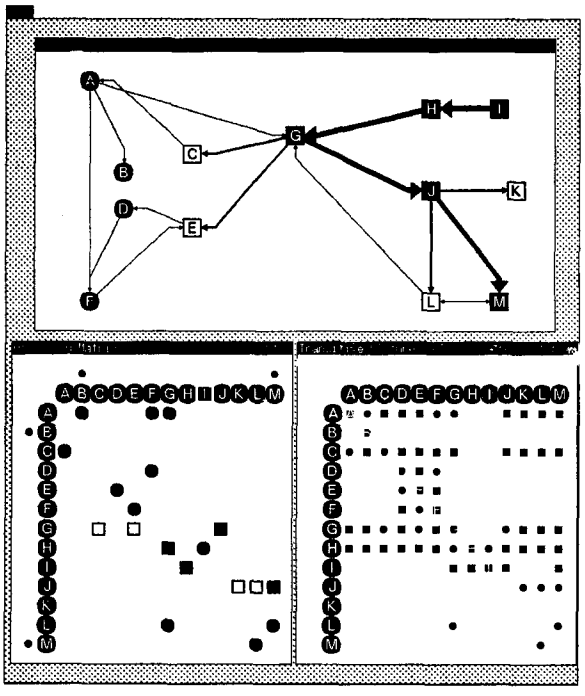

Figure 9 – *Computing the Transitive Closure*

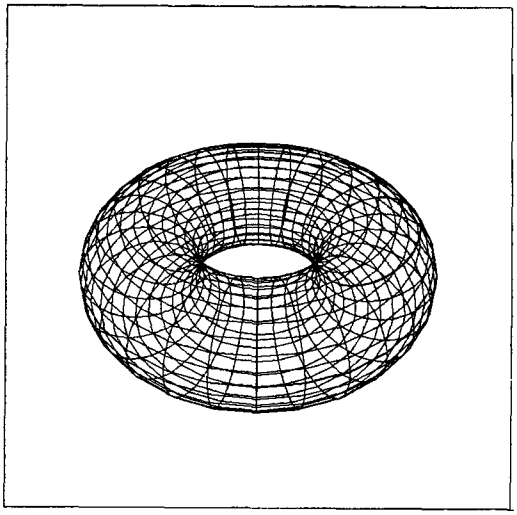

Figure 10 – *Perspective Projection of a Torus*

Of course, later in this lecture, we also include a dynamic display of the operation of the well-known Warshall algorithm for the same problem. This lecture is one of a series on graph searching, so students by this time are familiar with the basic concepts and the various graphical artifacts. Also, as usual, the real educational value is in the dynamics of the display, which we are obviously unable to show.

4.4 Sample Display – Differential Geometry

Figure 10 shows a typical display in the mathematics course on differential geometry. This courseware allows a student to define a parametric curve or surface and to modify its parameters. For example, while viewing the torus one can specify its two radii. In addition, to better understand a curve at each point, one can add a "tube." Typical tubes are the normal or tangent to a curve. Other tubes have been used which are based on the curvature or torsion at each point. Further, in order to truly visualize the 3-dimensional object, one can rotate it about the x, y, or z axes (or any combination thereof), and can

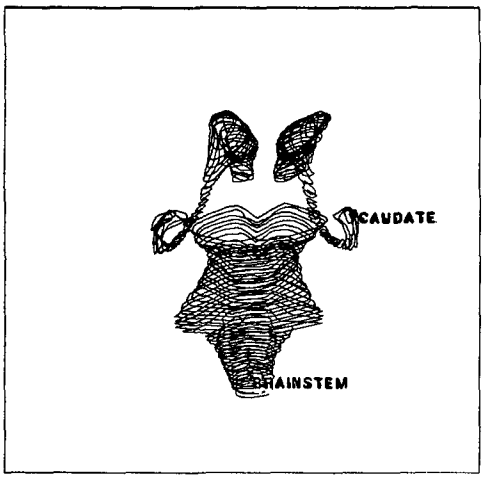

Figure 11 – *Brainstem and Caudate*

specify either a perspective or a parallel projection. This system has also been used to view 3-dimensional projections of 4-dimensional objects, such as the hypercube.

4.5 Sample Display – Neural Science

The introductory neural science courses use displays such as Figure 11 to help students visualize the structure of the brain. The program constructs a 3-dimensional image of the various structures of the brain by displaying serial cross-sections. It allows the student to rotate, translate, and zoom the image in order to get a better perspective of it. The user can select which of the many parts of the brain are to be displayed, get a brief online description of each part, and see the hierarchy of the components. The system has been used very successfully in introductory neural science where before students had to construct a mental image of the brain by studying many 2-D slices.

5. What Have We Learned?

In the introductory programming course, the most useful aspect of the demos was the ability to single step through the code and see the contents of the variables at each step, as well as the input and the output. Class demos were usually done in "broadcast" mode, with each student's screen mimicking the instructor's. Occasionally class demos were done in "real-time" mode, where each pair of students executed the demo by themselves at their own speed and with their own input data. Regardless of the mode by which demos were presented, we found that the students made demonstrable gains in speed of comprehension over that of the traditional lecture-followed-by-homework approach. Students felt that multiple dynamic views of the same phenomenon helped them to visualize the workings of a program much better than just stepping through the code on paper. Students enjoyed the graphical user-computer interaction, and developed sensitivity to modern user interface issues such as the need for prompting, self-disclosing popup menus and the ability to undo and redo user actions within a session.

In the algorithms and data structures course, the demos emphasized graphical displays of the data structures and the dynamics of the algorithm manipulating the displays. The class ran demos in "playback" mode, where each student replayed a script of keystrokes that the instructor had previously saved. The scripts had pauses at key points so that students would be able to keep up with the instructor. This method has the advantage that students can, during their free time, playback the exact pictures that they saw during class. In addition, during class, students can pause to consider a particular display and then "catch-up" (or even go ahead slightly, if they so desire).

In the algorithms course, each lecture required about fifteen hours of programming (by very experienced hackers), and about two hours for developing a "script." As the courseware writers became more familiar with the system, many lectures were programmed in less five hours. Even this amount of time is too high, and we believe that as BALSA matures, this time will go down dramatically.

Our environment is not designed as a CAI system. We do not have tools to measure student comprehension nor do we expect students to run the lectures for self-learning. The dynamic graphics is a tool for explaining and understanding complex, abstract, and dynamic concepts. We are able to show students things not possible before: they learn more than before and they learn it more effectively. Student response in the algorithms course to the question *What do you think of the demos?* was uniformly positive. We have not done a bona fide controlled experiment to measure the advantages of the laboratory setting, but our feelings are that the interactive graphics in the classroom is a significant advance over other pedagogical models.

6. Where Do We Go From Here?

The main area in which we plan to expand on our use of the system is in the exploration of new modes of interaction in the classroom environment which exploit the technical facilities that we have available. Most of our lectures are based on "broadcast" mode or on a "playback" mode where students "keep up" with the instructor through simple keystroke commands. There are capabilities in BALSA to allow us to do much more: for example, a five-minute period could be allotted in the lecture to allow each student to enter her own data, or to rerun complicated algorithms at her own pace or on a case that is of particular interest to her.

Now that we have a significant amount of courseware and a year's experience living in the environment, we expect to make significant progress in this area on our next pass through the courses.

Another thing that we would like is to be able to have students actually write or modify small programs in the class, and have their programs visually animated, as the demo programs are. We are currently interfacing BALSA with the PECAN program development system [Brow84]. PECAN represents programs as abstract syntax trees, uses a syntax-directed editor to manipulate pieces of the tree, and displays multiple views of the tree both graphically (e.g., as Nasi-Sneiderman diagrams, flow-graphs, parse trees) and textually [Reis83].

In the algorithms course, we have an interactive demonstration of each of the forty chapters in [Sedg83]. The areas we have covered include arithmetic algorithms (e.g., random numbers, Gaussian elimination, and curve fitting); sorting; searching; string processing (e.g., pattern matching, parsing, and cryptology); geometric algorithms (e.g., convex hull, range searching, and closest point); graph algorithms (e.g., breadth- and depth-first searching, weighted and directed graphs, connectivity); and advanced topics (e.g., algorithm machines, linear and dynamic programming, and NP-complete problems). We would like to use color and extended processor power to further improve the presentation of the information. This also involves significant research efforts into ways to display information never before seen. Already, we have used the courseware environment in research to develop and analyze new algorithms [Sedg84].

Another area that we'd like to improve on, although we might be limited by the hardware, is the speed needed to display the objects in say, the math and neural science environments.

Unfortunately, the cost of a laboratory such as the one we have described is still prohibitive for most universities. The list price of each workstation is about $20,000 (including the cost of the disk pro rated). All indications are that these prices will drop significantly in the next few years. In order to disseminate our courseware to other universities, we are preparing a videotape with guide. In addition, commercial companies are considering developing products modeled after our environment for the personal- and home-computer marketplaces.

This laboratory has been the impetus to establish Brown's new Institute for Research in Information and Scholarship (IRIS), an ambitious plan to equip the 10,000 members of the Brown community with workstations by the end of the decade [Ship83]. Within the next year, we expect to establish a number of similar laboratories devoted to other disciplines. Although all of the courseware which we will describe has been developed for the Apollo's, it has a layered design, and can be readily ported to other graphics-based personal workstations running a Unix-like operating system, such as Suns and PERQs.

7. Acknowledgements

Many people have helped to make the laboratory successful. In particular, Andy van Dam's tireless efforts have made the physical environment far more impressive than it otherwise would have been; he has also contributed to the project as the instructor of the introductory programming course. Tom Doeppner was also instrumental in procuring initial funding for the project.

David Durfee and Jeff Coady have been the men behind the scenes, ensuring that all hardware and vendor-supplied software is working correctly. Steve Reiss, Joe Pato, and Dave Nanian wrote significant pieces of the underlying software. Mike Strickman has been a key implementor of the courseware environment.

The mathematics viewing system was implemented by Eddie Grove and Rich Hawkes; the neural science viewing system by Steve Drucker. Both of these systems started as final projects for the graduate graphics course and were converted into production quality during the summer of 1983. Perry Busalacchi, Karen Smith and Liz Waymire ran experimental versions of the introductory programming course in the spring of 1983 (with 20 students) and in the summer of 1983 (with 60 students) under Andy van Dam's supervision, and wrote much of the

courseware that is still being used in the introductory programming course. Kate Smith Greenfield has helped out in developing courseware for the algorithms course.

Special thanks goes to Janet Incerpi whose help in the preparation of this paper was invaluable.

8. References

[Brow84] Brown, Marc H. and Reiss, Steven P., "Toward a Computer Science Environment for Powerful Personal Machines," in *Proc. of the 17th Hawaii International Conference on System Sciences*, January 1984.

[Brow83] Brown, Marc H., Meyrowitz, Norman and van Dam, Andries, "Personal Computer Networks and Graphical Animation: Rationale and Practice for Education," *ACM SIGCSE Bulletin* **15**, 1 (February 1983), 296-307.

[Pato83] Pato, Joseph N., Reiss, Steven P., and Brown, Marc H., "Brown University Workstation Environment Summary Paper," Technical Report, Brown University, Providence, RI, 1983.

[Reis83] Reiss, Steven P., "PECAN: A Program Development System that Supports Multiple Views," Technical Report, Brown University, Providence, RI, 1983.

[Ship83] Shipp, William S., Meyrowitz, Norman and van Dam, Andries, "Networks of Scholar's Workstations in a University Community," in *Proc. of the IEEE Compcon Fall 1983.*

[Sedg84] Sedgewick, Robert and Vitter, Jeffrey S., "Shortest Paths in Euclidean Graphs," Technical Report, Brown University, Providence, RI, 1984.

[Sedg83] Sedgewick, Robert, *Algorithms*, Addison-Wesley, Reading, MA, 1983.

The following references were not cited directly. However, they are very good background reading and viewing for program visualization and animation.

[Baec75] Baecker, Ronald, "Two System Which Produce Animated Representations of the Execution of Computer Programs," *ACM SIGCSE Bulletin* **7**, 1 (February 1975), 158-167.

[Baec81] Baecker, Ronald, "Sorting out Sorting," 16mm color sound file, 25 minutes, 1981.

[Hero82] Herot, C., et. al., "An Integrated Environment for Program Visualization," in *Automated Tools for Information Systems Design*, H.J. Schneider and A.I. Wasserman, Ed., North Holland Publishing Co., 1982, pp. 237-259.

[Know66] Knowlton, Ken, "L6: Bell Telephone Laboratories Low-Level Linked List Language," two black and white sound films, 1966.

[Myer80] Myers, Brad, "Displaying Data Structures for Interactive Debugging," CSL-80-7, Xerox PARC, Palo Alto, CA, 1980.

[Plat81] Plattner, Bernhard and Nievergelt, Jurg, "Monitoring Program Execution: A Survey," *Computer* **14** (November 1981), 76-93.